Abstract: This dissertation examines the problem of designing large scale multi-robot systems for pursuit-evasion tasks, which involves the detection of all targets initially located in some environment of interest. We consider targets that are omniscient and have unbounded speed. Robots, however, are very restricted in their capabilities and have only limited sensing and communication range. We develop theory to describe the problem and algorithms to coordinate a large team of robots to solve the pursuit-evasion task.

One main contribution is a rigorous graph model of multi-robot pursuit-evasion, called Graph-Clear, complementing existing literature on graph-searching. We determine its complexity and provide algorithms and extensions for a variety of scenarios. A second contribution is a model for multi-robot pursuit-evasion in two dimensional environments, called Line-Clear, that abstracts the sensing capabilities of the robot team to the ability to sense on lines between obstacles and thereby detect targets. We present terminology and algorithms that enable the coordination of the movement of such lines while attempting to minimize the number of robots needed to cover these lines with sensors. To improve the applicability of the proposed models we also present two automated methods that extract instances of Graph-Clear and Line-Clear from grid and polygonal maps. These methods are then combined with the algorithms for Graph-Clear and Line-Clear to enable the coordination of real and simulated robots for the detection of all targets within sample environments. We also extend the approach to an online version that does not require a map of the environment and works with simple robots, imperfect control, no localization and limited communication range.

University of California
Merced

Multi-Robot Pursuit-Evasion

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Electrical Engineering and Computer Science

by

Andreas Kolling

2009

The dissertation of Andreas Kolling is approved.

Songhwai Oh

Steven M. LaValle

Alberto Cerpa

Stefano Carpin, Committee Chair

University of California, Merced

2009

Dedicated to my parents,

Angelika and Elmar.

Table of Contents

1 Introduction **1**

1.1 Multi-Robot Pursuit-Evasion 3

1.2 Goal and Overview of Contributions 4

2 Related Work **7**

2.1 Pursuit-Evasion on Graphs 9

2.2 Visibility-Based Pursuit-Evasion 13

2.3 Probabilistic Graph Searching 18

2.4 Direct Pursuit 20

2.5 Cooperative Multi-Robot Approaches 21

2.6 Coverage and Clearing 24

2.7 Pursuit-Evasion and Control Theory 24

2.8 Probabilistic Approaches 28

2.9 Sensor Networks and Tracking 30

2.10 Real Systems 33

2.11 Other Related Fields 37

3 Graph-Clear: Multi-Robot Pursuit-Evasion on Graphs **40**

3.1 Motivation 43

3.2 Problem Formulation 47

3.2.1 An example of Graph-Clear 52

3.3 The Complexity of Graph-Clear 55

3.4 Recontamination for Optimal Strategies 62
3.4.1 Cuts 62
3.4.2 Recontamination does not help 64
3.5 Label-Based Strategies on Trees 69
3.6 Optimal Contiguous Strategies on Trees: a Polynomial Algorithm 79
3.6.1 Full cut sequences 79
3.6.2 Constructing cut sets 85
3.7 Improved Non-Contiguous Strategies: Hybrid Algorithm 91
3.7.1 Batches 93
3.7.2 Criteria for optimal partitions 94
3.7.3 The partitioning algorithm 97
3.7.4 Discussion and Conclusion 107
3.8 Probabilistic Graph-Clear 107
3.8.1 Probabilistic Model 109
3.8.2 Modeling faulty sensors 113
3.8.3 Probabilistic Extensions to Graph-Clear 116
3.8.4 Discussion and Conclusion 122
3.9 Modified Graph-Clear: Sweeps Prevent Recontamination 123
3.10 Applying Strategies to Graphs 126
3.11 Discussion and Conclusion 128

4 Line-Clear: Multi-Robot Pursuit-Evasion in 2d 131

4.1 Line-Clear: Definitions 133

4.1.1 Covering Sweep Lines with Sensors . . . 139
4.2 Sweep Schedules through Graph-Clear . . . 140
4.2.1 Voronoi Diagrams . . . 141
4.2.2 Constructing Sweep Schedules from Voronoi Diagrams . . . 144
4.2.3 Optimality Considerations . . . 154
4.3 Reduction to a Combinatorial Problem . . . 157
4.4 Finding Optimal Sweep Schedules for Simply-Connected Environments . . . 163
4.4.1 Split Points for Sweep Lines . . . 176
4.5 Discussion and Conclusion . . . 177

5 Extracting Surveillance Graphs From Maps . . . 180
5.1 Voronoi-based Extractions . . . 180
5.1.1 Blocking . . . 182
5.1.2 Vertex sweeping . . . 183
5.1.3 Initial Graph Construction . . . 184
5.1.4 Improving the graph . . . 187
5.1.5 Implementing blocking and sweeping actions . . . 188
5.1.6 Experimental Results . . . 189
5.1.7 Discussion and Conclusion . . . 193
5.2 Line-Clear Extractions . . . 196
5.2.1 Implementation . . . 197
5.2.2 Experiments . . . 198

5.2.3 Discussion and Conclusion 204

6 Applications and Experiments 207

6.1 First Experiments with Simple Sweeps and Two Robots 208

6.1.1 Extracting Surveillance Graphs 208

6.1.2 Implementing Surveillance Graph actions 209

6.1.3 Experiment Design 214

6.1.4 Results and Discussion 216

6.1.5 Conclusion 217

6.2 Line-Clear without Maps 218

6.2.1 Bootstrapping 221

6.2.2 Moving a Sweep Line 223

6.2.3 Splitting a Line 229

6.2.4 Obstacle Search 230

6.2.5 Surveillance Graphs and Line Coordination 231

6.2.6 Implementation and Testing 240

6.2.7 Discussion and Conclusion 247

6.3 Discussion and Conclusion 252

7 Discussion and Conclusion 254

References 256

List of Figures

1.1 An overview of the contributions of this dissertation. 6

3.1 An example that illustrates how a graph for Graph-Clear can relate to an actual environment. The environment is shown in grey with its graph embedded. All weights in this example are equal to one. Connections between regions that are connected by edges are shown in black. The center region is the "eagle" example redrawn from [SRL04]. It can be cleared using the algorithm from [SRL04] with only one robot and a simple gap sensor with sufficiently large range. During its execution it recontaminates the top part of the region and hence cannot guarantee that no target enters the vertex undetected. We hence need blocks on the edges, i.e. to position sensors on the black regions. Note that the entire environment can be very large so that the sensor only satisfies the large range assumption within a vertex. 46

3.2 An example environment and one possibly associated surveillance graph. Numbers inside vertices are the sweeping costs, and numbers on the edges are blocking costs. 53

3.3 A possible strategy to solve the Graph-Clear problem associated with the graph shown in Fig. 3.2. The first column displays the status, the second the applied action, and the third the cost. The reader should note that in the third row an action sweeping two vertices at the same time is applied, and that a final action removing all blocks is executed in the end (with 0 cost). The cost of this strategy is 12, i.e. the maximum value read in the third column. It is easy to see that such strategy is not optimal. 54

3.4 An example that illustrates the consequences of allowing simultaneous moves in weighted edge-searching. Part a) shows a graph with its weights. Part b) shows the graph with eight robots on the top vertex and none in the bottom vertices. The arrows indicate two sliding moves with four robots that finish clearing the graph with eight robots when executed simultaneously. Part c) shows how to clear the graph with strictly sequential moves with the same recontamination rules but needing more robots. 56

3.5 The construction of an optimal strategy for a star. Cleared and contaminated vertices are grey and white respectively. Blocked edges are marked with as double-stroked line. First all leaves are cleared, leaving the edge to the leaf blocked. For leaf v_i,$i = 1, \ldots, n$ the total cost while clearing it is $i + 1$. Finally the center vertex is cleared with cost $n + 1$. 57

3.6 An illustration of the large graph constructed from an instance of the MCIESS. Part a) shows the constructed surveillance graph from the MCIESS graph in part b). A star is represented by a cloud, a bundle of nN or more edges by a double line and 3 edges by a thick line. Part c) is a close-up of the star C_1 and its edges to other star. In part d) C_1 is shown in more detail with its center, connectors and leaves. 59

3.7 A contiguous strategy on a tree is executed based on the labels on edges. Blocked edges are crossed through twice, cleared vertices are gray. A vertex with dashed lines attached represents an entire subtree rooted at that vertex. A subtree being cleared is marked with the corresponding root vertex drawn in thick dashed lines. The label associated to this procedure is shown in a) with the direction of the robots marked by an arrow. 72

3.8 This is the worst case example for $d = 6, s_{max} = 6$ leading to a worst case label cost of 15. Blocking weight w is on the left and label on the right of the colon for every edge. Each vertex has its weight in its center. Only labels for the direction towards the leaves from the vertex marked with a black arrow are shown. . . . 77

3.9 A comparison of the average upper bound across 1000 weighted trees and actual maximum label values for varying number of vertices. 77

3.10 Given v_y we define subtrees T_i as seen in the figure. 80

3.11 Illustration of the cut sequences associated to edges and the subtrees involved in the construction. There are three possibilities for $\bar{\mathcal{S}}_{v_x}(e)$, depending on the costs of the cuts. E.g. if weights on the tree are s.t. executing $\{v_x, v_y\}$ costs as much as executing the full cut $V(T_y)$ right away, then the first possibility is the full cut sequence. Otherwise, if $\{v_x, v_y\}$ costs less and has smaller blocking cost than $\{v_x\}$, then the second possibility is the full cut sequence, and so on. 86

3.12 Execution of the hybrid strategy. 91

3.13 The dynamic programming table for the example from table 3.3. . 103

3.14 Possible partitions resulting from the dynamic programming table for the example from 3.3. A partition is represented by a sequence of arrows where a diagonal arrow means that the vertex of the row to which the arrow is pointing is in V_2 while a horizontal arrow indicates that the vertex is in V_1. 105

3.15 A grid with a sensor placed in its center as a black circle and with the cells observed by the sensor in grey. A darker grey tone denotes a smaller false negative probability. 114

3.16 An illustration of the computation of detection probabilities for blocks through the worst case path a target can take through a block. 116

3.17 The basic block in this figure only needs one robot, while the first reinforcement leading to an improvement in the detection capability needs two additional robots. In order to get this fact the reader should consider that intruders may also move diagonally on the grid.117

3.18 An example of algorithm 7 121

3.19 Adding a virtual edge to compute the cost of clearing G starting from v_3. The lambda function on the virtual edge, represented as a dotted line will represent the probability of clearing everything beyond that edge. 122

3.20 Results of the experiments with 9 sets of parameters. The upper lines for each number of vertices is always the cost for the constant cycle blocking strategy and the lower for the dynamic cycle blocking strategy. 128

4.1 Examples of points that are valid sweep lines a), b) and c). The remaining lines are invalid and not sweep lines. 135

4.2 Multiple robots covering a sweep line between two obstacles. As the distance between the obstacles grows another robot is added at the appropriate location. 140

4.3 An illustration of the definitions of surjective surfaces and equidistant faces. The surjective surfaces are drawn with thin lines and the equidistant faces, a subset of the surjective surfaces, are drawn in thick and dashed lines. 142

4.4 The GVG vertices and edges in the environment marked as circles and dashed lines, respectively. Note that some edges are continuing on the intersection of two obstacle boundaries, but not all intersections lead to an edge as seen in the corridor to the left. To form a proper graph an additional vertex at infinity could be introduced and connected to these edges. 144

4.5 Illustration of the concept of moving and splitting sweep lines. The arrows indicate the direction of movement of the sweep line on the left side until it splits into two sweep lines which continue independently. 145

4.6 Left: A Voronoi Diagram resulting from line segments of multiple polygonal obstacles by considering each open segment and their endpoints as independent obstacles. Note vertices and edges inside the polygons are also drawn and result from the fact that the line segments are considered obstacles for the purpose of construction the Voronoi Diagram. Right: Conversion of the Voronoi Diagram into a surveillance graph. Dashed lines indicate lines that are associated to vertices and edges and represent blocks and sweeps. The movement of lines is represented by their thickness, i.e. thin lines move towards thicker lines. 146

4.7 A sweep line moving along an edge e_{ij} and crossing vertex v. The grey area is the cleared part $\mathcal{R}(t)$ bounded by obstacles and sweep lines. 147

4.8 Illustrating the forward movement of a sweep line across a vertex v. Part a) shows l_{ij} at time t_{block}, part b) at time t_{swap}, part c) at time t_{split} and part d) shows the two new moving sweep lines l_{ik} and l_{jk} at time t_{end}. 148

4.9 An example in which the minimum of d_i on $\mathcal{SS}_{ij}$, marked as a grey dot on a grey dashed line in a), does not lead to a valid sweep line. The blocking position is then earlier on $\mathcal{SS}_{ij}$ as seen in b). 150

4.10 Four cases for a vertex with degree three. Current sweep lines are black while future sweep lines that are to be reached are grey. Contaminated and cleared sides of current sweep lines are marked. In a) one sweep line splits into two sweep lines. In b) two sweep lines merge into one sweep line, the converse of a). In c) all points are still contaminated and two sweep lines are established to being clearing. In d) all points outside the figure are cleared and all sweep lines will disappear, the converse of c). Part a) represents three cases, one for each choice of direction for the current sweep line, i.e. either starting between C_i and C_j as seen or between C_i,C_k or C_j,C_k. Similarly, part b) also represents three cases. For c) and d) there is only one choice of directions, leading to overall eight possible sweeps for the Voronoi vertex associated to C_i,C_j,C_k represented by four weights, since each case has an associated inverse at identical cost. 153

4.11 The blocking positions of sweep lines in the environment are marked as dashed lines each creating an edge between two vertices which now correspond to a region as partitioned by the blocks. The Voronoi Diagram of the environment is presented in fig. 4.4. . . . 154

4.12 A six-way intersection constructed around a circle with diameter d and larger corridors given by parameter d_1. Values for d_3 and d_2 follow from the circles diameter. All center obstacles are aligned around the circle with the exception of the leftmost obstacle which is moved towards the center by $\epsilon > 0$. 155

4.13 An illustration of an optimal sweep schedule for the environment from fig. 4.12 given that $d_1 > d_2 + d$. 156

4.14 An illustration of the beginning of a sweep schedule created from the Voronoi Diagram of the environment from fig. 4.12 starting at v_8. Given that v_8 is the starting vertex, the next split is necessarily at v_3 as shown in the figure. 157

4.15 A simply connected environment with two sets of two sweep lines $\{l_1, l_2\}$ and $\{l_3, l_4\}$. Sweep lines l_1 and l_2 together with parts of obstacles C_1, C_3 and C_4 form the boundary of a cleared region for a sweep schedule. This boundary can be traversed by following C_1, C_3, C_4, C_1 and hence $B(t) = \{1, 3, 4\}$. The same traversal applies to a possible sweep schedule represented by $\{l_3, l_4\}$ and these represent the set of sweep lines with lowest cost that have this traversal sequence. 159

4.16 Adding an obstacle index twice to $B(t)$ at different times violates contiguity of the cleared part. 160

4.17 Three points on C_{o_i} and the sweep lines they form to C_{i_l} and C_{i_r}. 162

4.18 Example of two sweep schedules. The sweep on the left is suboptimal while the one on the right is the optimal solution. On the bottom of each side the evolution of the set $B(t)$ is shown. 163

4.19 An illustration of how a set of choices T_k^i splits into left and right sides, depending on which obstacle index in T_k^i is chosen. A choice is represented by an edge towards a left and right side with the chosen index written on the edge. 165

4.20 Part a),b),c) and d) show the environment in four different states with the respective sets of choice that exist in the respective state. 166

4.21 An example of a choice tree for $o_1 = 1$. The empty set T_0 is not drawn. The recursive construction is given in fig. 4.19 167

4.22 A compressed version of the tree structure from fig. 4.21. Every unique T_k^i is drawn exactly once. The number of edges grows as a polynomial with degree 3 in the number of obstacle indices. 168

4.23 Two different paths in the tree-like structure from fig. 4.22. Note that if a choice set splits into a left and right we need to follow both sides. Fig. 4.24 shows how these two paths lead to a surveillance graph. 170

4.24 The figure shows two surveillance graphs that correspond to the two paths chosen in fig. 4.23. The cleared vertex at the bottom represents the choice of o_1 as the first obstacle. 171

4.25 This figure shows how a choice set T_k^i leads to an edge with weight $b(T_k^i)$. The next vertex beyond that edge is determined by the choice made in T_k^i and all alternatives with their respective weights are shown. The next choices made in the choice sets for the left and right side create new vertices. The edges towards these are marked with dashed lines. 172

4.26 Illustration of the computation of labels $\lambda_j(e_k^i)$ and $\lambda(e_k^i)$ for $k = 1, 2, 3$. 173

4.27 Computing labels for edges representing choice sets implicitly prunes all outgoing choices from a choice set to one. The figure shows such a pruned tree and only the edges for choices are shown that correspond to a vertices that leads to the minimum label. 174

5.1 Illustrating the advantage of narrow connections between open regions. For robots with a limited sensing range the environment in part a) can be cleared with 3 robots, while the one on part b) requires 4. The reader should note that the surface is the same. . 184

5.2 A simple environment with a Voronoi edge in the center as a dotted line and the clearance function in the graph on top. The minima on the Voronoi edge are marked by grey circles. 185

5.3 A Voronoi Diagram and its minima. The Voronoi Diagram is marked with grey dashed lines, the minima with grey circles and the lines to the closest obstacle points with thin black lines. Obstacle boundaries are thick black lines. Corners in corridors tend to produce minima, unless a narrow part proceeds it as seen in the upper left corner of the figure. 186

5.4 An example in which a discrete approximation of the Voronoi Diagram in a grid map leads to introduction of unwanted edges. The black line in the center is the Voronoi Diagram edge and minima are marked by grey lines 187

5.5 A contraction of a vertex with degree two and its neighbor. Part a) shows the initial graph and part b) the graph after the contraction. 188

5.6 The map created by the P3AT at UC Merced with initial graph construction. The thick black lines are boundaries between free and occupied space. The small black circles are vertices placed in their corresponding region which are separated by thin lines. . . . 190

5.7 The sdr_site_b from Radish [HR03] with initial graph construction.The thick black lines are boundaries between free and occupied space. The small black circles are vertices placed in their corresponding region which are separated by thin lines. 191

5.8 The map created by the P3AT at UC Merced with initial graph construction on the left. The thick black lines are boundaries between free and occupied space. The small black points are vertices placed in their corresponding region which are separated by thin lines. On the right is the final graph resulting from contractions. . 195

5.9 The sdr_site_b from Radish [HR03] with initial graph construction. The thick black lines are boundaries between free and occupied space. The small black points are vertices placed in their corresponding region which are separated by thin lines. On the right is final graph resulting from contractions. 196

5.10 Robots following a sweep line with δ overlap and splitting into two sweep lines at a critical point. Robots with solid disks are moving towards future positions marked as robots with dashed disks. Note that the four robots require additional 4 robots to reach the dashed positions. 199

5.11 UCM map with borders thickened for illustration purposes. The graph is embedded with thin lines as edges inside free space. Vertices are small circles. The map is a polygon map obtained from the original grid map from fig. 5.8 after applying the α-shape and line simplification with $\alpha = 10$ and $\varepsilon = 3$. Distances are measured in pixel. To illustrate scale six horizontal lines of length 5,10,20,40,60 and 100 pixel are added. 200

5.12 SDR Map with borders thickened for illustration purposes. The graph is embedded with thin lines as edges inside free space. Vertices are small circles. The map is create from the original grid map from fig. 5.7 exactly as fig. 5.11. 201

6.1 Map of part of the Science and Engineering building at UC Merced built with a SICK laser on a Pioneer P3AT and the gmapping software [GSB]. The grey thin lines show the discrete approximation to the Voronoi Diagram, the thicker black lines the boundary of the environment, and the thick dashed grey lines show the boundary of regions associated to different vertices. Vertices are circles with edges as black dashed lines. 210

6.2 Guaranteed blocking positions for blocking using a camera with $\pi/2$ opening angle. The small triangle is the sensor, its coverage is grey and the obstacles associated to the minimum clearance value on the Voronoi edge (dashed line) are black squares. The other obstacles are squares with dashed boundaries. The circle shows the guaranteed obstacle free area. 213

6.3 Example of an improved sweeping implementation. On the left is the graph representation with cleared parts in grey and blocked edge with a stroke. The center shows how a robot sweeps, ensuring that no intruder can enter. Sensor coverage is shown in grey. On the right we have the status of the graph after the sweep. 213

6.4 A generalization of the vertex sweep implementation to vertices of degree larger than 3. The tree on the left corresponds to the environment on the right with the Voronoi Diagram as dashed lines. Each arrow indicates the movement of one robot. The leaves s_i and b_i are starting and ending points respectively. 214

6.5 The two Pioneer P3AT equipped with a SICK PLS200 laser. . . . 215

6.6 The paths computed for all vertices in the environment from fig. 6.1. Paths from one robots are shown with a black and the other with a white arrow. 217

6.7 The paths computed for all vertices in the environment from fig. 6.1. Paths from one robots are shown with a black and the other with a white arrow. 217

6.8 A diagram showing the high level states of the algorithm. 221

6.9 The maximum reach given 12 robots. 222

6.10 An example of how robots moving on a line cannot find the optimal way of moving it forward and the left side stops at position a). Yet, as the right side proceeds the left and right tangents approach the same values at position b). At this point the left side will move again and the line will shrink again. The bottom of the figure shows the local view from the robots of the tangents. 224

6.11 An illustration of the main line moving forwards to extend the cleared area marked in grey. On the right hand side are possible configurations for the tangent of the line. A tangent at an endpoint away from line leads to its length increasing while a tangent inwards leads to a decrease. The three cases depicted are one where 1) both sides lead to a decrease, 2) one side leads to a decrease and one to an increase 3) both sides lead to an increase in length. On the left figure some robots in the gray area serving as reserve are shown. 226

6.12 In this example a sweep line hits an obstacle and subsequently moves the two line-leaders into the direction for which the line lengths decrease. This reduces the number of robots needed by two. Then the sweep line splits, needing one additional robot for a total of seven robots. Robots that are in the reserve are not shown. 229

6.13 Top left: a line runs out of robots and cannot move further (the arrow indicates how it will move back). Top right: it then moves back. Bottom left: it then searches for a new obstacle. Bottom right: the line is split in two and each of them can individually move. 233

6.14 The radii of the circles are multiples of r_{follow}. The line from which the search originated is marked as a thick black line with thick circles representing the left and right line leader. The original line requires 7 robots and the small black dots show which points can be reached with a total of $r_{all} = 8$ robots by separating them onto the two new line segments into, i.e. for $i = 1$ there are $i + 1$ robots on the left segment and $8 - i$ robots on the right segment. The grey circles indicate the points that could be reached if we had $r_{all} = 9$. 234

6.15 In this example a line moving forward runs out of robots and moves backward. It then initiates a search but cannot reach the third obstacle. Executing a search from a different line position can, however, reach it. 235

6.16 This figure shows the three vertices added to a graph when a split occurs. Three vertices and three edges with their weights are added and connect to the existing graph. 235

6.17 Example of the construction of a graph as a result of the exploration with lines. The cleared and known vertices are marked in grey. The lines leading to their creation are marked as thick dashed lines with the associated number of robots needed. The robots explore the environment, following the arrows and stop before discovering the vertex with weight 7. Hence the neighboring vertex 4 is not considered explored. 238

6.18 A simply-connected environment with the starting point for the clearing and homebase at a). 242

6.19 The surveillance graph corresponding to a sweep schedule created with the methods from Section 5.2. The surveillance graph has 141 vertices resulting in 7 robots for clearing it. 243

6.20 With 8 robots the distributed algorithm fails to clear this environment starting at the bottom. This is due to the failed obstacle discovery with 5 robots at the top right while 3 robots block the edge to the left. In part a) the team runs out of robots and move back to part b). Once the line shrinks it attempts an obstacle search in part c) but the reach is too limited and it fails. The team then backtracks and tries the left side while blocking the right with 3 robots and fails in an identical manner. 246

6.21 The first split of the line moving with 9 robots that subsequently clears the environment. 247

6.22 A successful search after the second search step shown in part b). After the search the line splits and the right side clears the leaf in part c) and returns in part d) to join the left side. 248

6.23 For the left side the robots have to backtrack extensively. In part a) they fail with three robots and backtrack to collect 3 more. They fail again with 6 robots and have to backtrack again to collect the bottom three. 249

6.24 The last steps of the clearing. In part a) the robots extend far to follow the boundary and finish with steps b) and c). The graph resulting from this is shown in d). 250

6.25 Reducing the number of robots to 8 forces an obstacle search which leads to a different graph than fig. 6.24 as shown in d). 251

6.26 The final graph created by 9 robots embedded in the environment. 252

List of Tables

3.1 Results of the experiments. Values are averaged across 1000 random trees. 78

3.2 A simple example of a set of vertices and their assignment into batches. 95

3.3 Another example of vertices. 103

3.4 Reduction of the number of robots needed when using dynamic cycle blocking expressed in terms of the percentage of the number of agents needed to block all cycles at once. 127

5.1 Summary of the experimental results. 194

5.2 Summary of the experimental results from [KC08a]. 202

5.3 Summary of the experimental results with $\alpha = 10$, $\varepsilon = 7$. Note that some MST-edges are degenerate and have 0 weight. 204

5.4 Summary of the experimental results with $\alpha = 10$,$\delta = 2$, $\varepsilon = 3$. Note that degenerate MST-edges from table 5.3 do not appear here. 205

Acknowledgments

First, I would like to thank my advisor Stefano Carpin for his ongoing support and advice throughout the last three years, particularly for letting me follow up on my ideas and helping to develop them. I am also thankful for the support of my dissertation committee, Alberto Cerpa, Songhwai Oh and Steven LaValle whose encouragement and suggestions were always very much appreciated.

A great many thanks also go to my fiancée, Malgorzata Skorek, for making this work possible and my time at UC Merced all the more enjoyable. I could not imagine having done all this without her loving support. Finally, I am also thankful for the delightful company of my lab-mates, Ben Balaguer and Gorkem Erinc, and friends at UC Merced, especially Ankur Kamthe and Gayatri Premshekharan.

Vita

1982	Born, Bremen, Germany.
2001–2004	B.S., Mathematics, Jacobs University, Germany.
Summer 2003	Research Intern, Indiana University Bloomington, USA.
2003-2004	Student Lab Assistant, Social Cognition Lab, Jacobs University, Germany.
2004–2006	M.S., Computer Science, Jacobs University, Germany.
2007–2008	Teaching Assistant, University of California, Merced.
2007–2009	Graduate Student Researcher, University of California, Merced.
2008–2009	Teaching Fellow, University of California, Merced.

Publications

A. Kolling, S. Carpin. Pursuit-Evasion on Trees by Robot Teams. *IEEE Transactions on Robotics*, in press, 2009.

A. Kolling, S. Carpin. Cooperative observation of multiple moving targets: an algorithm and its formalization. *International Journal of Robotics Research*, 26(9):935-953, 2007.

A. Kolling, S. Carpin. Surveillance strategies for target detection with sweep lines. In *Proceedings of the 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, accepted for publication.

A. Kolling, S. Carpin. Probabilistic Graph-Clear. In *Proceedings of the 2009 IEEE International Conference on Robotics and Automation*, 3508-3514.

A. Kolling, S. Carpin. Stochastic Analysis of Controller Area Network Message Latencies with Observable Operator Models. *Society of Automotive Engineers World Congress*, Detroit, 2009.

A. Kolling, S. Carpin. Extracting surveillance graphs from robot maps. In *Proceedings of the 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2323-2328.

A. Kolling, S. Carpin. Multi-robot surveillance: an improved algorithm for the GRAPH-CLEAR problem. In *Proceedings of the 2008 IEEE International Conference on Robotics and Automation*, pp. 2360-2365.

A. Kolling, S. Carpin. The GRAPH-CLEAR problem: definition, theoretical properties and its connections to multirobot aided surveillance. In *Proceedings of the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*,

pp. 1003-1008.

A. Kolling, S. Carpin. Detecting intruders in complex environments with limited range mobile sensors. *Robot Motion and Control 2007. K. Kozlowski (Ed.), Lecture Notes in Control and Information Sciences (LNCIS) Vol. 360*, Springer, 2007, pp. 417-246.

A. Kolling, S. Carpin. Multirobot Cooperation for Surveillance of Multiple Moving Targets - A new behavioral approach. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation*, pp. 1311-1316.

H. Jaeger, M. Zhao, A. Kolling. Efficient estimation of OOMs. *Advances in Neural Information Processing Systems 18 (Y. Weiss, B. Schlkopf, J. Platt, eds.)*, MIT Press, Cambridge, MA., pp. 555-562, 2005.

H. Jaeger, M. Zhao, K. Kretzschmar, T. Oberstein, D. Popovici, A. Kolling. Learning observable operator models via the ES algorithm. In *S. Haykin, J. Principe, T. Sejnowski, J. McWhirter (eds.), New Directions in Statistical Signal Processing: from Systems to Brain*, MIT Press, 2005.

CHAPTER 1

Introduction

Every day a large number of rescue, reconnaissance, security and military personell expose themselves to tremendous risks in search of some essential piece of information, such as the location of a victim or hostile forces. Even when confronted with limited success rates countless volunteers engage in dangerous missions for the hope that they may save lives, up to a point where more lives are lost in the effort than are saved by it. Uncertainties and limited information for many of these missions only worsen the problem and make any risk analysis or predictions futile. Yet, the necessity to carry them out despite all risks is obvious. What remains is to empower those involved with tools that reduce risks, improve success rates and enable more missions to be carried out. Robotic tools promise to offer these advantages and their usage in search missions has just recently picked up pace. It is the continuation and acceleration of this trend that we aim to support with the work in this dissertation.

One of the largest disasters in recent decades, the collapse of the towers of the World Trade Centers, was also one of the first to see the use of new robotic rescue technologies in practice. Their mission on this day was rather limited and they only served to identify ten sets of remains [Mur04]. Yet, their deployment reinforced the belief in the potential that robotic systems have in freeing personell resources, reduce risks and improve the overall success rate of difficult search missions. Today, about eight years later, the number of robotic systems assisting

human personell in search and surveillance tasks has risen dramatically. Initial research efforts primarily and rightfully focused on the challenges of human robot interaction and robustness of individual robotic components. Autonomy is often only a marginal part of these systems and tight interaction between a human operator and a single robot is the norm. Hence, they do reduce risk and improve the capabilities of the team, but do not significantly reduce the need for human resources. Furthermore, they require reliable communication between the robot and a base station which increases the cost of the robotic system as a whole. Also the actual robots, apart from the base station, tend to be more expensive since every single robot needs to deliver significant capabilities to warrant the added expense of a human operator. Despite these shortcomings, rescue or military personell given the chance to experiment with robots in the field frequently resist to return them, which is a strong testament to their pivotal role.

Many more robot components, sensors and actuators in addition to computation and communication devices, now participate in a Moore's Law like reduction in cost and space requirements. This reduction is fueled by recently appearing mass markets for smart-phones, intelligent vehicles, sensor networks and robots as consumer products. These developments enable the design of robotic systems comprised of a large number of robots without exploding the cost. Such *multi-robot* systems offer different advantages than multiple single robot systems. They can satisfy the criteria of **robustness through redundancy** instead of requiring reliability of every component through a costly design. This same property of cost-reduction also allows the requirements on autonomy to be less stringent. With an expensive single robot its autonomy has to ensure that the robot remains functioning. With replaceable robots we can afford to loose one of them on occasion which allows us to require less robust autonomy. But foremost multi-robot systems deliver **scalability through coordination** that no single robot system

can provide. A single robot is inherently limited in its spatiality. Multi-robot systems scale to larger and more complex tasks merely through the addition of more robots. Yet, to enable their widespread usage and take full advantage of their benefits one has to enable the efficient coordination of robots with limited capabilities. As a consequence, the study of multi-robot systems is often focused on the coordination of a larger number of robots, each with limited capabilities, and the scalability of the entire system for the completion of complex tasks. The design and study of such multi-robot systems for pursuit-evasion, a particular search task, is the subject of this dissertation. This particular search task describes the detection of unknown, mobile and possibly hostile targets.

1.1 Multi-Robot Pursuit-Evasion

The term **pursuit-evasion** generally refers to a worst-case search scenario in which the search targets are mobile adversaries with unbounded speed and complete knowledge, but bound to move on continuous trajectories. In the context of graphs this entails that a target can move to any vertex of the graph as long as its path does not include vertices or edges on which it will be detected. For two dimensional environments targets are simply bound to move on continuous curves but also at unbounded speed. Given this worst-case adversary assumption the task of detecting all targets located in an environment becomes equivalent to the clearing of all contamination from an initially fully contaminated environment. A contaminated part of the environment then simply refers to the possibility of a yet undetected target being located therein. For a cleared part we can then ensure that no undetected target is located in it. Robots can clear parts of the environment and can restrict the spreading of contamination by blocking potential paths for the target. If not blocked, then contamination spreads instantaneously

as a consequence of the unbounded speed. A variety of pursuit-evasion problems have been studied on graphs and in two dimensional environments. We discuss a few of these in Chapter 2. The worst-case target assumption is useful when no information about targets is available and when it is critical that no target is missed. In this sense the approach is conservative and more benign target behavior can only lead to improvements. But it additionally simplifies the theoretical treatise as we shall see in the chapters that are to follow.

The term **multi-robot** in this thesis refers to robotic systems in which individual robots are only equipped with limited capabilities. In practice this entails very limited sensing and communication ranges which render each robot useless for real pursuit-evasion tasks, unless it is part of a large team. Chapter 2 reviews some related work in this domain.

1.2 Goal and Overview of Contributions

The goal of this dissertation is:

> *To develop a theoretical foundation and algorithms that enable the design of scalable and fully autonomous multi-robot systems to solve pursuit-evasion tasks.*

One of the main contributions to this end is a rigorous formalization in form of a graph model that captures combinatorial aspects which arise when coordinating large robot teams in large environments. This model is coined Graph-Clear and presented in Chapter 3. In Graph-Clear the environment is represented by a graph with vertices associated to regions and edges connecting vertices that are associated to adjacent regions. The capabilities of the robot team are abstracted to actions, so called sweeps and blocks, on vertices and edges that, if executed

in proper order, can clear a graph. Each action requires a certain number of robots given by a weight on the vertex or edge. The Graph-Clear problem is to identify sequences of actions, so called strategies, that clear an initially fully contaminated graph at lowest cost. It is closely related to previous pursuit-evasion problems introduced on graphs which we will review in Section 2.1 and as such also contributes to the graph and game theory literature. While the graph abstraction is useful to tackle combinatorial and scalability issues in a rather general manner, there is still a gap between the coordination of real robots and strategies on graphs. To utilize the strategies of a graph one needs to implement routines or behaviors for the robots that enable them to execute a sweep or block action. In principle, these implementations can differ dramatically for different robotic platforms and sensors and there are only few requirements that they have to fulfill. Chapter 6 discusses examples of such implementations.

To complement Graph-Clear and gain insights into limited range pursuit-evasion in two dimensional environments we introduce a second problem coined Line-Clear in Chapter 4. Therein the capabilities of the robot team are abstracted to the ability to cover lines that move through the environment. A solution to Line-Clear entails a schedule for the creation and movement of these lines and the goal is to compute a schedule that can be executed at lowest cost. Coordinating these moving lines, covered by chains of robots, involves solutions to the Graph-Clear problem and we thereby establish a direct relationship between multi-robot pursuit-evasion in two dimensional environments and graphs. To utilize a Line-Clear schedule for real robots one only has to implement a line-following behavior for the robot team. Implementations for this are also discussed in Chapter 6.

Another practical aspect in the utilization of Graph-Clear strategies is the extraction of a graph from a given robot map. In theory, graphs could be cre-

ated manually, but it is desirable to have an automated extraction method which is discussed in Chapter 5. The presented methods are subsequently applied in Chapter 6 to create graphs from maps collected by robots from real environments and then coordinate their movement, closing the loop from the theory and algorithms to the design of multi-robot systems operating in real or simulated environments.

Figure 1.1 shows an overview of the above mentioned contributions.

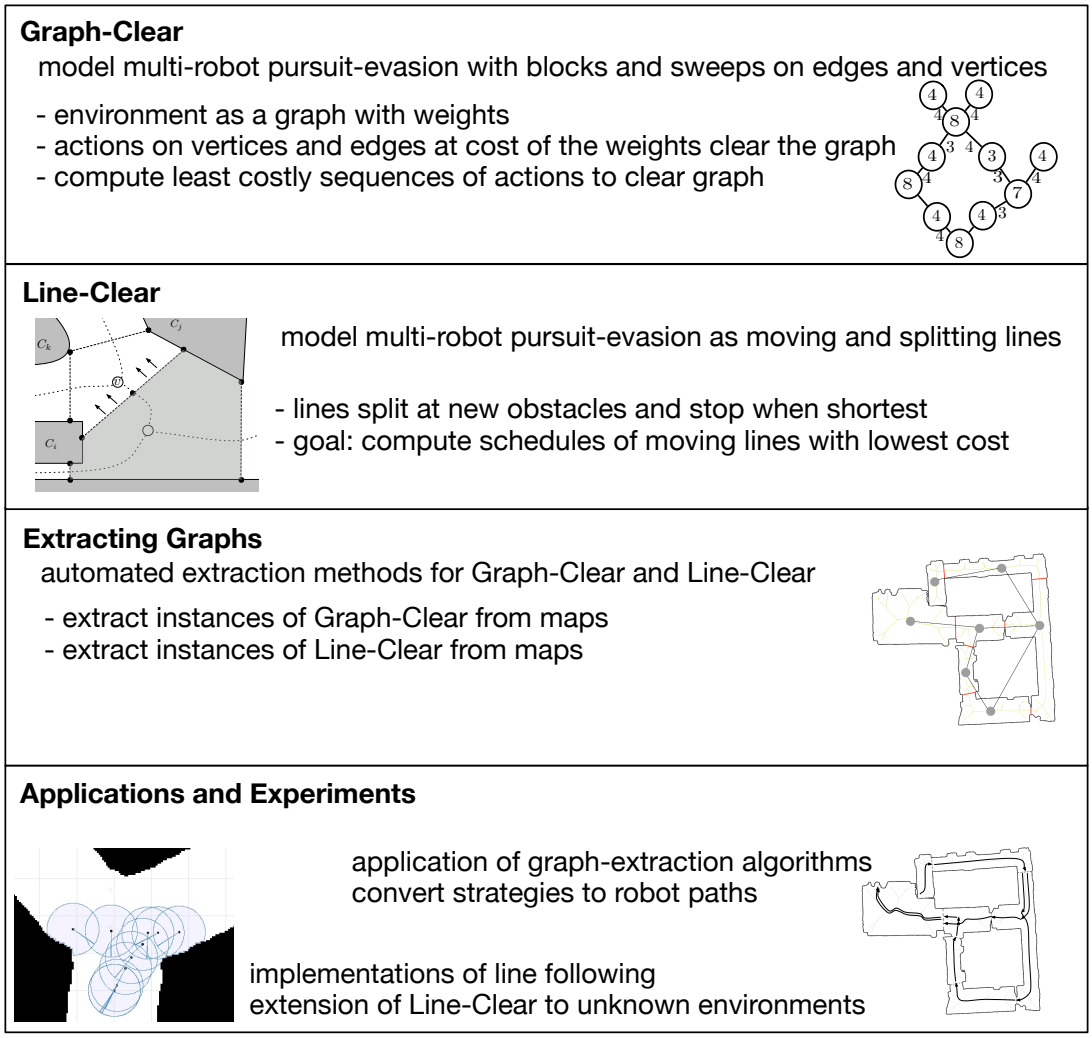


Figure 1.1: An overview of the contributions of this dissertation.

CHAPTER 2

Related Work

Search, surveillance, reconnaissance and target detection problems have a very long history in computer science. They have been studied in manifold variations and these often involve challenging combinatorial or geometrical problems. For these problems computer scientists and mathematicians have contributed a wide range of results and approaches relating to topics such as game theory, optimization, graph theory, computational geometry and many others. Recently, a growing interest of the robotics and sensor network research communities on these topics has reinvigorated their study and added new perspectives. The purpose of this chapter is to give an overview of this large and varied area of research. In this endeavor we shall put an emphasis on topics most closely related to our contributions. Yet we shall also try to give an overview of the following areas of interest:

1. pursuit-evasion and searching on graphs,
2. visibility-based pursuit-evasion,
3. probabilistic graph searching,
4. direct pursuit,
5. coverage and clearing,

6. pursuit-evasion and control theory,
7. cooperative multi-robot approaches,
8. probabilistic approaches,
9. sensor networks and tracking,
10. and real robot systems.

There are many topics that we will not be able to address. These include the actual process of target detection [SOD02] which can be based on a variety of sensors such as cameras, lasers, sonars, infrared or microphones, each with their specific challenges.

One of the first computational problems related to search and surveillance has been the art gallery problem. Excellent surveys in this field are written by O'Rourke [OR87] in 1987 and Shermer [She92] in 1992. In short, the art gallery problem is concerned with finding the minimum number of omnidirectonal cameras needed to surveil an art gallery. It is hence dealing with the deployment of static sensors with unlimited range. But it was not long after the original problem was proposed that variants with mobile guards were proposed. One of the first results with mobile guards patrolling on edges or diagonals in a polygon were due to O'Rourke who showed that the minimum number of mobile guards for any polygon of n vertices is $\lfloor n/4 \rfloor$ [OR87]. Continuing along this line a large body of work in robotics has been developed that is now known as visibility-based pursuit-evasion. We review this area in detail in Section 2.2. It is concerned with the detection of omniscient and fast intruders with robots with unlimited range sensors in a variety of two dimensional environments. Interestingly it also has a connection to pursuit-evasion problems on graphs, most notably edge-searching,

which we review in Section 2.1. Our own contributions presented in this dissertations can be seen as a continuation of visibility-based pursuit-evasion towards sensors with limited range.

Regarding the terminology it is useful to note that in the related literature across the subjects we are about to address there is obviously no coherent terminology regarding searchers and intruders. Searchers are sometimes also referred to as pursuers or robots while invaders are also referred to as evaders or targets. At times the use of a particular term may have a different meaning, but such usage is too inconsistent to assign such meaning here. Generally, the properties of searchers and intruders are clear given the context and we use the terms interchangeably.

2.1 Pursuit-Evasion on Graphs

In this section we review the topic known as graph-searching. One of the first graph-searching problems, now often called edge-searching, was proposed by Parson in [Par76] and [Par78]. In edge-searching searchers traverse edges in a graph to capture an invader that can move arbitrarily fast. The concept of contamination is used to represent the possibility of an intruder occupying an edge and a searcher moving along the edge clears this contamination. A search strategy consists of such moves along edges and placement and removal of searchers on vertices. While a searcher is on a vertex it is considered guarded and no contamination can spread through it. Conversely, contamination spreads through all vertices that do not have a searcher placed on them. Eventually a search strategy has to clear all contamination which implies that all intruders initially contained on the graph must have been detected, regardless of their speed. The problem in edge-searching is to find the smallest number of searchers, known as

the *search number* $s(G)$ on a graph G, with which which one can clear the graph. In [MHG88] Megiddo et al. showed that the decision variant of finding $s(G)$ is NP-complete. In their proof they reduced edge-searching to the well-known *min-cut into equal sized subsets* problem [GJ79]. Meggido et al. also presented a linear time algorithm for finding the search number in trees and an $O(n\ log\ n)$ algorithm to compute a search strategy for capturing the evader. The proof relies on an early manuscript from 1982, later published as [LaP93], in which LaPaugh showed that for a pebbling version of edge-search recontamination can be avoided without changing the search number. Another proof that recontamination is not necessary in edge-searching is given by Bienstock and Seymour in [BS91]. The two papers [MHG88] and [BS91] are particularly relevant for our work since they relate to the NP-hardness proof in Section 3.3 and recontamination result in Section 3.4. Also directly relevant is [BFF02] in which a weighted variant of edge-search is first introduced. It also includes the requirement that strategies are contiguous, i.e. the cleared parts of the graph form a connected sub-graph. Consequently, we will discuss these three papers in more detail in Chapter 3.

Apart from edge-searching there are a number of other graph-searching problems. One variant, called node-search, has been defined by Kirousis and Papadimitriou [KP86]. Therein the invader is caught on an edge when searchers are located on both adjacent vertices, i.e. searchers can *see* into edges. They also show that the node-search number[1] is identical to the vertex separation plus one[2]. In the domination search game from [FKM03] the searchers can see even further and detect intruders in adjacent vertices. It is interesting to note that in the domination search game recontamination can actually help. Other relations between parameters of graph layouts and variants of graph-searching have been

[1] analogue to the search number for edge-searching

[2] The vertex separation simply denotes the minimum number of vertices that if removed separate the graph into disconnected sets of vertices.

studied extensively. A first relation to layout problems has been established by Makedon and Sudborough in [MS83], [MS89] by showing that $s(G)$ is equal to the cutwidth of G if the maximum degree of any vertex is 3. Later, Ellis, Sudborough and Turner in [EST94] also established a relation between $s(G)$ and the vertex separation. Another relation of edge-searching to a graph layout problem is found in [MPS85]. Therein the topological bandwidth is related to the search number and node-search number. Such relationships between graph-searching and graph layout problems are particularly interesting since they can lead to the utilization of efficient graph layout algorithms to efficiently compute search numbers or strategies. An optimal layout for the vertex separation problem restricted to tree can be computed in $O(n \cdot \log n)$ while the vertex separation number can be determined in $O(n)$. Further results for the layout problem for the vertex separation from Skodinis in [Sko00] show how to compute the optimal layout in $O(n)$. A survey on graph layout problems is presented by Diaz in [DPS02]. There are even further variants of graph-searching, such as the mixed-search and inert-search and most of these have also been related to layout problems and surveys. A recent annotated bibliography of graph-searching is available in [FT08]. For our graph-searching problem, coined Graph-Clear, introduced in Chapter 3 it is not known whether interesting relationships to graph layout problems exist nor whether they can lead to improved algorithms. From a theoretical perspective such questions may be of interest, yet they are beyond the scope of this dissertation.

From a robotic perspective the work done in graph-searching is useful to coordinate robots in order to solve pursuit-evasion tasks in environments for which a robot team has a graph representation. The original sense of intruder also included intruders such as in computer systems for network security [FGY00]. In robotics, however, one usually refers only to physical intruders which are often

called targets. Consequently, there have also been developments in robotics to augment graph-searching and introduce new variants that model robotic problems more closely or new algorithms that are more suitable when running on a robot team. One such new variant is our Graph-Clear problem from Chapter 3 which was first introduced in [KC07c] and later refined in [KC09b]. It originated from earlier work in [KC07b]. One extension to graph-searching that is very particular to the robotic context is the consideration of a probabilistic failure model for sensors presented in [KC09a]. But also other graph-searching problems, such as edge-search and node-search, are being used to coordinate robots. Most notably, in [KHG09] an edge-searching variant is discussed for which contamination is moved to vertices instead of edges, i.e. intruders are located in vertices. This problem shares many properties with edge-search and the application of edge-search algorithms to this problem is straightforward. In fact, the problem is equivalent to mixed-search which is a combination of node-search and edge-search, i.e. intruders are detected by moves along edges and whenever both endpoints of an edge have a searcher located on them. In [HKS08] Hollinger et al. address the issue of converting search strategies from trees to a graph. For this spanning trees are generated at random and strategies are converted similar as presented in [KC07c] in which only one spanning trees, namely the inverse minimum spanning trees, is considered. We shall see some details on this procedure in Section 3.10. It is interesting to note that the procedure to try multiple spanning trees can also directly be applied to Graph-Clear. Adding more robotic-centric aspects in [HSK09] pursuit-evasion search strategies are combined with improvements to increase the likelihood of early detection given a target motion model. Also the methods therein can be applied to our Graph-Clear problem in an analogue fashion and greatly increase its utility. Another pursuit-evasion problem is discussed in [KHS09] in which robots can see into a set of vertices depending on

the geometry of the environment from which the graph was constructed. Graphs are created via partitioning a two dimensional environment into convex cells and visibility of these cells is given whenever all points from one cell are visible from all points of the other cell via a straight line of sight. It is hence assumed that all robots have an unlimited range sensor.

To conclude this section, there is a great amount of work done in graph searching, some of it with a theoretical focus but also some with the explicit motivation to apply it to robotic scenarios. But all graph based approaches require some connection between the graph and the environment in which robots are placed. This requires the solution of a subproblem, namely the generation of suitable graphs for the pursuit-evasion problem for given environments. This problem has not been addressed sufficiently in the robotics literature and our approach published in [KC08a] and presented in Chapter 5 is so far one of the few focusing on this problem with the exception of the recent work in [KHS09]. An alternative approach to robotic pursuit-evasion that is directly concerned with two dimensional environments, namely visibility-based pursuit-evasion. Also here there is a large amount of work done which we shall present in detail in the next section.

2.2 Visibility-Based Pursuit-Evasion

One of the earlier visibility-based pursuit-evasion problems was introduced by Sugihara et al. [SSY90]. Therein they considered rays whose direction can be changed but the position of the emitter remains stationary. An intruder is detected if the ray hits it. This problem has recently been investigated by Obermeyer et al. [OGB07, OGB08]. Obviously the mobility restrictions are rather severe and lifting these to let the source of the ray move freely brings is to what

is commonly referred to as visibility-based pursuit-evasion. It was first introduced for polygonal environments by Suzuki and Yamashita in [SY92] and it is a natural extension of graph-searching towards a more robot centric setting with an emphasis on mobility of the searchers. This became the concept of a k-searcher, which is mobile searcher that has k flashlights that emit beams to detect intruders. An ∞-searcher, on the other hand, is a point source for which such beams go into all directions. Sufficient and necessary conditions for existence of a search schedule with a k-searcher in simple polygons are presented, but most problems remained open and spurred much further work. In [CSY95] Crass et al. considered multiple intruders that can enter through an edge and are not allowed to reach another edge of the polygon. Sufficient conditions for the polygon to be searchable are presented for a ∞-searcher first, but then also shown to be valid for the 2-searcher. In [YUS97] Yamashita et al. introduce upper and lower bounds on the so called search number, analogous to the search number for graph-searching, of a polygon. They show tightness of these bounds with worst-case constructions of certain polygons. Some of the upper bounds are in fact derived from graph-searching. A first complete algorithm to solve the visibility-based pursuit evasion problem was given by LaValle et al. in [LLG97] for the ∞-searcher. The approach is motivated by information states that change when critical boundaries are crossed by a searcher. The information states are associated to gap-edges of the sensors, i.e. those edges of the polygon covered with the sensor that are adjacent to free space. These critical boundaries partition the polygon into cells. For simple polygons that can be searched with one searcher this produces at most $O(n^3)$ cells. Combining this decomposition of the polygon in graph form with the information state produces a new graph which is the one a solution is sought in. The size of this graph, however, is exponential and hence does not scale very well to larger or very complex environments. A

few pointers on how to extend the algorithm to multiple searchers, losing completeness, are also presented. In a similar context, in [GLL99] Guibas et al. established NP-hardness of finding the minimal number of ∞-searchers needed for any polygon and presents the algorithm also shown in [LLG97]. They also show that recontamination can sometimes help to find better solutions, contrary to the result for edge-searching. Finally, in [PLC01] a quadratic algorithm for solving the visibility-based pursuit-evasion problem in polygons was presented. They give three necessary and sufficient conditions for searchability. Furthermore, they prove the conjecture by Suzuki and Yamashita that a polygon searchable by an ∞-searcher is also searchable by a 2-searcher. Curved environments were first considered in [LH01]. The approach therein extends the critical boundaries from [LLG97] to smooth boundaries of the environment (based on inflections and bitangents on the boundary of the environment). The environment is simply-connected and the pursuer has omni-directional vision. The development of an online-version is outlined as a valuable direction for further work. Back to polygonal environments LaValle et al. in [LSS02] presented an algorithm for a pursuer with only a flashlight (1-searcher). The algorithm solves the problem by Suzuki and Yamashita for 1-searchability and produces a search strategy if one exists. Simple polygons with n edges and m concave regions are considered and the algorithm has a complexity of $O(m^2 + m \cdot \log(n) + n)$. The basis of the algorithm is a so called *visibility obstruction diagram* which is a 3-partition of the configuration space. The configuration space is the Cartesian product of points on the boundary. The 3-partition distinguishes 1) diagonal configurations: points on the same edge 2) feasible configurations: mutually visible points and 3) all others. In this diagram certain paths, called winning paths, lead to a strategy of the pursuer in the polygon. The search space of the diagram is reduced to a skeleton by considering critical points on the boundary of the polygon. In this structure a path can

be found efficiently. A previous version of this algorithm is presented in [SSL00]. While the previous solves most of the problems proposed by [SY92] many other questions remained open. Hence research naturally continued. In [SLS02] an algorithm for two 1-searchers in a polygon is presented.

In [GTL04] some of the results from [TLM03] were used to design an online algorithm for pursuit-evasion in an unknown simply connected environments. The sensor is a simple gap sensor. The motion strategy is based on critical events regarding gaps, i.e. appearances, disappearances, splits and merges of gaps. The approach uses sentries that the pursuer can place (and collect again) in the environment The number of dropped sentries is bound by $O(log\ m)$ where m is the number of bitangents (relating to critical events). The used gap-navigation-graph is a tree, since the environment is simply-connected. One interesting theorem is that if one robot can clear the environment, then so can the robot using the gap-navigation-tree with at most two sentries. A lot of the methods presented therein are similar to what appears in [SRL04]. More on gap navigation trees is presented in [TGL04].

A culmination of much of the work in visibility-based pursuit-evasion can be found in [SRL04]. Therein Sachs et al. present an on-line algorithm for a point pursuer moving in an unknown, simply-connected, piecewise-smooth planar environment. The pursuer is only equipped with a sensor that measures depth-discontinuities. Also the controls are minimalist as only wall-following or a movement along the measured depth-discontinuities is allowed. Furthermore, imperfect control is assumed. The approach incrementally builds a navigation graph based on the motion primitives. The informations state about possible locations of the invader is superimposed on this graph forming the so called information graph. It achieves an online version by 'envisioning' preliminary so-

lutions in the information graph. It is a complete algorithm that enables the limited pursuer to clear the same environments that a pursuer with a complete map, perfect localization, and perfect control can clear.

Further results regarding multiple robots, i.e. precisely two 1-searchers is done by Simov et al. in [SSL09]. The environment is restricted to a simple polygon. This paper extends [SLS02]. They present an $O(n^2 + nm^2 + m^4)$ algorithm to compute a search strategy in a polygon with n edges and m concave regions, if one exists. The algorithm is complete. It is also based on an information state graph using an elaborate geometrical characterization of the polygon. Further work mentions an extension to an online approach similar to [SRL04], an extension to 360 degree vision or an algorithm for any number of pursuers.

Gerkey et al. in [GTG06] extend visiblity-based pursuit evasion to a limited field of view. It introduces a more applicable flavor by consider the ψ-searcher instead of the k-searcher (which are similar concepts), considering a limited field of view sensor rather than multiple beams. They show that the problem of finding the minimal number of ψ-searcher is also NP-complete (as for pursuit-evasion graphs and visibility-based for the ∞-searcher) for a given polygonal environment. They focus thereon on a complete algorithm for a single ψ-searcher.

One very interesting approach is developed by Tovar et al. [TL06] , who consider bounded speeds for evader and pursuer. The settings is again a simply-connected polygonal environment. The evader is assumed to attempt to escape detection. The further information about speed adds significant 'power' to the algorithm, enabling it to compute solutions in cases where previous approaches failed. It involves the computation of a reachability set (generally an intractable problem). Modifying the evader and pursuer speed ratio relates the problem to the infinite evader speed for visibility-based problems or the 0-speed for cover-

age problems (see Section 2.6). One key thing is the fact that with bounded speeds recontamination can be modeled, i.e. previously visible regions are not instantaneously recontaminated, but depending on the distance to the contaminated regions, only recontaminated after a certain time has passed. There are still quite some open questions in this direction when considering further assumptions on the evaders motion. A difficulty of the approach is to describe how the recontamination regions, so called fans, evolve.

Despite its well-developed theory, the visibility-based approach still has severe limitations for practical applications. Unlimited range sensors and frequently revisited areas and only limited applicability for multi-robot teams are its major weaknesses. Our own contributions aim to introduce models and algorithms that can fill this gap and be useful when sensing ranges are restricted.

2.3 Probabilistic Graph Searching

Another extension of graph-searching is the consideration of random motion on the graph. One randomized approach for detection of an intruder on a graph is presented by Adler et al. in [ARS02]. Two variations of pursuit-evasion are considered. In the first both a searcher and an intruder can move at most one node per round while in the second the searcher can move unrestricted, i.e. jump from any node to another. Note that the worst-case target assumption of an omniscient target is hence not used in this context. The game ends when searcher and intruder are at the same node. The measure for a searcher strategy is the expected time of capturing the intruder, the escape length. The main result they show is that the escape length is $O(n \cdot \log(diam(G)))$ where n is the number of nodes in the graph. The paper is primarily interesting from a game-theoretic perspective by considering competing optimal strategies. In [IKK04b] Isler et al.

consider another randomized pursuit-evasion on a graph with two searchers. The information the intruder has about the searchers is considered as visibility (full, limited or none). Limited visibility means that the intruder can see into nodes adjacent to the current location. They show that two searcher always suffice and present a polynomial time algorithm that checks whether one searcher suffices. The also considers reactive intruders that only move when the searcher is close (evading it).

Randomized approaches are also presented for visibility-based pursuit-evasion such as in [IKK04a, IKK05] by Isler et al. Their strategy is based on a triangulation tree of the polygon. At each node its children are picked with certain probabilities (equally distributed across children) while the searcher moves down to a leaf of the tree. They show based on a worst-case construction that there is no better strategy. Then they extend the approach to two searchers and relate it to the Lion and Man problem [Guy91]. In [IBD04] Isler et al. consider the visibility-based pursuit-evasion setting in a polygon. It is an extension of a discrete randomized strategy to a continuous dynamic model of the searcher. The polygon is triangulated and similar strategies than in the graph case from [IKK04b] are used. Additionally, motions that bring the searcher from triangle to triangle are presented. Further research is indicated for the multiply connected case. Continuing along this lines of work in [ISS05] Isler et al. consider a pursuit-evasion game in which the searcher attempts to collide with the intruder. This is a useful problem for collision avoidance since one robot can assume the worst-case pursuit scenario for the motion of another robot and then act accordingly to avoid a collision. Note that this type of pursuit-evasion is in fact a direct pursuit problem with knowledge about the intruders state which we shall shortly discuss in the next section.

2.4 Direct Pursuit

Most previously discussed pursuit-evasion problems generally assumed an omniscient and fast intruder. This type of pursuit-evasion hence uses the concept of contamination that is to be cleared. Naturally, these problems are not concerned with what happens if a target is detected by sensors nor how to keep it from escaping again. This later problem, however, has also sometimes referred to as a pursuit-evasion problem. This seems intuitive since now there is an actual chase, an active pursuit and an active evasion. In this dissertation, however, we shall refer to this kind of pursuit-evasion problem, when searcher and intruder have knowledge about each others position, as direct pursuit. This involves a whole different set of considerations such as motion planning and target movement prediction. In one approach to this problem LaValle et al. in [LGB97] use the predicted target trajectory to compute an optimal trajectory for the pursuer to maintain visibility while being subject to a constraint function (e.g. total distance travelled or trying to maintain a certain distance). Also the case for partially-predictable targets is considered. Since the state-space increases to four dimensions when considering partially predicted targets a new approach is taken with respect to to the predictable case. A state-feedback strategy is now presented that chooses the strategy with the smallest worst-case loss. Alternatively, one can choose to minimize the expected loss. Another approach assumes a probabilistic uncertainty model to predict target's motion. Experiments with two real robots were conducted. While the minimization in the state-space is a critical part for the computation the experiments showed that the approach is feasible to implement. More results in this area are obtained by Murrieta-Cid et al. [MMH08, MMS07, MMA05, MTH05, MMH05]. An interesting extension of direct pursuit from our perspective is the consideration of many searchers and

many intruders. This requires a different set of solutions in order to have scalable algorithms and brings us to the coordination of large multi-robot systems which we shall discuss in the next section.

2.5 Cooperative Multi-Robot Approaches

Search and surveillance have naturally been of interest to multi-robot researchers since multi-robot systems have the advantage that they offer scalability to large environments and provide better coverage by being physically distributed. An excellent survey in cooperative mobile robotics is given by Cao et al. in [CFK97]. Since then, however, some progress related to robotic surveillance has been made. In [Par02] Parker defined the problem of Cooperative Multi-robot Observation of Multiple Moving Targets (CMOMMT) and presented a first solution called A-CMOMMT. In CMOMMT a large number of targets and robots are randomly distributed in a large and uncluttered environment. The goal is to maximize the joint observation time of all targets with robots that can only communicate and sense locally. The proposed approach, coined A-CMOMMT, is distributed and based on simple heuristics for the local control of individual robots. It is compared to baseline approaches in simulation for large numbers of targets and robots and on real robots for a moderate number of targets and robots. The problem has further been addressed in [KC07a]. Therein an improved behavior-based approach is presented and is compared experimentally to A-CMOMMT. It outperforms A-CMOMMT but also has proven performance bounds that are shown to be approached in practice as well. The main drawback of the CMOMMT problem is that it considers a simple scenario, particularly the restriction to uncluttered environments without complex obstacles. Another multi-robot centric approach with a lot of practical heuristics is presented in [JS02]. Therein Jung

and Sukhatme are investigating a region based approach for surveillance in complex environments which considers complex obstacles. Previous experimental results with Player/Stage [BCG05] with a similar approach are found in [JS01]. In [JS02] robots are cooperating in order to control the distribution of robots across regions. A robot is migrating to another region when it detects that the ratio of targets to robots is less than a defined threshold. In this case the robots move to the most urgent region, a measure depending on the distances to the regions and the robots and targets in the regions. Each robot has to maintain these variables from the broadcast packages it receives. For the local following of targets within a region each robot tries to maximize the number of targets observed by computing the center of gravity of all known targets and positioning itself a certain distance apart, depending on the field of view of the sensor and the maximum distance across targets to the center of gravity. Experiments with Player/Stage [BCG05] and Pioneer DX-2 robots were carried out. The results show that the redistribution of robots into needy sections gives a significant gain in performance on the total coverage in environments with corridors. In an empty environment a strategy with only local following gives better coverage while both strategies perform equally well in open environments with some occlusions. Furthermore, they investigated static sensors in cooperation with mobile sensors and showed that in an office-like setting with two sensors the performance increased if there was one of each kind instead of two of one kind. The approach has a twofold merit, for one the cooperation of static and mobile sensors is introduced and a first indication of the benefit of combining both is given via the experiments. The second is an introduction of a hierarchy of control, within a region robots follow a different behavior until they are called to another region, i.e. we have local control within one region and an emulated global control between regions while still maintaining the variables for the global control on each robot independently

(via the broadcast messages about target density). Further details on the density functions for targets/robots and utilities are given [JS06]. The idea of hierarchical control is similar to how we envision Graph-Clear can be useful, namely by providing a global coordination solution for local methods.

In [KHS05] Krishna et al. present an approach that they determine to be closely related to [Par02] and [JS02]. Their addition is assuming prior knowledge about the arrival statistics, described by a Poisson distribution, of targets crisscrossing a known rectangular region. A similar metric than in [Par02] is used, i.e. the total amount of time that targets are observed. The movements are determined by the expected number of targets surveilled for the next T time steps of a given sensor. The environment is partitioned into cells for the computation of the best trajectory. The main coordination between robots is to assign higher priorities to paths with higher number of detections. Overlaps are avoided by introducing a penalty proportional to the overlap and then recomputing the lower priority path. This approach is not particularly suited for overlapping sensors and have only few elements of cooperation and no real communication. Previously, in [KHP04] Krishna et al. present a priority driven system that decides when a mobile sensor should remain stationary or follow a target. It also incorporates some elements of cooperation between sensors by using a simple set of rules for the resource allocation. Again a cell modeling is used and no occlusions are present. Another paper by Krishna and Hexmoor [KH05] is on resource allocation for multi-sensor surveillance.

Overall, these multi-robot approaches for the continued observation of as many targets as possible show that with a few heuristics one can already assemble systems that do rather well for this task within a certain context. Theoretical guarantees for system performance, however, are rarely encountered and hence

different approaches have to be compared experimentally. The continued observation of targets becomes primarily of interest if a pursuit-evasion scenario is coupled with continuous monitoring. As the robots responsible for the pursuit-evasion problem extend cleared areas other robots may continue to observer those targets that are already once detected and now within cleared area.

2.6 Coverage and Clearing

Yet another related area to multi-robot pursuit-evasion is the coverage or clearing of large environments. This is primarily of interest if one searches for a static or slow moving target, i.e. one has to relax the assumption that the target is fast and omniscient. In this case it is more desirable to compute short paths that visit all parts of an environment. This is similar to attempting to cover all of an environment, such as in applications in which robots paint, deliver seeds or collect crops in a field. But also security application such as mine countermeasures are related to this problem. A detailed survey on the field is presented by Choset in [Cho01]. Cellular decompositions are often used in algorithms for coverage or clearing. Here approximate, semi-approximate and exact decompositions have been used. Some experiments suggest that randomized approaches may outperform exact approaches with respect to the time to completion. For our purposes, it is interesting to note that some algorithms for intruders with limited speed share some ideas and aspects with coverage algorithms.

2.7 Pursuit-Evasion and Control Theory

Pursuit-evasion problems have also received considerable attention from researcher with an emphasis on control theory. A number of scenarios and approaches have

been considered in this context, often with strong theoretical results. But also other problems that are considered in control theory can be useful for pursuit-evasion by providing solutions for optimal coverage and rendezvous problems, particularly when very limited capabilities of the robots are considered. We shall first discuss a few approaches to these two problems and then discuss control problems directly related to pursuit-evasion.

In [CMK04] Cortés et al. are concerned with decentralized control laws that allow a team of robots to coordinate their coverage of the environment in accordance with a utility function that is derived from optimal coverage and sensing policies. Their approach involves centroidal Voronoi diagrams and a local gradient descent. This approach is particularly useful if one has access to a density function that describes the level of importance of parts of the environment. The robots then distribute themselves more closely and dense around areas of high interest. An extension of this approach is found in [MB06]. Therein the determinant of the Fischer Information Matrix for range-measurement models is used as an objective function for the previous approach from [CMK04]. This is leads to decentralized control laws that produce the desired global behavior of the sensors arranging in an optimal configuration. The numerical simulations use an Extended Kalman Filter for the estimation of target locations and for sensor fusion.

Another related problem is that of rendezvous of multiple robots discussed in [CMB04]. For robots solving pursuit evasion tasks the ability to find an agreement over their location is an essential capability. In [CMB04] Cortés et al. present an algorithm for a network of mobile agents to achieve an agreement over the location in the network. The communication topology is represented via proximity graphs that capture the information on which agents can communicate to each other and

are closest to each other. Each agent utilizes this information to compute the circumcenter of all neighbors and itself and moves toward this point. While moving, agents try to maintain connectivity, i.e. to follow a restricted motion to avoid losing communication links. A similar problem is discussed in [JLM03] in which Jadbabaie et al. provide a distributed algorithm for orienting a group of agents into the same direction. There are several variations of the algorithm, one with and one without agents that act as leaders. In all forms the standard agents compute their new orientation according to the average orientation of all neighbors. This way a common orientation should be reached. This approach can be helpful for implementing line-following behaviors for Line-Clear which we introduce in Chapter 4.

Control theoretic approaches that are directly related to our topics have also become popular recently. Most notably in [GCB06] Ganguli et al. present an approach to deploy robots, which are called guards in this context, in a polygonal environment starting from particular location. For this the environment is partitioned into star-shaped polygons and robots are assumed to have an omnidirectional and unlimited range sensor. A control algorithm for moving the sensor with respect to this partitioning is given. The approach is more closely related to art gallery problem as the final positions of the mobile sensors provide complete coverage. But it also has the flavor of pursuit-evasion approaches as the sensors clear parts of the environment as they proceed to reach their final configuration. Communication and coordination aspects are also considered in this approach and the coordination of the movements to the desired positions is local and hence distributed. Particularly interesting is how the sensors maintain the line-of-sight wireless connectivity in this approach. One main assumption, whose removal is subject to further work, is that the agents are initially collocated. Even more closely related to pursuit-evasion is [BBH07] in which an interesting

approach for capturing intruders in a planar environment is presented. Here the targets can move with some speed that can potentially be larger than that of the robots. A team of robots aligns themselves in special configurations, so called trapping chains. The motivation stems from animals hunting in a pack encircling their prey. The team is attempting to capture a target by moving according to the reactive rabbit model (also used in probabilistic pursuit-evasion on graphs discussed in Section 2.3). Using this model for the movement of the target probabilistic bounds are established. While the assumption of a planar environment with many robot capturing a reactive rabbit target is not particularly realistic it is very interesting with regard to its connections to other fields. The trapping chain is solving a problem that has a game-theoretic pursuit-evasion twist. Namely, trying to constrain the possible movement of the evader making it impossible for it to escape. But the movement of these trapping chains themselves is similar to coverage algorithms. Using the approaches from coverage algorithms one could extend this to more complicated environments. Another very interesting paper with a more general flavor on distributed motion coordination is [MCB07]. The focus is on surveying theoretical tools for modeling, analysis, and design of motion coordination algorithms. They present most of the results shortly mentioned here.

Another valuable resource is the book [BCM09]. Therein general classes of distributed control algorithms for a variety of problems are presented. It puts an emphasis on the analysis of the communication topology and proven guarantees of performances under certain conditions. It contains a detailed presentation of rendezvous and connectivity maintenance algorithms and unifies a great deal of the existing literature into one notation and framework. It also adapts interesting notions of complexity to robotic networks such as time complexity, space complexity, mean and total communication complexity, and energy complexity. For

our purposes, the algorithms for boundary estimation and tracking are especially interesting since these could be modified to obtain a distributed control algorithm to have a team of robots follow a one-dimensional boundary, which could be a line from Line-Clear as discussed in Chapter 4.

2.8 Probabilistic Approaches

There are also a number of approaches that incorporate probabilistic motion models of targets and work similar to exploration algorithms by trying to visit locations close to the boundary of cleared space towards locations that have a high probability of a target being there. One such approach is presented by Moors et al. in [MRS05]. The contamination in a region is modelled by a Markov process so that likely positions of targets diffuse. The environment is modelled in a simple grid and bayesian filters are used to update the probabilities of a target being located in a grid cell given the diffusion process and sensor observations. The sensors have only limited range and work against the diffusion process. One interesting aspect of the algorithm from [MRS05] is the fact that a graph is created that covers the environment so that every point is visible at least from one node. For this random nodes are placed until the graph is covering the environment. On the graph an A^* search with a suitable heuristic is used to search for robot paths on the graph that reduce contamination. Since this approach is computationally expensive a partitioning of the environment with yet another heuristic is presented. The heuristic attempts to split the graph into roughly two equal parts with a minimal border. Then A-star is run on both parts sequentially while the border is guarded by some sensors. Simulation experiments are conducted on realistic environments to illustrate the practicability. In a follow-up in [MS06] Moors and Schulz present an improved Markov motion model that incorporates

so called *intended directions* (e.g. it may be more likely that a previous direction is maintained) when modeling the target trajectory. From an algorithmic perspective, however, and for the coordination of large numbers of robots the contributions of [MRS05, MS06] fall short. The approach does not scale to large environments and relies strongly on heuristics, yet it may be useful for smaller environments and hence as an implementation of the sweep action for vertices in Graph-Clear.

Another probabilistic approach, this time concerned with stationary targets is found in [CB07]. Therein Chung and Burdick present an approach that casts the search for one stationary intruder into a Bayesian decision framework. Their treatise focuses on the single searcher but they claim that it can be extended to multiple searchers without much ado. An optimal lookahead search determines a search trajectory that maximizes the detection probability within a window, similar to receding horizon control. An alternative strategy to determine a trajectory is presented as *saccadic* search. Here the trajectory goes from the cell with the highest maximal belief probability to the next. Yet another strategy to obtain trajectories is inspired by the Drospohila's search for food (fruit-fly) on a surface while being guided by visual sensory feedback. Here the trajectory is simply a line from a peak probability to the next closest peak reachable within a time-step. The simulations are done in a simple 10 by 10 grid. The prior belief for the target distribution is a simple Gaussian centered at a corner cell. The approach is easy and interesting in as much as its connections to exploration and Bayesian approaches goes.

Yet another probabilistic approach is found in [GTG05]. Therein the pursuit-evasion problem is cast into stochastic optimization problem that can be solved by the PARISH algorithm. Just as the other approaches this is used for a small

number of searchers in small environments and requires a considerable amount of computation and communication between agents.

It is obviously desirable to integrate a Bayesian perspective into robotic pursuit-evasion, considering that sensors are usually best modelled with a failure rate. Yet, most such attempts have trouble with scaling to large team sizes and environments. Our contribution in Section 3.8 attempts to fill this gap by unifying a probabilistic approach with Graph-Clear. Locally, i.e. within vertices or edges, one can then apply computationally expensive probabilistic techniques for the local coordination of robots while Graph-Clear scales the approach to large team sizes and environments. On another note, there is also an interesting relationship between exploration and pursuit-evasion that has not often been adressed in the literature since pursuit-evasion problems generally require a map. One notable exception is the online approach from [SRL04]. Extending other pursuit-evasion solutions to work without maps and combine them with exploration is obviously another desirable direction to pursue. A first attempt to do so for our Line-Clear problem is given in Chapter 6.

2.9 Sensor Networks and Tracking

Robotics researcher generally focus on motion and planning when dealing with pursuit-evasion tasks. Yet, there are other components that are relevant for the detection of targets and some of these, particularly target tracking and data association when dealing with many unreliable sensors, are discussed within the context of sensor networks. Sensor networks are a promising tool for surveillance applications, especially in environments that are controllable and where a lot of hardware can be deployed statically without malicious targets threatening to destroy it. Hence much of the research on sensor networks focuses on static

nodes. Other issues that are focused on are usually energy and communication constraints. The literature on sensor networks is too vast to be reviewed in breadth here, but we shall present a few selected publications that use methods that relate to ours.

One interesting approach for tracking is presented by Oh and Sastry in [OS05]. In terms of sensor inaccuracy this paper is on the far end of assuming very noisy sensors, i.e. faulty target detections. A method to integrate many such detections is presented by solving a hidden state estimation problem, via Hidden Markov Models (HMMs). Computation, track and storage information are distributed. It is suitable for indoors applications since it models potential trajectories with edges on the graph representing passages. The application of HMMs follows naturally. Some pruning strategies are presented to make the algorithm more efficient. The multi-object tracking assumes that the sensors can distinguish between targets which can be a strong assumption for many sensors. The methods presented in [OS05] can be very useful for merging sensor information from a sensor network and using the high-level detection for the team of mobile robots, who supposedly detect targets individually, i.e. often a target is seen by only one robot. Assumptions about likely target trajectories could improve the performance of the algorithm by reducing the state space. This corresponds to a dynamic graph structure for each target and may make the algorithm suitable for outdoor applications which otherwise would lead to a densely connected graph. Energy consideration play a rolo in [LL07] by Li and Liu which presents an approach for tracking targets with a mobile sensor network while trying to minimize energy consumption, i.e. travelled distance. Many more algorithms focused on tracking can be found in the book [BLK01] by Bar-Shalom et al. which is a comprehensive collection of target tracking methods.

Another paper from the sensor network direction introducing mobility is [SOS05]. Therein a sensor network is used to coordinate other mobile sensors. The detections of the sensor network are collected by a central instance which computes an assignment of pursuers to detected targets. The goal is to minimize the time to capture the targets, which are considered to be pieces of space debris. Interestingly, the vehicle dynamics are taken into consideration and play an important role in the calculation of the time to capture. Given the case that the number of pursuers is larger or equal to the number of evaders the problem can be solved satisfactory by a reduction to the linear bottleneck problem. Cooperation between static sensor networks and mobile robots is a promising area and can in practice lead to greatly reduced system cost. Batalin and Sukhatme also contributed to this idea and investigated task-allocation to a team of robots via wireless communication networks. They coined the term Distributed In-network Task Allocation (DINTA) and Multi Field DINTA (MF-DINTA). In [BS03b] they assumed a pre-deployed sensor network with a shared clock which guides robots with the help of alarms. Based on these alarms the network creates a navigation field for the robots to follow. With a similar spirit in [BS03a] Batalin and Sukhatme develop a coverage algorithm using markers that are dropped off by the mobile sensors. It explores a dynamic region. The number of markers the robot can drop is unlimited, which enables them to remove restrictions on localization capabilities while maintaining a competitive performance to previous approaches in which the robot could localize itself and the markers and navigate to markers. In [BSH03] Batalin, Sukhatme and Hattig use a sensor network to navigate a mobile robot. Experiments with a real network and a robot validate their approach. In [BS05] the previous work by Batalin and Sukhatme culminates into the so called LRV algorithm. It tackles the problem of deployment of the sensor network and exploration of the environment simultaneously. The acronym

LRV stands for least-recently-visited directions which illustrates how the robot moves around the deployed network while deploying further nodes. The deployed networks is modeled as a graph. Their algorithm is proven to be complete on graphs and optimal on trees. The cover-time is expressed in terms of the number of edges visited in order to visit each node at least once. The network, during its evolution, partitions the environment naturally into a graph which could then be searched with pursuit-evasion approaches.

The methods used for sensor networks, particularly for tracking and data-association, can be useful also to robotic pursuit-evasion. Interpreting the robots as a mobile sensor network these algorithms can disentangle multiple detection and provide a reliable count on how many targets cross over to the cleared part and are hence detected. They can be used to make sense of observations from multiple very noise sensors and, as seen in many of the example presented above, actual static networks can be used to provide robot teams with valuable information.

2.10 Real Systems

We have seen that pursuit-evasion problems are of interest to a number of different fields within and related to robotics. Naturally, practitioners designing robotic systems have also taken up the task to design multi-robot systems that excel at pursuit-evasion tasks.

One example is the Scout robot designed by a team at the University of Minnesota [DBC02]. The design is aimed at performing semi-autonomous surveillance, reconnaissance, or search and rescue missions with a clear focus on reconnaissance capabilities. Other papers of this group include [HER00, MRV05].

Another contribution was made in [RSE00] by Rybski et al. It presents a team of robots consisting of one *ranger*, a larger platform and smaller *scouts*. The paper focuses on the deployment of the scouts by the ranger. The scouts are then used to detect moving objects with a camera. The ranger communicates with all scouts. Most of the previous research had been done with single and highly capable platforms and this paper marked a first usage of a larger heterogenous robot team. In their scenario Rybski et al. use one ranger for deployment and another one for running the control processes for the scouts. The scouts act independently of each other and try to find dark regions from which they observer the environment. The map is a standard grid and is analyzed to identify doors. Into each door a scout is launched. This is continued until all scouts are deployed. The paper presents an interesting approach for a homogeneous robot team. The low level behavior on the scout to position itself autonomously are neat. It falls short on many other aspects such as uncontrolled deployment and scouts are deploys wherever they can, i.e. simply at each door. There is also no explicit coordination between agents apart from the transportation towards deployment by the ranger. The effectiveness of the system also suffered from reported noisy control when robots attempt to circle around to get 360° camera views. Additionally noise in communication lead to some problems for scouts behavior. To summarize, the system is an impressive starting point and exposed a lot of fundamental problems that have to be considered when designing practical systems.

Stoeter et al. in [SRE00] and [SRS02] continue from the work by Rybski. They propose an architecture considering the robots as dynamic resources. The control is based on a hierarchical behavior tree (on a location-transparent wireless network). They rely on a the *Common Object Request Brooker Architecture* (CORBA) [HV99] for distributed processing. They consider dynamic resource allocation. A *Resource Controller Manager* defines when behaviors can access

resources. Priorities are used to negotiated concurrent access. No real high-level control is proposed therein and the contribution is geared towards abstracted interfaces that one can use to design high-level controls.

Another real system is presented in [VSK02, VRS01]. Therein Vidal et al. use a probabilistic framework to cast the pursuit-evasion game and the map building into one problem, similar to the work discussed in Section 2.8. Their hybrid system architecture consist of two following layers of abstraction: 1) High-level pursuit policy computation, map building and inter-agent communication and; 2) Low-level tactical planning, navigation, regulation and sensing. They use the expected capture time as the performance metric. Computer vision is used to detect invaders. The probabilistic sensor model is simple and based on false negatives and false positives. Pursuers have perfect knowledge about their own location. Two greedy policies for controlling the robots are presented:

1. Local-max: maximize the probability of capturing an evader in the next time step considering only one-step reachable cells. This strategy is in general not persistent (i.e. improving).

2. Global-max: same as above, but considering the entire map. It is persistent on average.

The system architecture computes the pursuit policies centrally in the strategy planner. The map-builder is also central. All low-level tasks, including trajectory planning, are executed locally on the respective agent. From simulations they compute mean capture times across trials and show that it is more difficult to capture fast evasive targets than slow and randomly moving targets. Furthermore, it is shown that global-max outperforms local-max. The key part of this approach is the high-level probabilistic control. The entire team maintains a cen-

tralized map of the environment including the probability distribution of possible locations of an evader. The control moves the agents into the spots with the highest likelihood of an evader being located. Similar to [MRS05] the target motion model is a simple Markov process. And also here the better policy, global-max, does not scale well and grows exponential with the problem size. Furthermore, the key parts of the coordination are centralized.

The paper by Pugh and Martinoli [PM07] sheds some light on the relationship between high-level abstractions and practical implementation. More precisely, they pick the example of a simple multi-robot search algorithm and illustrate the practical problems encountered when implementing the high-level method. They use Webots [Mic04] for their simulation. The arena is a rectangle with one target located inside. They present three approaches, one coordinated (i.e. precise) approach and two random approaches. The perform two sets of experiments, one with a perfect sensor and one with a probabilistic sensor model. They show that the benefit of the coordinated (precise) approach is smaller for the second set. Another set of experiments with positional noise shows that the coordinated approach suffers the most from this noise when there are only few robots. With more robots the differences between the random and coordinated approaches diminish. The main contribution of the paper is to show the importance of incorporating uncertainty in distributed, coordinated multi-robot algorithms to reflect the challenges of a real-world application. Indeed most of the high-level approaches do not incorporate such assumptions. Some of them attempt to be robust to such problems (noise, individual robot fault) by maintaining a simplistic approach such as in [Par02]. It is, however, possible to extend precise and coordinated approaches to incorporate imperfect control, noisy localization and noisy sensors. The visibility-based approach in [SRL04] is such an example where noisy control can be dealt with, when within certain bounds and we shall make a similar

attempt in Chapter 6.

2.11 Other Related Fields

Apart from the selected areas presented above there are a few other related areas that warrant mentioning. One such area is known as covert robotics and refers to research done by Marzouqi and Jarvis in [MJ05]. They consider a path-planning problem given the requirement that the probability for detection by given sensors placed at known locations in the map is lowest. The sensor properties for detection have to be known in advance. They compute a visibility risk map in which the target can then search for a low-risk path that avoids visibility by the sensors. Such approaches may be of interested when computing the worst-case target path for our probabilistic method from 3.8 which needs access to the worst-case probability of missing a target.

Another very interesting area is that of aerial pursuit. The added complications relate to the dynamics of the aerial vehicle, usually denoted as an unmanned aerial vehicle (UAV), which are often severe. Recently in [Fre07], Frew presented a method to coordinate two aerial vehicles on their flight paths in order to keep a moving target under surveillance. Another aerial approach, this time with swarms is contributed by Hexmoor et al. in [HMB05]. Multiple unmanned aerial vehicles are given the task to scan a set of targets, i.e. cover them a defined period of time while allowing occasional interruptions. The control is distributed and target assignment is based on negotiation. Movement is modeled with two degrees of freedom in a plane, somewhat unusual for aerial approaches. Intervention by a human operator regarding the target assignment is allowed. They introduce concepts such as user suggestion and intervention, role negotiation, agent personality and load balancing. User suggestion is defining regions of higher interest that the

vehicles can consider moving to. Role negotiation refers to the target assignment amongst the vehicles. Parameters, called conformity, sociability, commitment, disposition and target bidding define how a vehicle reacts to user suggestions, peer-density (avoiding or seeking other vehicles), target loss and how long it will attempt to re-acquire a target, continue to track a target despite being outbid and bidding in which the closest first bid wins (i.e. the closest vehicle to its closest target gets the target). Losing a bid results in wandering around until the next bidding round. These heuristics are similar to those presented in [KC07a] which uses help calls and bidding for target assignments. Another system which focuses on shared control of multiple UAVs by human and automated agents for search and rescue tasks is presented in [BG08]. All agents are submitting their request for aerial photographs and the system computes an optimal assignment of these images to UAVs. Then the photographs are inspected with regard to the presence of a desired target to allow the agents to update their belief about the targets location. The search proceeds in multiple round with multiple requests.

Another interesting area is the prediction of target trajectories. This obviously plays a role for probabilistic approaches that incorporate target motion models and for direct pursuit. Some results and methods for trajectory prediction for sensor networks are presented in [YTW05]. The predicted trajectory is used to conserve energy of the sensor by only activating them when a trajectory of a target is expected but the methods can potentially be used in other circumstances. In yet another approach, Gauss-Markov models have been developed in [LH99] and [LBC98] for special types of sensor networks. Furthermore, interactions on targets are taken into account in the design of the movement model using a Markov Chain Monte Carlo based particle filter presented in [KBD03]. Another interesting and more flexible approach is presented in [EG98]. The goal in [EG98] is to provide a path planner with possible colliding paths of moving objects. An

autoregressive model with parameters estimated with a conditional likelihood method based on previous observations of the targets motion is used. It even deals with rotations of non-circular objects, assuming reference points on the object. In [MS03] further results from this area of research are presented and integrated into a framework that classifies targets. The classification enables the algorithm to identify the dynamics from hard-coded records and it then predicts the movement with this additional information. For the prediction it also uses an Extended Kalman Filter [Kal60]. Another approach focused on training HMMs is presented in [VFL09] with an emphasis on human and vehicle motion.

Finally, there is a connection between pursuit-evasion without known maps and exploration algorithms that is a promising area for further investigations. Here Burgard et al. [BMF00] made significant contributions that are a good first pointer for work in this direction.

CHAPTER 3

Graph-Clear: Multi-Robot Pursuit-Evasion on Graphs

In this chapter we introduce a novel pursuit-evasion problem on graphs, called Graph-Clear, to tackle combinatorial problems that arise in multi-robot pursuit-evasion. A graph model is particularly suitable since it scales well with larger environments and we can abstract away from the geometrical features of an environment by representing its topology as a graph. As presented in Section 2.1 there is a vast amount of literature on graph-based pursuit-evasion and a great many variants of formulations. Yet, weighted graphs received a rather limited attention and previous models are not well suited for multi-robot pursuit-evasion. This is discussed in more detail in Section 3.1.

In Graph-Clear an environment is represented by a weighted graph on which one can execute *sweep* actions on vertices and *block* actions on edges. A sweep action detects all intruders in a vertex, while a block action detects intruders that attempt to cross an edge. It is assumed that all edges incident to a vertex are blocked while the sweep operation is executed. These actions represent routines that the robot team can execute in the actual environment. Because of the limited sensing hypothesis, more than a single robot is in general necessary to successfully perform these intruder detection operations. Weights on vertices and edges therefore associate a cost to each action denoting the number of robots

needed to execute it. The goal of Graph-Clear is to find a sequence of these actions, a so-called strategy, that detects all intruders in the environment using the least number of robots. Intruders are assumed to be omniscient and capable of moving at unbounded speed. In particular, they are assumed to have full knowledge of the environment, of the pursuers positions, and even of their strategy. We represent the possibility of an intruder being located somewhere with the concept of contamination. Initially the entire graph is contaminated and each sweep and block clears contamination from vertices and edges. The task of finding any intruder is equivalent to removing all contamination. Since intruders have full knowledge, recontamination of previously clear vertices or edges occurs whenever an intruder has a path to that vertex or edge on which it cannot be detected.

To apply Graph-Clear and use strategies for the coordination of a real robot team one needs to solve two subproblems. First, one has to provide implementations of sweep and block actions. These can differ widely and depend on the type of environment, robot platform, and sensors. Hence, they often require committing to a particular sensing model or hardware. For a vertex, the corresponding implementation for the sweep action has to guarantee the detection of any intruder inside the region that the vertex is associated to, given that no intruder can enter or leave the region while sweeping takes place. Similarly, an implementation of a block action on an edge has to guarantee that no intruder can cross it undetected. The second subproblem is the automatic extraction of graphs from a given environment. This is not strictly necessary since graphs can be generated manually, but it is highly desirable for most applications and it opens further interesting research questions. Some results on extracting graphs and weights from occupancy grid maps and implementations for sweep and blocking routines are presented in Chapter 5. Another useful application of Graph-Clear is in computing solutions to the Line-Clear problem discussed in Chapter 4.

The main motivation for Graph-Clear is the design of robot algorithms that will ultimately run on large robot teams. Yet, this chapter is primarily devoted to the formalization and analysis of Graph-Clear as a formal pursuit-evasion problem on a graph. This formalization allows us to address the main computational challenges resulting from the consideration of large environments and large robot teams in a formal and deterministic setting. An extension of Graph-Clear to probabilistic scenarios with faulty sensors and imprecise actuators in presented in Section 3.8. More precisely, this Chapter presents the following original contributions:

1. Graph-Clear is rigorously formalized. This formalization allows to exploit a number of formerly developed results in related literature (Section 3.2).

2. We prove that the decision version associated with the Graph-Clear problem is NP-hard (Section 3.3).

3. For the special case of contiguous strategies which ensure that all intruder-free vertices are always connected, we prove that recontamination does not help (Section 3.4).

4. Given that for the general case of graphs the problem is NP-hard, we focus our attention on trees. In Section 3.5 we start presenting some terminology useful for clearing trees and present simple algorithms that compute strategies.

5. In Section 3.6 we present an algorithm to produce optimal contiguous strategies for trees. The algorithm is shown to use the least number of robots, and has time complexity quadratic in the number of vertices.

6. A more complicated algorithm for the case when cleared vertices do not have to be connected is presented in Section 3.7.

7. Section 3.8 presents a probabilistic extension of Graph-Clear and algorithms to compute strategies that guarantee the detection of all targets with a desired probability.

8. Section 3.9 discusses a modification to Graph-Clear that considers sweep actions on vertices that also prevent recontamination.

9. The problem of applying the algorithms for trees to graphs with cycles is shortly addressed in Section 3.10.

The chapter closes with conclusions in Section 3.11. The results presented in this chapter have been published separately in [KC07b], [KC07c], [KC08b], [KC09a], [KC09b].

3.1 Motivation

There is a rich literature concerning a variety of pursuit-evasion problems on graphs, also often referred to as graph-searching. Graph-searching and its variations also require solutions to the subproblems, mentioned in the previous section, if one aims to utilize them to coordinate robot teams. The edge-searching problem [Par76] is perhaps the most prominent and oldest of these problems, and it is the most closely related to the model we present in this paper. To motivate the introduction of Graph-Clear, we now describe weighted edge-searching and its differences to Graph-Clear in more detail than in Section 2.1. The edge-searching problem asks to determine a sequence of moves detecting all intruders in a graph using the least number of robots. A move consists of either placing or removing a robot on a vertex, or sliding it along an edge. A vertex is considered guarded as long as it has at least one robot on it, and any intruder located therein or attempting to pass through will be detected. A sliding move detects any intruder

on an edge. In the weighted variant weights on vertices denote the number of robots needed for each vertex to be considered guarded, while weights on edges denote the number of robots needed for a slide move to detect all targets [BFF02]. Consequently, for each move in a sequence one needs to additionally specify how many robots are used for it.

The key differences between weighted edge-searching and Graph-Clear are in the requirements imposed for the implementation of basic operations. To apply edge-searching one needs to provide implementations for guarding and sliding, while in Graph-Clear one needs to implement sweeping and blocking. An implementation of guarding has to guarantee that all intruders in the associated region for the vertex are detected and furthermore that no intruder can enter or exit the region undetected. Sweeping does not require the latter. Instead, in Graph-Clear while sweeping a vertex we require a block on each edge to prevent targets from entering or exiting. The consequence is that some robot algorithms cannot be used for guarding operations. The example in Fig. 3.1 uses an algorithm for detecting targets inside the region of a vertex that does not satisfy the guarding requirements and is hence not directly suitable for edge-searching. To satisfy the guarding requirements one would have to augment the algorithm by additionally positioning robots at the entrances. Then the cost of this combined routine becomes a weight in weighted edge-searching which represents the number of the robots needed to search the region and to keep entrances covered. But, once the robots searched the region and hence cleared the vertex we still have to guard the vertex to prevent recontamination of its neighbors. In practice this continued guarding after the actual search does not need to involve any of the robots that performed the search, but only those covering the entrances. But in weighted-edge searching we still pay the full cost for the guarding operation. This is because in edge-searching guarding of a vertex performs two basic functions,

namely the prevention of spreading of contamination from and to its neighbors, and additionally the detection of all intruders in the vertex. One could try to overcome this problem by having weights on edges represent the cost of entering a vertex, searching and covering the entrances while the weight on the vertex only represent the cost of covering the entrances. But then sliding along an edge costs more than guarding the vertex. Not only is this unintuitive, but the formulation of weighted edge-searching from [BFF02] does not allow edge weights larger than the weight of the adjacent vertex.

Even if the above problem was remedied there is yet another problem. Suppose we clear the center vertex and then one of its neighbors. At this point the entrance to the neighboring vertex does not need to be blocked any further and the weight for guarding the center vertex should change to reflect this. Since weights are fixed this cannot be captured. There is no immediate remedy for this since in edge-searching the spreading of contamination is avoided only by actions on vertices and never on edges. In Graph-Clear, on the other hand, the search of a region and the prevention of recontamination from neighboring regions are separated and the latter occurs on edges. In colloquial terms, in edge-searching the intruder movement is restricted by robots in vertices while in Graph-Clear we move this capability to the edges, effectively disentangling detection in a vertex from the prevention of recontamination. Evidently, this is useful for vertices that represent complex regions and edges that are simple connections between these regions. On the other hand, edge-searching is useful for simple vertices and complex connections between these, such as the network of tunnels example often mentioned as a motivation for edge-searching. Another important distinction between weighted edge-searching and Graph-Clear is discussed in detail after the definitions for Graph-Clear in Section 3.2.1. It relates to unintuitive consequence that the addition of weights to the traditional edge-searching problem has.

Namely, that allowing simultaneous moves can improve solutions to the weighted edge-searching problem.

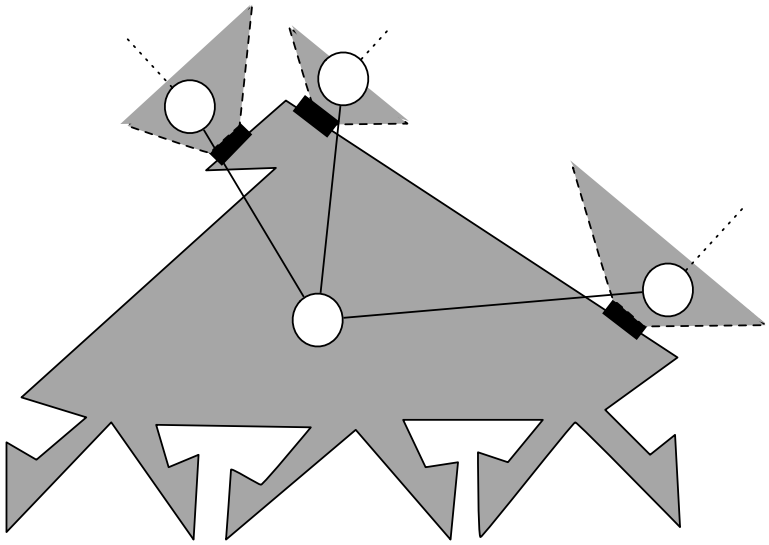


Figure 3.1: An example that illustrates how a graph for Graph-Clear can relate to an actual environment. The environment is shown in grey with its graph embedded. All weights in this example are equal to one. Connections between regions that are connected by edges are shown in black. The center region is the "eagle" example redrawn from [SRL04]. It can be cleared using the algorithm from [SRL04] with only one robot and a simple gap sensor with sufficiently large range. During its execution it recontaminates the top part of the region and hence cannot guarantee that no target enters the vertex undetected. We hence need blocks on the edges, i.e. to position sensors on the black regions. Note that the entire environment can be very large so that the sensor only satisfies the large range assumption within a vertex.

3.2 Problem Formulation

A formal definition of Graph-Clear is presented in this section. It is assumed that the reader is familiar with graphs and trees, and is referred to [CLR01] for the basic notation and terminology we use. The first part presents a formulation in terms of graph theory concepts, and the language is chosen accordingly. The connection between Graph-Clear and real world problems is presented in Section 3.2.1.

Definition 1 (Surveillance graph) *A* surveillance graph *is a triple* $G = (V, E, w)$*, where* (V, E) *is an undirected graph with vertex set* V*, edge set* E*, and* $w : V \cup E \rightarrow \mathbb{N}^+$ *as a weight function*[1]*. Vertices and edges in a surveillance graph have a state. The state of a vertex can be* clear*, or* contaminated*, while the state of an edge can be* clear*,* contaminated *or* blocked*. If* x *is a vertex or an edge, its state is indicated as* $\nu(x)$*.*

Notation: Depending on the context, edges will be indicated either as e or as (v_i, v_j), with $v_i, v_j \in V$. Throughout the paper the notation (u, w) indicates an undirected edge between vertices u and w. If v is a vertex, $edges(v)$ is the set of all edges having v as end point. The degree of a vertex v is the number of edges having v as end point, i.e. $degree(v) = |edges(v)|$. If G is a graph, $V(G)$ is its set of vertices and $E(G)$ the set of edges. Also, possible state values will be abbreviated using the letters $\mathcal{R}$ for clear, $\mathcal{C}$ for contaminated, and $\mathcal{B}$ for blocked.

Assumption: from here onwards unless otherwise stated we shall assume that $|V| = n$ and $|E| = m$.

[1] *In this manuscript* $\mathbb{N}^+$ *indicates the set of positive natural numbers, while* $\mathbb{N}$ *indicates the set of natural numbers (i.e. including 0).*

Definition 2 (State space and state of a surveillance graph) *The* state space *of a surveillance graph G is the set*

$$\mathcal{V}(G) = \{\mathcal{R}, \mathcal{C}\}^n \times \{\mathcal{R}, \mathcal{C}, \mathcal{B}\}^m.$$

The state *of the surveillance graph G is an element $\nu = \{\nu_1, \ldots \nu_{n+m}\} \in \mathcal{V}(G)$ such that $\nu_i = \nu(v_i)$ for $i = 1 \ldots n$, and $\nu_{n+j} = \nu(e_j)$ for $j = 1 \ldots m$.*

The state of a graph is a string of symbols from the alphabet $\{\mathcal{R}, \mathcal{C}, \mathcal{B}\}$ indicating the state of every vertex and edge. The first n symbols indicate the state of vertices, and the last m the state of edges.

Definition 3 (Recontamination Path) *Let G be surveillance graph with state ν, and let $x, y \in V \cup E$. A* recontamination path *between x and y is a path of edges and vertices on which no edge is blocked, i.e. for all edges e_i of the path we have $\nu(e_i) \neq \mathcal{B}$.*

The reader should note that while defining the concept of recontamination path we generalize the usual definition of path in a graph. Rather than defining paths only between a couple of vertices we also consider paths between edges, and between a vertex or an edge, or vice versa. In every situation we impose that the edges along the path should not be blocked. Two types of actions can be applied to a surveillance graph, namely *blocking* on edges and *sweeping* on vertices. The goal of these actions is to change the state of vertices and edges so that no contaminated edges or vertices remain. While blocking can be applied to any edge, sweeping can be applied to a vertex v only if all edges originating from v are blocked.

Definition 4 (Action set and actions) *The* action set *of a surveillance graph G is the subset of $\{0, 1\}^{n+m}$ where each element $a = \{a_1, \ldots, a_{n+m}\}$ (called* action*) satisfies the following constraint:*

- *if* $a_i = 1$ *with* $1 \leq i \leq n$, *then* $a_{n+j} = 1$ *for each edge* $e_j \in edges(v_i)$

If $a_i = 1$ *with* $1 \leq i \leq n$, *we say that the action* a *sweeps vertex* v_i, *and if* $a_{n+j} = 1$ *with* $1 \leq j \leq m$ *we say that action* a *blocks edge* e_j. *The action set of* G *is indicated as* $\mathcal{A}(G)$.

The constraint imposed in the above definition ensures that all edges are blocked while a vertex is being swept. Instead of imposing this as a constraint one could also show that it follows from the definition of recontamination. For the sake of brevity and clarity we choose to impose it.

Definition 5 (Sweeping and blocking cost) *Let* G *be a surveillance graph. The* sweeping cost *of a vertex* $v \in V$ *is* $w(v)$, *while the* blocking cost *of an edge* $e \in E$ *is* $w(e)$.

Definition 6 (Cost of an action) *Let* G *be a surveillance graph and let* $a \in \mathcal{A}(G)$ *be an action. The* cost of action *a is:*

$$c(a) = \sum_{i=1}^{n} a_i w(v_i) + \sum_{j=1}^{m} a_{n+j} w(e_j)$$

The former definition states that the cost of a is the sum of the blocking and sweeping costs for the edges blocked and vertices swept by a. It follows that the total cost of sweeping a single vertex v is the following

$$s(v) = w(v) + \sum_{e_j \in edges(v)} w(e_j) \tag{3.1}$$

because in order to sweep a vertex it is necessary to block all its edges.

Definition 7 (Transition function) *Let G, $\mathcal{V}(G)$ and $\mathcal{A}(G)$ be defined as above. The* transition function *ζ maps a state and an action into a new state:*

$$\zeta : \mathcal{V}(G) \times \mathcal{A}(G) \to \mathcal{V}(G).$$

Given $a \in \mathcal{A}(G)$ and $\nu \in \mathcal{V}(G)$, the new state ν' is defined as follows:

1. *if $a_{n+j} = 1$, $1 \leq j \leq m$, then $\nu'(e_j) = \mathcal{B}$*
2. *if $a_i = 1$, $1 \leq i \leq n$, then $\nu'(v_i) = \mathcal{R}$*
3. *if $\nu_{n+j} = \mathcal{B}$, $a_{n+j} = 0$, $1 \leq j \leq m$, and no recontamination path between e_j and $x \in V \cup E$ with $\nu(x) = \mathcal{C}$ exists, then $\nu'_{n+j} = \mathcal{R}$*
4. *if there exists a recontamination path between $x \in V \cup E$ and $y \in V \cup E$ with $\nu(y) = \mathcal{C}$, then $\nu'(x) = \mathcal{C}$*
5. *$\nu'_i = \nu_i$ otherwise.*

The transition function establishes how the state of G changes when an action is applied. The five cases can be described in words as follows:

1. edges where a block action is applied become blocked;
2. vertices where a sweep action is applied become clear;
3. blocked edges where a block action is not applied anymore become clear if there is no recontamination path involving them;
4. vertices or edges for which a recontamination path towards a contaminated vertex or edge exists become contaminated;
5. vertices or edges maintain their previous state if none of the former cases apply.

Definition 8 (Strategy) *Given a graph state $\nu \in \mathcal{V}(G)$, a* strategy $\mathcal{S}$ *for ν is a sequence of actions $a_1, a_2, \ldots, a_k$ that when applied in sequence clears all elements in G, i.e.:*

$$\underbrace{\zeta \ldots \zeta(\zeta}_{k \text{ times}}(\nu, a_1), a_2) \ldots, a_k) = \underbrace{\{\mathcal{R}, \mathcal{R}, \ldots, \mathcal{R}\}}_{m+n \text{ times}}$$

Definition 9 (Cost of a strategy) *Let $\mathcal{S} = \{a_1, \ldots, a_k\}$ be a strategy. The* cost of strategy $\mathcal{S}$ *is*

$$ag(\mathcal{S}) = \max_{i=1\ldots k} c(a_i) \tag{3.2}$$

We now can formally define the Graph-Clear problem.

Definition 10 (Graph-Clear problem) *Let G be a surveillance graph whose state is $\nu = \{\mathcal{C}, \mathcal{C}, \ldots, \mathcal{C}\}$. Determine a strategy $\mathcal{S}$ for ν of minimal cost.*

From now onwards, before a strategy is applied to a surveillance graph we assume that the state of all its elements is $\mathcal{C}$, unless stated otherwise. We distinguish between two types of strategies using the concept of contiguity adapted from [BFF02].

Definition 11 (Contiguous and non-contiguous strategies) *Let G be a surveillance graph in state $\nu = \{\mathcal{C}, \mathcal{C}, \ldots, \mathcal{C}\}$. A strategy $\mathcal{S}$ for G is contiguous if the subset of cleared vertices and cleared or blocked edges always forms a connected subgraph of G. Otherwise, a strategy is said to be non-contiguous.*

This distinction is useful, since contiguous strategies on trees turn out to be easier to compute, as presented in Section 3.6. Another type of strategy is the following.

Definition 12 (Progressive strategy) *A* progressive strategy *is a strategy for a surveillance graph G in state $\nu = \{\mathcal{C}, \mathcal{C}, \ldots, \mathcal{C}\}$ preventing recontamination.*

In particular, we will later on concentrate our study on progressive contiguous strategies.

Definition 13 (Cost of a graph) *Let G be a surveillance graph and let $\mathcal{S}$ be a strategy of minimal cost for G. We define the* cost of graph G *as $ag(G) = ag(\mathcal{S})$.*

Similarly for a graph G in any state ν we write $ag(G, \nu) = ag(\mathcal{S})$, where $\mathcal{S}$ is a strategy with minimal cost for G in state ν. For a subset of vertices $U \subseteq V$ let $a_1, \ldots, a_k$ be a sequence of actions that clears all vertices of U starting from state ν with minimal cost $\max_{i=1\ldots k} c(a_i)$. [2] We write $ag(U, \nu)$ for this minimal cost.

The cost of a strategy is the number of robots needed to clear the environment according to the sequence of actions defined by the strategy. Since we seek strategies with the least number of robots, we will consider only strategies for which at most one vertex at the time is swept. This approach is justified by the following lemma whose simple proof is omitted.

Lemma 1 *Let $\mathcal{S} = \{a_1, \ldots, a_k\}$ be a strategy for G. Then there exists a strategy $\mathcal{S}'$ with cost $ag(\mathcal{S}') \leq ag(\mathcal{S})$ that sweeps no more than a vertex at a time.*

3.2.1 An example of Graph-Clear

We now work out a simple example outlining the connection between Graph-Clear and practical surveillance scenarios. A surveillance graph is used to model complex environments. For our illustration we choose to intuitively associate vertices with rooms, and edges with connections between rooms (i.e. doors or corridors). Fig. 3.2 shows a simple indoor environment and one possible surveillance graph. A contaminated vertex is a room that may hide an intruder, while

[2] Such sequences are not necessarily strategies, unless $U = V$, but they can be thought of as partial strategies.

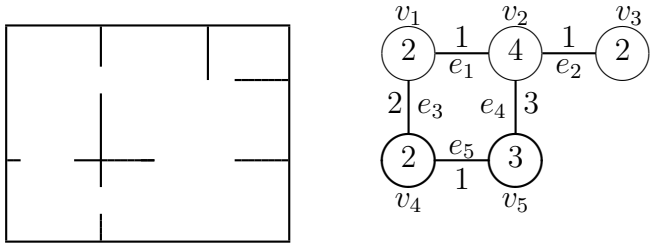


Figure 3.2: An example environment and one possibly associated surveillance graph. Numbers inside vertices are the sweeping costs, and numbers on the edges are blocking costs.

a clear vertex is known to be intruder free. Intruders may also hide in connections between rooms, therefore edges can also be clear or contaminated. Robots equipped with sensors are used to detect intruders [3]. In our problem formulation, an intruder disappears as soon as it is detected by a robot (i.e. it falls within its sensing range). A blocking operation applied to an edge ensures no intruder can pass through that connection without being detected by the robots blocking it. Since our focus is on scenarios involving robots with limited sensing capabilities, more than one robot may be necessary to block large connections (see the values displayed in Fig. 3.2). To detect all possible intruders inside a room a sweeping operation is performed. Once again, since robots have limited capabilities, multiple robots are requested to make sure one room is free of intruders. Since a room may have multiple entrances, it is necessary to block all of them to prevent intruders from entering parts of the room that have been swept already (recontamination). The recontamination path concept adds significant challenges and asymmetries to the problem. In fact, while we assume that robots are capable of detecting intruders only when they fall within a limited sensing

[3]The physical sensor used can vary widely across applications and depends on what the target of interest is. Often cameras, lasers and sonars are used for actual experiments with robots, but also acoustic or infrared sensors can be useful.

range, our definition of recontamination implies that as soon as recontamination is possible, it immediately occurs. We therefore suppose intruders have complete knowledge of the environment and of the robots' positions, and they may move with unbounded speed on continuous paths to take instantaneous advantage of the existence of recontamination paths. The Graph-Clear problem asks how to schedule sweeping and blocking operations so that eventually the environment is completely cleared using the least number of robots. Fig. 3.3 shows a possible strategy to solve the Graph-Clear problem on the environment depicted in Fig. 3.2.

$\nu(G)$	a	$c(a)$
$\mathcal{CCCCC}$ $\mathcal{CCCCC}$	10000 10100	5
$\mathcal{RCCCC}$ $\mathcal{BCBCC}$	00010 10101	6
$\mathcal{RCCRC}$ $\mathcal{BCBCB}$	01100 11011	12
$\mathcal{RRRRC}$ $\mathcal{BBRBB}$	00001 00011	7
$\mathcal{RRRRR}$ $\mathcal{RRRBB}$	00000 00000	0
$\mathcal{RRRRR}$ $\mathcal{RRRRR}$		

Figure 3.3: A possible strategy to solve the Graph-Clear problem associated with the graph shown in Fig. 3.2. The first column displays the status, the second the applied action, and the third the cost. The reader should note that in the third row an action sweeping two vertices at the same time is applied, and that a final action removing all blocks is executed in the end (with 0 cost). The cost of this strategy is 12, i.e. the maximum value read in the third column. It is easy to see that such strategy is not optimal.

Before Graph-Clear can be of practical relevance in a robotic scenario, it requires a method to partition an environment into regions which then become vertices in the surveillance graph as well as implementations for the sweeping

and blocking actions. Some solution for extracting surveillance graphs from occupancy grid maps are presented in Chapter 5. One of these methods is based on detecting narrow parts of the environment using its Voronoi Diagram. It not only extracts the graph, but it also assigns appropriate weights based on given sensing abilities of the robots, and a predetermined clearing method for vertices. From now onwards we assume that for a given sensor model the corresponding surveillance graph is available.

Before proceeding it is worth to consider a fundamental difference between weighted edge-searching and Graph-Clear, apart from the motivation presented in Section 3.1 and Fig. 3.1. This difference results from the addition of weights. Recall that in edge-searching a single move is either 1) moving along an edge, 2) placing a robot on a vertex or 3) removing a robot from a vertex. In weighted edge-searching these moves additionally receive an integer representing the number of robots participating in the move. The weight on the edge or vertex determines how many robots are needed so that the move clears the edge and guards the vertex. This seems like a straightforward extension of the previous model, but it has unintuitive consequences. Consider the example in Fig. 3.4. Therein, allowing simultaneous moves can improve the number of robots needed. This results from the fact that the guarding operation on one vertex can need more than the sum of the slide and guarding operations towards all neighbors.

3.3 The Complexity of Graph-Clear

The theorem claiming NP-hardness of the Graph-Clear problem was first presented [KC07c], but the full proof was omitted due to space constraints. We here offer the complete proof based on the new notation developed in Section 3.2. The proof is mainly an adaption of the proof of NP-completeness of edge-search on

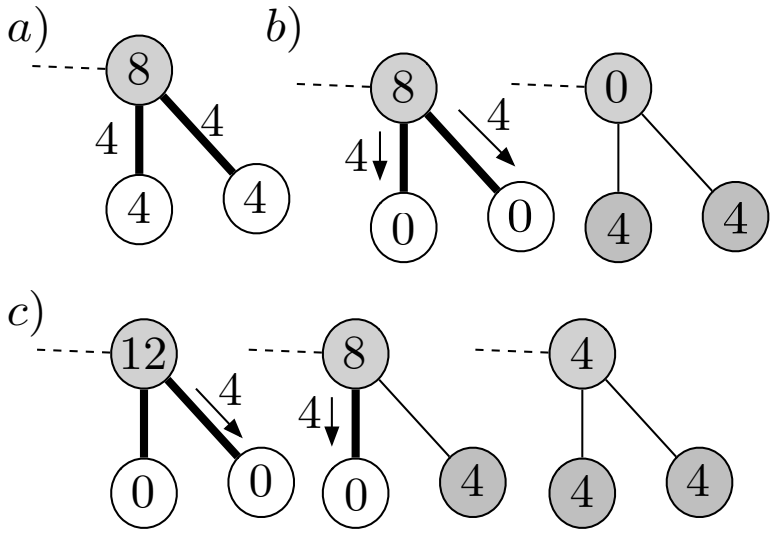


Figure 3.4: An example that illustrates the consequences of allowing simultaneous moves in weighted edge-searching. Part a) shows a graph with its weights. Part b) shows the graph with eight robots on the top vertex and none in the bottom vertices. The arrows indicate two sliding moves with four robots that finish clearing the graph with eight robots when executed simultaneously. Part c) shows how to clear the graph with strictly sequential moves with the same recontamination rules but needing more robots.

a graph presented in [MHG88]. The key idea is to substitute complete graphs used in [MHG88] with stars defined in the following. The method constructs a surveillance graph with all weights equal to one and the statement hence also holds for the simpler case of unweighted surveillance graphs. Throughout this section all weights are assumed to be equal to 1.

We first define the concept of *stars* and prove a bound on the clearing cost.

Definition 14 (Stars) *A* star *of order n, written G_n, is a surveillance graph that is a tree with $n+1$ vertices $v_0, \ldots, v_n$ and n leaves. The vertex v_s that is not a leaf shall be called* center.

Lemma 2 *Let G_n be a star of order n. Then $ag(G_n) = n + 1$.*

Proof: Let v_0 be the center of G_n. According to Eq. 3.1 the cost to clear v_0 is $n + 1$. To clear G_n with cost $n + 1$, clear v_0 first and block all its n edges. Then use the $n + 1$-th robot to clear each leaf, (see Fig. 3.5).□

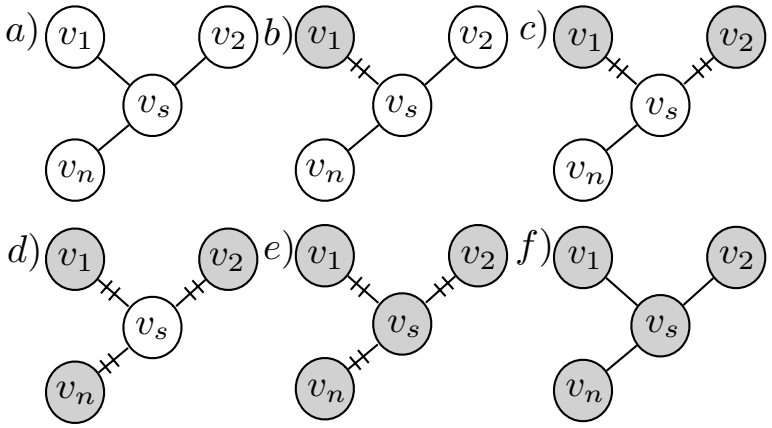


Figure 3.5: The construction of an optimal strategy for a star. Cleared and contaminated vertices are grey and white respectively. Blocked edges are marked with as double-stroked line. First all leaves are cleared, leaving the edge to the leaf blocked. For leaf v_i,$i = 1, \ldots, n$ the total cost while clearing it is $i+1$. Finally the center vertex is cleared with cost $n + 1$.

Let us consider the following decision version of the Graph-Clear problem:

INSTANCE: $G = (V, E, w)$, a surveillance graph with $w(x) = 1\ \forall x \in V \cup E$, and a natural number P

QUESTION: is $ag(G) \leq P$?

Proving that this problem is NP-hard relies on a polynomial reduction to the *min-cut into equal sized subset* problem (MCIESS from now on), a problem known to be NP-complete [GJ79]. MCIESS is posed as follows.

INSTANCE: An undirected graph $G = (V, E)$ with an even number of vertices,

and a natural number K.

QUESTION: is there a partition of V into two subsets V_1 and V_2 with $|V_1| = |V_2| = \frac{1}{2}|V|$ such that $|\{(u,v) \in E : u \in V_1, v \in V_2\}| \leq K$?

Theorem 1 *The decision version of Graph-Clear is NP-hard.*

Proof: Let $G = (V, E)$ and $K > 0$ be an instance of the MCIESS problem. Let $n = |V|, d = \max_{v_i \in V}\{degree(v_i)\}$, $N = 6(d + K)$ and $M = nN(n + 2)$. An instance $H = (U, F, w), P$ of the Graph-Clear problem is built in polynomial time as follows:

1. for each $v_i \in V$ create a star of order M called C_i (i.e. $C_i = G_M$, with $i = 1 \ldots n$)
2. let C_A be an additional star of order M (i.e. $C_A = G_M$)
3. add edges between leaves of the star with at most one edge for each leaf:
   (a) add nN edges for each pair C_i, C_j, $i \neq j$, note that $i, j \in \{1, \ldots, n, A\}$
   (b) add N more edges between each C_i and C_A
   (c) add 3 more edges between C_i and C_j if $(v_i, v_j) \in E$
4. $w(x) = 1 \ \forall x \in U \cup F$
5. $$P = (M + 1) + \left(\frac{n}{2}\right)^2 nN + 3K$$

Note that it is possible to give each leaf v of a C_i at most one edge since we have M such leaves and never add more than $M - 1$ edges in total to any C_i. Fig. 3.6 visualizes the construction. All vertices that received an edge during the construction will be called *connectors*, and all that did not remain leaves. There

is at least one leaf remaining for each C_i. We now show that the Graph-Clear instance H, P admits a positive answer if and only if the MCIESS instance G, K does.

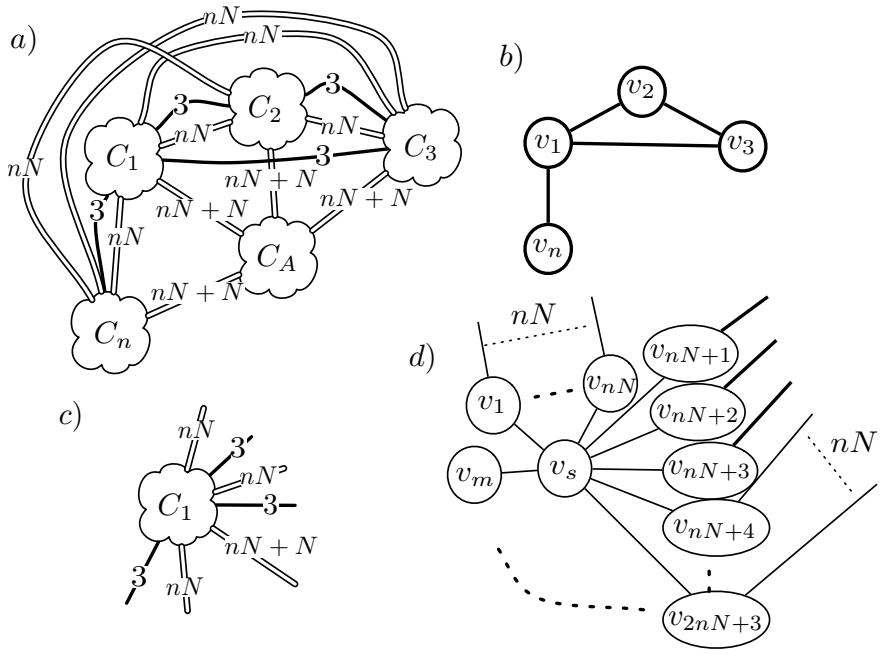


Figure 3.6: An illustration of the large graph constructed from an instance of the MCIESS. Part a) shows the constructed surveillance graph from the MCIESS graph in part b). A star is represented by a cloud, a bundle of nN or more edges by a double line and 3 edges by a thick line. Part c) is a close-up of the star C_1 and its edges to other star. In part d) C_1 is shown in more detail with its center, connectors and leaves.

Assume the instance G, K admits a positive answer, i.e. we have a partition of V into V_1 and V_2 s.t. $K' \leq K$ edges connect $V_1 = \{v_1, \ldots, v_{n/2}\}$ and $V_2 = \{v_{n/2+1}, \ldots, v_n\}$. Let us then consider the following strategy $\mathcal{S}$ for H: clear $C_1, \ldots, C_{n/2}, C_A, C_{n/2+1}, \ldots, C_n$ in this order. Being more specific, according to lemma 2, the cost to clear the leaves and center of C_1 is $M + 1$. The number of

edges from C_1 to other C_is is at most $n^2N + N + 3d < M$ and hence there is at least one vertex in C_1 with degree one. We start clearing C_1 by clearing each degree-one vertex. Then we clear the center and keep all its edges blocked. This procedure costs $M+1$. After having cleared the center we remove all blocks from edges to leaves while retaining those to connectors, which are still contaminated. The number of remaining blocks is at most $n^2N+N+3d$. Consider a path between the clear center of C_1 and another C_i. This path is not a recontamination path because the edge to C_1 is blocked. An additional cost of 2 is incurred to move this block to the edge ending on the center of C_i. Doing this for all connectors of C_1 costs at most $n^2N + N + 3d + 2$. Hence with cost $M + 1$ we can clear C_1, and leave at most $n^2N + N + 3d$ blocks in edges in other C_is, effectively reducing the cost needed to clear them.

Clearing C_2 now has cost no higher than $(M-nN)+1$, since it is equivalent to clearing a connected G_{M-nN} as nN edges to C_2's center have been cleared after clearing C_1. Additionally we still have the cost of blocking edges from the first step. So the total maximum cost is $(M - nN) + 1 + n^2N + N + 3d$. After clearing C_2 the total number of blocks that have to remain is $2 \cdot (n^2N + N + 3d) - nN$, since we do not have to block the nN edges between C_1 and C_2 anymore, but all those between C_2 and all contaminated C_is. Generalizing this formula, the cost of each step $2 \leq i \leq n/2$ is:

$$[M - (i-1)nN] + 1 + (i-1)(n^2N + N + 3d - (i-2)nN) \tag{3.3}$$

which gives us at the worst step $n/2$:

$$M + 1 + (\frac{n}{2} - 1)(\frac{n}{2} + 1)nN + (\frac{n}{2} - 1)(N + 3d) < P$$

For C_A we need at most:

$$M + 1 + \left(\frac{n}{2}\right)^2 nN + 3K' \leq P$$

For C_i with $n/2 + 1 \leq i \leq n$ an upper bound analogue to formula 3.3 applies. Hence there exists a strategy for H of cost at most P. By definition this means $ag(H) \leq P$, so the answer to the instance H, P is positive as well.

For the converse suppose that $ag(H) \leq P$. This means there exists a strategy for H of cost not higher than P. By Lemma 1 while clearing H using the optimal strategy there has to be a step at which $n/2 + 1$ centers of the C_is are cleared and $n/2$ are not. Consider the step of clearing the $n/2 + 1$-th center, and let C_j be the star it belongs to. Let us assume that $C_j \neq C_A$. The least possible cost at this point is:

$$(M + 1) + \left(\frac{n}{2}\right)^2 \cdot nN + \frac{n}{2}N.$$

This bound is derived as follows: $M + 1$ is the cost to clear the center of C_j (3.1), $(\frac{n}{2})^2 nN$ is the cost to block paths between clear centers and contaminated centers different from C_j and $\frac{n}{2}N$ is the cost to block paths between clear centers and C_A. But $(M + 1) + \left(\frac{n}{2}\right)^2 \cdot nN + \frac{n}{2}N > P$, which is a contradiction since the strategy has cost not higher than P. So $C_j = C_A$. Clearing C_A costs $M + 1$ and blocking the cleared centers from the contaminated centers costs at least $\left(\frac{n}{2}\right)^2 nN$, resulting from the nN edges added in construction step 3a between the $\left(\frac{n}{2}\right)^2$ pairs of stars. The additional edges from step 3c between cleared and contaminated centers result from the original instance of MCIESS, but there can be at most $3K$ such edges between these centers since $P = (M + 1) + \left(\frac{n}{2}\right)^2 nN + 3K$. Hence there are at most K edges between vertices in the original MCIESS instance that correspond to cleared and contaminated stars. Hence, we can define $V_1 = \{v_i : C_i \text{ has clear center}\}$ and $V_2 = \{v_j : C_j \text{ has contaminated center}\}$ and get a cut between V_1 and V_2 with at most $\lfloor (3K + 2)/3 \rfloor \leq K$ edges. □

3.4 Recontamination for Optimal Strategies

This section shows that recontamination is not necessary for optimal contiguous strategies on trees. This result is essential for the construction of such strategies in polynomial time described in Section 3.6. The proof is based on the concept of *cuts* which we will introduce first.

3.4.1 Cuts

Definition 15 (Cut) *Let T be a surveillance tree. A* cut *of T is a subset of $V(T)$ whose induced subgraph is connected. We will indicate cuts with the letter γ, and the cut $\gamma = V(T)$ is called* full cut. *Γ is the set of all cuts of T.*

Cuts can be thought of as describing all cleared vertices of a tree T. Hence, it is useful to describe the cost of blocking its boundary so that recontamination does not occur.

Notation: Let G be a surveillance graph and $X \subset V(G)$. Then:

$$\delta X = \{(x, y) \in E(T) \mid x \in X \wedge y \notin X\}.$$

δX is the subset of edges connecting vertices in X to vertices not in X.

Definition 16 (Cut blocking costs) *Let γ be a cut of T. Its* cut blocking costs *is*

$$b(\gamma) = \sum_{e \in \delta V(\gamma)} w(e). \tag{3.4}$$

In colloquial terms b is the cost to prevent recontamination of γ once it is fully cleared. By definition, a cut γ can be cleared by executing a sequence of actions with cost $ag(\gamma, \nu)$, given that T is in state ν. As a shorthand we say that

we *execute* a cut γ at cost $ag(\gamma, \nu)$. Consequently, executing γ modifies the state of T. Let us formalize the notion of sequential execution of cuts and define its cost[4].

Definition 17 (Cut sequence and its cost) *Let Γ be the set of all cuts for a surveillance tree T, and let T be in state ν^1. We define a* cut sequence $\mathcal{S}$ *as a sequence of r cuts $\gamma^1, \ldots, \gamma^r$ from Γ where γ^r is a full cut. At step $l = 1, \ldots, r$ cut γ^l is executed modifying the state ν^l to ν^{l+1}. The cost of $\mathcal{S}$ at step l is $c^l = ag(\gamma^l, \nu^l)$, and the cost of $\mathcal{S}$ is:*

$$c(\mathcal{S}) = \max_{1 \leq l \leq r} (c^l).$$

If all cuts in a cut sequence $\mathcal{S}$ are executed, T is eventually cleared because the last cut is a full cut by definition. It is immediate to see that such a sequential execution of cuts leads to a strategy for T, hence the use of the letter $\mathcal{S}$ for both strategies and cut sequences. The two different terms are introduced because they focus on different perspectives. A strategy is a sequence of actions and hence specifies exactly which sweep and block actions are taken at every step. In contrast, a step in a cut sequence only describes which vertices have to be in clear state, namely those belonging to the cut. How vertices of a cut are cleared depends on the strategy that executes it and is not specified by the cut. For example $\gamma^1 = V(T)$ is the simplest cut sequence, but executing γ^1 involves finding a strategy for all of T at once. On the other hand, cut sequences that add at most one vertex from one cut to the next can be immediately converted into a strategy.

[4]In the sequel, when talking about multiple cuts we will use the term *sequence* when the order matters, as opposed to sets.

3.4.2 Recontamination does not help

Results presented in this subsection are similar to the work by Bienstock and Seymour [BS91] for edge-search, and their definitions are herein adapted for Graph-Clear. In particular they introduced the concept of crusades that we borrow and call simple cut sequence. An analogous crusade-based construction was also used in [BFF02]. Although the basic ideas are similar, many mathematical technicalities are different, and details are therefore fully worked out in this dissertation.

Definition 18 (Simple cut sequence) *Let T be a surveillance tree. A* simple cut sequence *in T is a cut sequence $\mathcal{S} = \gamma^1, \ldots, \gamma^r$ such that $|\gamma^1| = 1$ and $|\gamma^i \setminus \gamma^{i-1}| \leq 1$ for all $2 \leq i \leq r$. For simple cut sequences, with a slight abuse of notation we write v_i for $\gamma^i \setminus \gamma^{i-1} \neq \emptyset$. If γ^i is a cut in a simple cut sequence, its* frontier $f(\gamma^i)$ *is defined as follows:*

$$\begin{cases} s(v_1) & \text{if } i = 1 \\ b(\gamma^i) + s(v_i) - \sum_{e \in \delta v_i \cap \delta \gamma^i} w(e) & \text{if } i > 1, \gamma^i \setminus \gamma^{i-1} \neq \emptyset \\ b(\gamma^i) & \text{if } i > 1, \gamma^i \setminus \gamma^{i-1} = \emptyset \end{cases} \tag{3.5}$$

The frontier *of a simple cut sequence $\mathcal{S}$ is:*

$$f(\mathcal{S}) = \max_{1 \leq i \leq r} \left\{ f(\gamma^i) \right\}. \tag{3.6}$$

The definition of simple cut sequences allows steps where $|\gamma^i \setminus \gamma^{i-1}| = 0$. This situation may arise in the uninteresting case $\gamma^i = \gamma^{i-1}$, or when $|\gamma^{i-1} \setminus \gamma^i| > 0$. This latter case corresponds to recontamination of all vertices in $\gamma^{i-1} \setminus \gamma^i$. Let us now define progressiveness for cut sequences.

Definition 19 (Progressive cut sequence) *A cut sequence is a* progressive cut sequence *if* $\gamma^1 \subseteq \gamma^2 \subseteq \ldots \subseteq \gamma^r$.

Note that progressiveness of a cut sequence is conceptually different from progressiveness of strategies. A progressive cut sequence can be executed by a strategy that is not progressive (consider for example the progressive cut sequence $\gamma^1 = V(T)$). Given a surveillance tree T we will now show that an optimal contiguous strategy implies the existence of an optimal contiguous strategy that is also progressive. This is done in three steps by first considering simple cut sequences, then cut sequences that are both simple and progressive and finally connecting these to existence of an optimal contiguous strategies that is progressive. These three steps are formalized with the following claims.

Lemma 3 *Let T be a surveillance tree, and let $\mathcal{S}_c$ be an optimal contiguous strategy for T of cost $ag(\mathcal{S}_c) \leq k$. Then there exists a simple cut sequence $\mathcal{S}$ for T such that $f(\mathcal{S}) \leq k$.*

Proof: Let $\mathcal{S}_c = \{a_1, a_2, \ldots, a_r\}$, and let $\gamma^1, \ldots, \gamma^r$ be the subsets of vertices cleared by $\mathcal{S}_c$ during the execution of its r steps. By lemma 1 we can assume that at most one vertex is cleared at each step of $\mathcal{S}_c$, and by hypothesis $\mathcal{S}_c$ is contiguous. Therefore $\mathcal{S} = \gamma^1, \ldots, \gamma^r$ is a simple cut sequence in T. Compare now 3.2 with 3.6, and 3.1 with 3.5. By substitution one can verify that $c(a_i) = f(\gamma^i)$ $(1 \leq i \leq r)$, and then $f(\mathcal{S}) \leq k$. □

The following lemma is needed in order to prove theorem 2. Its simple proof can be found in [BFF02].

Lemma 4 *Let γ^1 and γ^2 be two cuts of a surveillance tree $T = (V, E, w)$. Then*

$$b(\gamma^1 \cup \gamma^2) + b(\gamma^1 \cap \gamma^2) \leq b(\gamma^1) + b(\gamma^2)$$

Next, it is possible to show that for any simple cut sequence of bounded frontier, there is a simple progressive cut sequence whose frontier is not greater.

Theorem 2 *If there exists a simple cut sequence $\mathcal{S}$ in T with $f(\mathcal{S}) \leq k$, then there exists a simple progressive cut sequence in T with frontier not greater than k.*

Proof: Out of all simple cut sequences with frontier not greater than k choose $\mathcal{S} = \gamma^1, \ldots, \gamma^r$ satisfying the following properties:

1. $\sum_j f(\gamma^j)$ is minimal
2. $\sum_j |\gamma^j|$ is minimal, subject to the previous constraint.

We will now show that such $\mathcal{S}$ is a progressive simple cut sequence. This means:

a) $|\gamma^i \setminus \gamma^{i-1}| = 1$ for all $2 \leq i \leq r$.

b) $\gamma^1 \subseteq \gamma^2 \subseteq \ldots \subseteq \gamma^r$.

It is immediate to show that the property a) holds for $\mathcal{S}$. In fact, it cannot be the case that $|\gamma^i \setminus \gamma^{i-1}| = 0$, because otherwise the simple cut sequence obtained from $\mathcal{S}$ by excluding γ^i would invalidate property 1. Therefore $|\gamma^i \setminus \gamma^{i-1}| = 1$.

We now show that property b) holds for $\mathcal{S}$ as well. First, for an arbitrary index i let us consider $\gamma^* = \gamma^{i-1} \cup \gamma^i$. If $f(\gamma^*) < f(\gamma^i)$, then $\mathcal{S}^* = \gamma^1, \ldots, \gamma^{i-1}, \gamma^*, \gamma^{i+1}, \ldots, \gamma^r$ would be a simple cut sequence violating property 1. Therefore

$$f(\gamma^*) \geq f(\gamma^i).$$

With some work we can now derive a similar relationship involving $b(\gamma^*)$ and $b(\gamma^i)$. Since property a) holds, $v_i = \gamma^i \setminus \gamma^{i-1}$ is always well defined, i.e. $v_i \neq \emptyset$.

Let $v^* = \gamma^* \setminus \gamma^{i-1}$, and rewrite the previous inequality using 3.5 explicitly:

$$b(\gamma^*) + s(v^*) - \sum_{e \in \delta v^* \cap \delta \gamma^*} w(e) \geq$$
$$b(\gamma) + s(v_i) - \sum_{e \in \delta v_i \cap \delta \gamma^i} w(e).$$

By construction $v^* = v_i$, so the inequality simplifies to:

$$b(\gamma^*) - \sum_{e \in \delta v^* \cap \delta \gamma^*} w(e) \geq b(\gamma^i) - \sum_{e \in \delta v_i \cap \delta \gamma^i} w(e). \tag{3.7}$$

In order to further simplify the expression let us observe that there is exactly one edge between γ^{i-1} and v_i. If this was not the case there would be a cycle in the tree T. Let e' be this unique edge. By construction $\gamma^i \subseteq (\{v_i\} \cup \gamma^{i-1})$, and therefore e' is the only edge in $edges(v_i)$ with both extremes in γ^i. The exact same reasoning applies to γ^* and hence we get (remember $v^* = v_i$):

$$\sum_{e \in \delta v^* \cap \delta \gamma^*} w(e) \quad = \quad \sum_{e \in \delta v_i \cap \delta \gamma^i} w(e).$$

Inequality 3.7 then reduces to $b(\gamma^*) \geq b(\gamma^i)$, i.e. what we wanted. Let us turn our attention to $v_{i-1} = \gamma^{i-1} \setminus \gamma^{i-2}$. If $v_{i-1} \notin \gamma^i$, then $\gamma^1, \ldots, \gamma^{i-2}, \gamma^i, \ldots, \gamma^r$ is a simple cut sequence, violating property 1. Therefore $v_{i-1} \in \gamma^i$. Now consider the set $\gamma^{**} = \gamma^{i-1} \cap \gamma^i$. Since v_{i-1} belongs to both γ^{i-1} and γ^i, then $\gamma^{**} \neq \emptyset$, and γ^{**} is connected. Then, using inequality 3.7 while applying lemma 4 we can then conclude that $b(\gamma^{**}) \leq b(\gamma^{i-1})$. Now consider the cut sequence:

$$\mathcal{S}^{**} = \gamma^1, \ldots, \gamma^{i-2}, \gamma^{**}, \gamma^i, \ldots, \gamma^r.$$

$\mathcal{S}^{**}$ is a simple cut sequence. Moreover, it is easy to show that $f(\gamma^{**}) \leq f(\gamma^{i-1})$. Start with:

$$\begin{aligned} f(\gamma^{**}) &= b(\gamma^{**}) + s(v_{i-1}) - \sum_{e \in \delta v_{i-1} \cap \delta \gamma^{**}} w(e) \\ &\leq b(\gamma^{i-1}) + s(v_{i-1}) - \sum_{e \in \delta v_{i-1} \cap \delta \gamma^{**}} w(e). \end{aligned}$$

By simple set relations it follows that $\delta v_{i-1} \cap \delta\gamma^{i-1} \subseteq \delta v_{i-1} \cap \delta\gamma^{**}$ and then we get:

$$f(\gamma^{**}) \leq b(\gamma^{i-1}) + s(v_{i-1}) - \sum_{e \in \delta v_{i-1} \cap \delta\gamma^{i-1}} w(e).$$

The right side of this inequality is $f(\gamma^{i-1})$, then

$$f(\gamma^{**}) \leq f(\gamma^{i-1}) \leq k.$$

Therefore $\mathcal{S}^{**}$ is a simple cut sequence with frontier smaller or equal than k. Moreover, it must be that $|\gamma^{i-1} \cap \gamma^i| \geq |\gamma^{i-1}|$, otherwise we would violate property 2 with $\mathcal{S}^{**}$. But $|\gamma^{i-1} \cap \gamma^i| \geq |\gamma^{i-1}|$ implies that $\gamma^{i-1} \subseteq \gamma^i$, and then we have proven property b) as well, thus completing the proof.$\square$

Finally, by connecting simple progressive cut sequences to strategies we can show the main result of this section.

Theorem 3 *Let T be a surveillance tree, and let $\mathcal{S}_c$ be an optimal contiguous strategy for T of cost $ag(\mathcal{S}_c)$. Then there exists a progressive contiguous strategy of cost $ag(\mathcal{S}_c)$.*

Proof: By the previous lemma and theorem the existence of a progressive simple cut sequence with frontier not greater than $ag(\mathcal{S}_c)$ is guaranteed. Let $\mathcal{S} = \gamma^1, \ldots, \gamma^r$ be this progressive cut sequence. For $2 \leq i \leq r$ let $v_i = \gamma^i \setminus \gamma^{i-1}$, and let v_1 be the only element in γ^1. $\mathcal{S}$ leads directly to a contiguous progressive strategy by clearing the vertices v_i in order. First, consider v_1. By simple substitution $f(\gamma^1) = s(v_1)$. Assume that γ^i is cleared with cost $f(\gamma^i)$. At the end of the step a cost $b(\gamma^i)$ is required to avoid recontamination of γ^i. Adding v_{i+1} to γ^i has cost $s(v_{i+1}) - \sum_{e \in \delta v_{i+1} \cap \delta\gamma^i} w(e)$ which leads to cost $b(\gamma^i) + s(v_{i+1}) - \sum_{e \in \delta v_{i+1} \cap \delta\gamma^i} w(e) = b(\gamma^{i+1}) - \sum_{e \in \delta v_{i+1} \cap \delta\gamma^{i+1}} w(e) = f(\gamma^{i+1})$. Therefore a progressive contiguous clearing strategy of cost not greater than

$ag(\mathcal{S}_c)$ exists. Since we started assuming $\mathcal{S}_c$ is optimal, then so is $ag(\mathcal{S}_c)$ which concludes the proof. □

The main message of this section is that it is possible to construct optimal contiguous strategies on trees even when imposing that no recontamination should occur. It should be noted that the same theorem is desirable to be proven for graphs and the only use the fact that T is a tree once. Yet the proof is expected to be slightly more complicated. Such a proof for graphs, however, which also considers non-contiguous strategies would turns the NP-hardness proof of Section 3.3 into an NP-completeness proof, since strategies without recontamination are in NP. For all practical purposes the result on trees suffices since the algorithm in the next section is restricted to trees.

3.5 Label-Based Strategies on Trees

The result presented in Section 3.3 leaves little hope of finding optimal strategies for all instances of Graph-Clear with polynomial time complexity. This stimulates research to study more constrained versions of the problem. In particular we will show that if one restricts the attention to contiguous strategies on trees rather than graphs, then optimal solutions can be found with time complexity polynomial in the number of vertices. Alternatively, one can seek for approximated solutions for graphs, along the spirit of the algorithms presented in [Vaz01]. Approximate algorithms for Graph-Clear are an interesting direction for future research, but will not pursued any further in this dissertation.

From now onwards, for this section, we assume to operate on trees (i.e. connected acyclic graphs), and will therefore use the letter T rather than G. The problem of converting a surveillance graph into a tree is discussed in Section 3.10.

The algorithm presented herein was first published in [KC07c]. It does not always produce optimal strategies, thus motivating the extension presented in Section 3.6. This simpler algorithm also serves to introduce concepts needed also for the optimal algorithm presented later. In fact the new algorithm presented in Section 3.6 can be seen as a generalization of the one illustrated in this section. We will first focus on the contiguous case and then show how to compute analogue non-contiguous functions. We shall return to non-contiguous strategies in more detail in Section 3.7.

The algorithm computing contiguous strategies from [KC07c] is as follows. Numeric labels associated with edges are computed as described below, and then a strategy is produced based on the labels' values. Let $T = (V, E, w)$ be a surveillance tree. For each edge (v_x, v_y) two labels λ_{v_x} and λ_{v_y} are defined as follows:

- λ_{v_x} is the cost of clearing the contaminated subtree rooted in v_y after clearing v_x.
- λ_{v_y} is the cost of clearing the contaminated subtree rooted in v_x after clearing v_y.

Labels are computed bottom-up starting from edges incident on leaves. Due to the symmetry in the definitions of λ_{v_x} and λ_{v_y}, we discuss only the computation of λ_{v_x}. Consider an edge $e = (v_x, v_y)$. If v_y is a leaf node, i.e. $degree(v_y) = 1$, then

$$\lambda_{v_x}(e) = w(v_y) + w(e) = s(v_y)$$

since in order to clear v_y it is necessary to block the only edge it has. Next, let us assume v_y is an internal node, i.e. $degree(v_y) > 1$. Let us indicate the k neighbor

vertices different from v_x as $v_1, \ldots, v_k$, $k = degree(v_y) - 1$. Let $e_i = (v_y, v_i)$, $i = 1 \ldots k$, and let us define

$$\rho_i = \lambda_{v_y}(e_i) - w(e_i). \tag{3.8}$$

Reorder vertices so that $\rho_i \geq \rho_{i+1}$. The subtree rooted at v_y will be cleared according to the following strategy. First block all edges $e_1, \ldots, e_k$ and clear v_y. Then fully clear the subtree rooted at v_k. After clearing the subtree rooted at v_k no block on e_k is necessary anymore, and then remove it. Next, clear the contaminated subtree rooted at v_{k-1} and then remove the block from e_{k-1}. Next, clear the subtree rooted at v_{k-2}, and so on. Accordingly, in this strategy the total cost when clearing the contaminated subtree rooted at v_i is composed of all blocks at the other neighbors and the costs to clear the subtree itself, represented by the label $\lambda_{v_y}(e_i)$. This becomes:

$$c(v_i) = \lambda_{v_y}(e_i) + \sum_{l=1}^{i-1} w(e_l). \tag{3.9}$$

The value for $\lambda_{v_x}(e)$ is then computed as follows:

$$\lambda_{v_x}(e) = \max\{s(v_y), \max_{i=1,\ldots,k}\{c(v_i)\}\}. \tag{3.10}$$

Fig. 3.7 illustrates this approach graphically. Having ordered all neighboring vertices so that $\rho_i \geq \rho_{i+1}$ ensures that λ_{v_x} is minimized[5]. Once all labels are computed, a strategy $\mathcal{S}_v$ that starts clearing the tree T from a vertex v is defined as follows. Let $v_1 \ldots v_k$ be all k vertices neighbors of v. First, block all edges to v and clear v. Then, recursively clear the contaminated subtree rooted at v_i, with

[5]To show this assume there was an optimal ordering s.t. $\rho_i < \rho_{i+1}$ and show that you can then swap v_i and v_{i+1}.

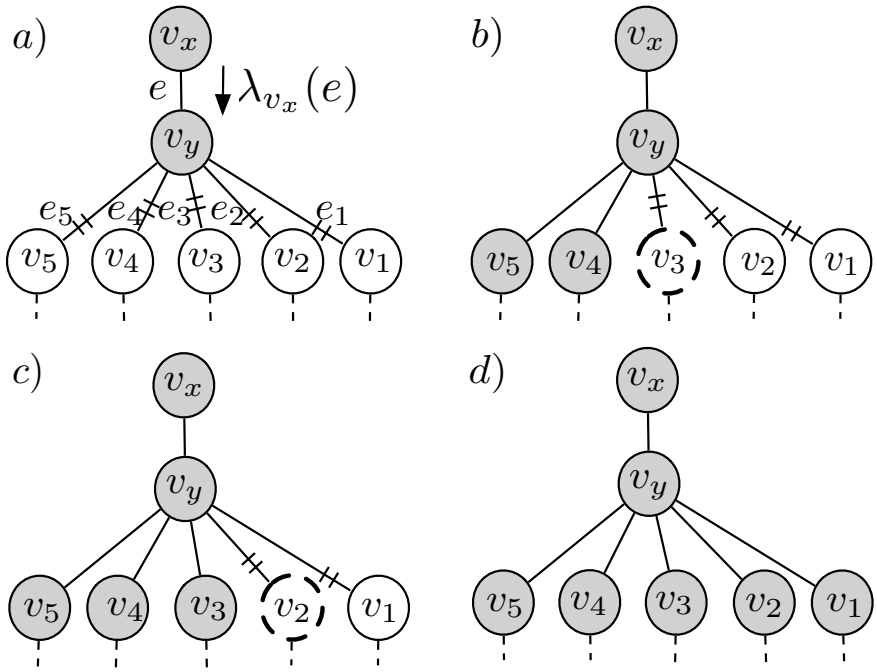


Figure 3.7: A contiguous strategy on a tree is executed based on the labels on edges. Blocked edges are crossed through twice, cleared vertices are gray. A vertex with dashed lines attached represents an entire subtree rooted at that vertex. A subtree being cleared is marked with the corresponding root vertex drawn in thick dashed lines. The label associated to this procedure is shown in a) with the direction of the robots marked by an arrow.

i from k to 1, using strategy $\mathcal{S}_{v_i}$ while blocking all edges $e_1, \ldots, e_{i-1}$. The cost of $\mathcal{S}_v$ is the following:

$$ag(\mathcal{S}_v) = \max\left\{s(v), \max_{i=1,\ldots,k}\{c(v_i)\}\right\}. \tag{3.11}$$

Once all labels have been computed it is possible to find the vertex v for which the quantity defined in Eq. 3.11 is minimized. Such vertex is the starting point to clear the tree. Given a surveillance tree, labels λ_v can be computed in

$O(n \log d)$ where n is the number of vertices and d the maximum degree across all vertices. However, it is possible to produce specific instances of Graph-Clear where the depicted algorithm does not yield an optimal contiguous strategy. This limitation motivates the formalism and ideas presented in Section 3.6.

In Algorithm 1 details of the above procedure are given. For the complexity analysis let us start considering the inner of the while loop. Its most costly step is the sorting in line 10. Let us define $d = \max_{v \in V} degree(v)$ and $d_i = degree(v_i)$. The complexity of the while loop body can be bound by $O(d \log d)$. Each vertex v_i is added to the queue at most $degree(v_i)$ times and at least $degree(v_i) - 1$. Considering that $\sum_{i=1}^{n} d_i = 2m$ and that $m = n - 1$ because we are dealing with a tree, it turns out that the the outer loop is executed at most $2(n-1)$ times. Combining these results we get that an overall complexity of $O(nd \log d)$ for a naive implementation[6]. But, clearly a smart implementation would sort $degree(v) - 1$ neighbors once for the first incoming label and reuse it. The computation for every other incoming label at the same vertex differs by only one vertex in the set of neighbors and we can hence compute subsequent sorted neighbor lists in $O(\log d)$ instead of $O(d \log d)$. Similarly, computing the maximum would at first sight take $O(d)$, but using results from Section in 3.7.1 on so called batches which describe the insertion and removal of vertices from the set of neighbors we can find the maximum in $O(1)$. Putting all this together gives $\sum_{i=1}^{n} degree(v_i) \log d \leq 2m \cdot \log d$ as a bound for the initial sorting and $\sum_{i=1}^{n} degree(v_i) \cdot \log d \leq 2m \cdot \log d$ for the computation of all labels other than the first incoming label. This leads to an overall complexity of $O(n \log d)$.

We can also produce a simple bound for the labels that are used to finally compute $ag(\mathcal{S}_v)$.

[6]The complexity of the for loop starting at line 18 is clearly dominated by the previous one, so it does not contribute to the asymptotic bound

1: Set all labels to 0 and initialize empty queue O

2: $O.enqueue(leaves(T))$

3: **while not** $O.empty()$ **do**

4: $\quad v_y \leftarrow O.dequeue()$

5: $\quad$ **if** $degree(v_y) = 1$ **then**

6: $\quad\quad v_x \leftarrow neighbors(v_y)$

7: $\quad\quad \lambda_{v_x}([v_x, v_y]) \leftarrow w(v_y) + w([v_x, v_y])$

8: $\quad$ **else if** $\exists v_x$ s.t. $\lambda_{v_x}([v_y, v_x]) = 0$ **then**

9: $\quad\quad k \leftarrow degree(v_y) - 1$

10: $\quad\quad$ Let $v_1, \ldots, v_k, v_x$ be neighbors s.t. $\lambda_{v_y}([v_y, v_i]) > 0$, $\lambda_{v_x}([v_y, v_x]) = 0$ and v_i ordered by ρ_i, $i = 1, \ldots k$.

11: $\quad\quad \lambda_{v_x}([v_x, v_y]) \leftarrow \max\{s(v_y), \max_{i=1,\ldots,k}\{c(v_i)\}\}$

12: $\quad a \leftarrow$ number of neighbors of v_x s.t. $\lambda_{v_x}([v_x, v]) > 0$

13: $\quad$ **if** $a = degree(v_x) - 1$ **then**

14: $\quad\quad O.enqueue(v_x)$

15: $\quad$ **else if** $a = degree(v_x)$ **then**

16: $\quad\quad$ **for all** $v \in neighbors(v_x)$ s.t. $\lambda_v([v, v_x]) = 0$ **do**

17: $\quad\quad\quad O.enqueue(v_x)$

18: **for all** $v \in Vertices(T)$ **do**

19: $\quad$ Let V be all k neighbors of v

20: $\quad ag(v) \leftarrow \max\{s(v), \max_{i=1,\ldots,k}\{c(v_i)\}\}$

21: **return** $\min_{v \in Vertices(T)}(ag(v))$

Algorithm 1: *Contiguous_strategy*(T)

Theorem 4 *Let* $s_{max} := \max_{\forall v \in Vertices(T)}\{s(v)\} > 2$. *Let* d *be the length of the longest simple path*[7] *in the tree* T *and* $d^* = \lceil d/2 \rceil$. *The label of any edge is bound as follows:*

$$\max_{v_x \in Vertices(T), e=(v_x,v_y)} \{\lambda_{v_x}(e)\} \leq s_{max} + d^* \cdot (s_{max} - 3) \tag{3.12}$$

Proof: We start by identifying the worst case for a label $\lambda_{v_x}((v_x, v_y))$ first at a leaf, then at the first non-leaf vertex and further up the tree. For edges to a leaf v_y we trivially have $\lambda_{v_x}(e) = s(v_y) \leq s_{max}$ and the worst-case is hence $\lambda_{v_x}(e) = s_{max}$ for all leaves. So let us assume v_y is not a leaf. Consider the subtrees T_i generated for each v_i by removing all edges of v_y except for e_i and rooting T_i in v_y (see fig. 3.10 in Section 3.6.1 for an illustration). For the first induction step let all subtrees T_i have a maximum depth of 1, i.e. all v_i are leaves. Recall equation 3.10, particularly the part $\max_{i=1,\ldots,k}\{c(v_i)\}$. For every $i = 1, \ldots, k$ we have $c(v_i) = \lambda_{v_y}(e_i) + \sum_{l=1}^{i-1} w(e_l) \leq s_{max} + s_{max} - 3$ since $\sum_{l=1}^{k-1} w(e_l) \leq s_{max} - 3$ because $w(v_y) \geq 1$, $w(e) \geq 1$ and $w(e_k) \geq 1$. Clearly, the worst case occurs if some vertex v_j is being cleared with $c(v_j) = 2 \cdot s_{max} - 3$, i.e. the worst case occurs exactly when $\lambda_{v_v}(e_j) = s_{max}$ and $\sum_{l=1}^{j-1} w(e_l) = s_{max} - 3$. This implies that $j = k, w(v_y) = 1, w(e) = 1, w(e_k) = 1$. Recall that $\rho_k = \lambda_{v_v}(e_k) - w(e_k) \leq \rho_i$. Hence all $w(e_i) = 1$ which implies that $k = s_{max} - 2$. Using the same argument and continuing the induction with increasing allowed depth of all subtrees T_i until it is limited to depth d^* leads to:

$$\max_{v_x \in Vertices(T), e=(v_x,v_y)} \{\lambda_{v_x}(e)\} \leq s_{max} + d^* \cdot (s_{max} - 3) \tag{3.13}$$

Evidently, we cannot continue the induction further than d^* since no two subtrees T_i and T_j with $j \neq i$ can have depth larger than d^* since that violates the assumption that d is the maximum length of a simple path in T. But to make

[7] *Path length is counted in terms of edges.*

the argument sound we have to discuss the maximum label cost for the case when one subtree has depth larger than d^*. So let us assume that one T_j for some j has a depth d_j which is larger than d^*. This implies that all T_i with $i \neq j$ have depth at most $d - d_j$ and hence worst-case labels $s_{max} + (d - d_j) \cdot (s_{max} - 3)$. Since $(d - d_j) < d_j$ and $\sum_{l=1}^{i-1} w(e_i) < s_{max}$ for all $i = \ldots 1, \ldots k$ the worst case for label $\lambda_{v_x}(e)$ is $s_{max} + d_j \cdot (s_{max} - 3) = \lambda_{v_y}(e_i)$ and is hence not getting worse. Therefore, eq. 3.13 holds and $s_{max} + d^* \cdot (s_{max} - 3)$ is the worst case label on any edge. □

Fig. 3.8 shows how the construction of the worst-case example works for given parameters. The bound of the theorem is obviously tight, i.e. it can be achieved by examples for any set of parameters. Yet, the construction is very strict and in practice we expect to be far below the bound in particular for large trees. To show this numerically we ran the algorithm on randomly generated weighted trees. Trees with 20, 40, 60, 80 and 100 vertices, random edges, a random weight for a vertex between 1 and 12 and a random weight for edges between 1 and 6 were generated. For each number of vertices a forest of 1000 trees was created. The average values for the labels, d^* and s_{max} are presented in table 3.5. Figure 3.9 compares the upper bound computed from d^* and s_{max} with the average maximum value of all labels.

We can modify the presented algorithm to compute non-contiguous strategies. This modification is rather straight-forward. The ordering of the vertices v_i remains the same, but the order of clearing them changes. We first clear the subtree at v_1, then proceed until v_k and clear v_y last. Historically the non-contiguous label algorithm has been introduced first, hence the indices of the vertices v_i correspond to the order in which they are cleared for the non-contiguous variant. The equations, however, remain the same. This is simple to notice by checking

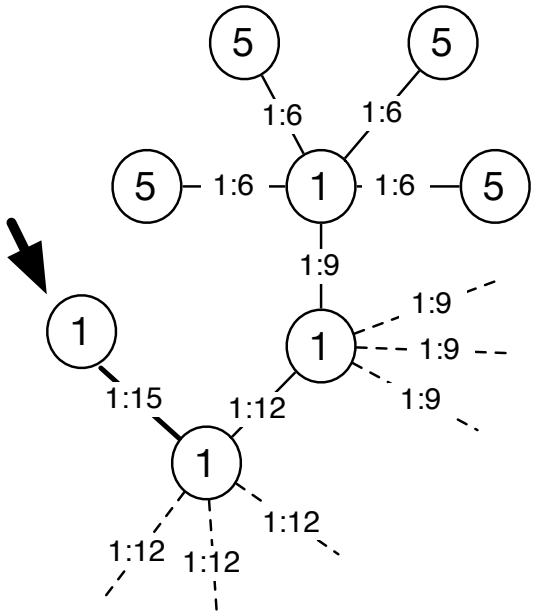


Figure 3.8: This is the worst case example for $d = 6, s_{max} = 6$ leading to a worst case label cost of 15. Blocking weight w is on the left and label on the right of the colon for every edge. Each vertex has its weight in its center. Only labels for the direction towards the leaves from the vertex marked with a black arrow are shown.

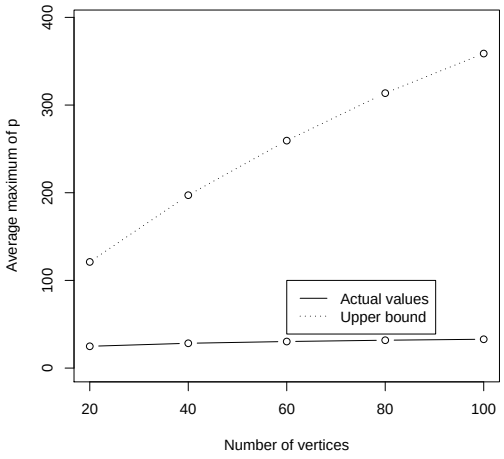


Figure 3.9: A comparison of the average upper bound across 1000 weighted trees and actual maximum label values for varying number of vertices.

n vertices	$\max(\lambda_{v_x}(e))$	d*	s_{max}
20	24.865	5.325	25.176
40	28.300	7.784	27.949
60	30.299	9.638	29.607
80	31.781	11.077	31.039
100	32.885	12.306	31.909

Table 3.1: Results of the experiments. Values are averaged across 1000 random trees.

the edges that have to be blocked after each clearing step. After v_1 is cleared only the edge between v_y and v_1 has to be blocked when starting to clear v_2. This is identical to the step in the contiguous variant, when v_2 is being cleared only the edge to v_1 has to be blocked. The key is to notice that for the non-contiguous variant v_y remains contaminated until all its subtrees are clear while for the contiguous variant it is cleared first. In Section 3.7 we shall discuss how to combine these two sequences for clearing the subtrees into an improved algorithm and explore what happens if we allow the clearing of v_y somewhere in between the clearing of the subtrees at the v_i's.

Finally, even though we will introduce an improved and optimal algorithm for contiguous strategies in Section 3.6 the algorithm presented here has one major advantage, which is also the reason why it is not optimal. The clearing procedure clears the entire subtree before it considers neighboring subtrees. This depth-first search behavior ensures that the robot team only enters each subtree once which puts a bound on the number of edges that are traversed. For the optimal algorithm subtrees will have to be entered multiple times and one edge can be traversed many more times. In practice, it usually matters how many edges the

robot team traverses and in these cases the presented algorithm remains useful.

3.6 Optimal Contiguous Strategies on Trees: a Polynomial Algorithm

Given that we established the existence of at least one optimal contiguous strategy that is progressive, our goal is now to devise an algorithm that computes one efficiently. To do so, we first introduce a final class of cut sequences, called *full cut sequences*, and identify some notable properties. Based on these findings, we next develop an $O(n^2)$ algorithm to compute an optimal contiguous strategy.

3.6.1 Full cut sequences

Let T be a fully contaminated surveillance tree and let v_y be the first vertex cleared by an optimal contiguous progressive strategy. Since v_y will never be recontaminated we can consider the subtrees at each neighbor $v_1, \ldots, v_k$ of v_y separately. More precisely, for each $i = 1, \ldots, k$ we will write T_i for the subtree of T rooted at v_y with all edges of v_y removed except the edge to v_i (see Fig. 3.10). Each of the T_is can be thought of as a surveillance tree with the same weights induced by the w function defined on T and its own state. Given that a cut sequence alters the state of T, we will indicate with ν_i^l the state of T_i before γ^l is executed on T.

We want to construct an optimal cut sequence $\mathcal{S}_{v_y}$ which starts clearing v_y first, and then removes all contamination from T. Its first cut is $\gamma^1 = \{v_y\}$, which we can execute with cost $ag(\gamma^1, \nu^1)$ leading to a new state ν^2. Note that, accordingly to the notation introduced above, ν^2 induces a state ν_i^2 for each T_i. The goal is now to find optimal cut sequences for each T_i starting at state ν_i^2 that

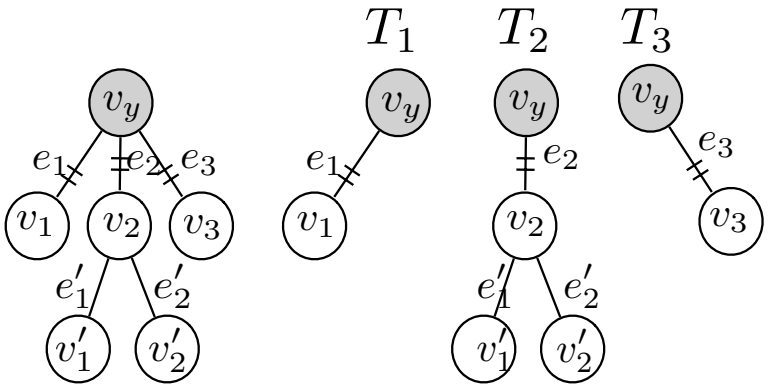


Figure 3.10: Given v_y we define subtrees T_i as seen in the figure.

we can use to continue $\mathcal{S}_{v_y}$ and turn it into an optimal cut sequence for the whole tree T. The building blocks for these sequences are identified by the following definition.

Definition 20 (Full cut sequence) *Let T be a surveillance tree in state $\nu^1 = \{\mathcal{C}, \ldots, \mathcal{C}\}$, $v_y \in V[T]$, and Γ_{v_y} the set of all cuts containing v_y. A* full cut sequence for v_y, *indicated as $\bar{\mathcal{S}}$, is built as follows. Sort all cuts of Γ_{v_y} s.t. $ag(\gamma, \nu^1)$ is increasing. All cuts with identical $ag(\gamma, \nu^1)$ are additionally sorted with $b(\gamma)$ increasing. First, add $\gamma^1 = \{v_y\}$ to $\bar{\mathcal{S}}$. Next, add the first cut of Γ_{v_y} with $b(\gamma) < b(\gamma^1)$. Then, add every next cut γ from Γ_{v_y} with the next larger $ag(\gamma, \nu^2)$ such that $b(\gamma)$ is smaller than for the previously added cut. Repeat this process until the full cut is added to $\bar{\mathcal{S}}$.*

The reader should observe that since by definition the full cut has $b(\gamma) = 0$ this definition is well posed for every T and terminates after a finite number of steps. Note that we did not require that a full cut sequence be either progressive or simple. A full cut sequence $\bar{\mathcal{S}}$ for v_y has a useful property that $ag(\gamma^l, \nu^l) = ag(\gamma^l, \nu^2)$ for all $2 \leq l \leq r$. This is formalized by the following more general lemma.

Lemma 5 *Let $\mathcal{S} = \gamma^1, \ldots, \gamma^r$ be a cut sequence such that $ag(\gamma^l, \nu^l)$ is monotonically increasing for all indexes l within the range $b \leq l \leq r$.*[8] *Then for all $b \leq l \leq r$:*

$$ag(\gamma^l, \nu^b) = ag(\gamma^l, \nu^l).$$

Proof of Lemma 5. The claim is true for $l = b$ by substitution. Let us now assume that $ag(\gamma^l, \nu^l) = ag(\gamma^l, \nu^b)$ for a certain value of l in the range $b \leq l < r$ and prove that $ag(\gamma^{l+1}, \nu^{l+1}) = ag(\gamma^{l+1}, \nu^b)$. By definition $ag(\gamma^{l+1}, \nu^b)$ is the cost of the optimal strategy removing all contamination from γ^{l+1} in state ν^b. Consider the following strategy. Start with T in state ν^b and execute γ^l. This will change the state of the tree to ν^{l+1}. Then execute γ^{l+1} starting from the current state ν^{l+1}. The cost of this strategy is

$$ag(\gamma^{l+1}, \nu^b) = \max((ag(\gamma^l, \nu^b), ag(\gamma^{l+1}, \nu^{l+1})).$$

This maximum cannot be $ag(\gamma^l, \nu^b)$ since by assumption $ag(\gamma^l, \nu^b) = ag(\gamma^l, \nu^l) < ag(\gamma^{l+1}, \nu^{l+1})$. Therefore $ag(\gamma^{l+1}, \nu^b) = ag(\gamma^{l+1}, \nu^{l+1})$. □

Next, for each T_i let $\bar{\mathcal{S}}_i$ be a full cut sequence for v_y. It is worth observing that by definition $v_y \in T_i$ for each T_i, so a full cut sequence for v_y in T_i is well defined. For each i, let us indicate cuts in $\bar{\mathcal{S}}_i$ as γ_i^j. To each cut $\gamma_i^j \in \bar{\mathcal{S}}_i$ with $j \geq 2$, i.e. excluding the first cut $\{v_y\}$, associate a value ρ_i^j defined as follows:

$$\rho_i^j = ag(\gamma_i^j, \nu_i^j) - b(\gamma_i^{j-1}) \text{ for } j \geq 2 \tag{3.14}$$

The reader should notice the similarity between 3.14 and 3.8, and in fact the

[8] *This is the case for full cut sequences picking $b = 2$. For $b = 1$ the lemma does not necessarily hold for full cut sequences since the cost for executing $\gamma^1 = \{v_y\}$ is allowed to be greater than subsequent costs.*

latter generalizes the former. Let

$$\bar{\Gamma} = \bigcup_{i=1,\ldots,k} \bar{\mathcal{S}}_i \setminus \{v_y\}$$

and order them with ρ increasing. This ordering is consistent with the ordering of each subsequence by construction of full cut sequences. Notice that $\gamma_i^1 = \{v_y\}$ for all i and hence for each the first cut is removed, which is also the cut for which ρ_i^j is not defined and for which the prerequisites for lemma 5 do not hold. Ties between cuts coming from different T_i can be arbitrarily resolved. Write the ordered sequence $\bar{\Gamma}$ as $\{\bar{\gamma}^2, \ldots, \bar{\gamma}^r\}$. Finally, create a cut sequence $\mathcal{S}_{v_y} = \gamma^1, \ldots, \gamma^r$ for T from $\bar{\Gamma}$ as follows:

$$\gamma^l = \begin{cases} \{v_y\} & l = 1 \\ \\ (\gamma^{l-1} \setminus T_i) \cup \bar{\gamma}^l & l = 2, \ldots, r \ \wedge \ \bar{\gamma}^l \subseteq T_i \end{cases} \tag{3.15}$$

In colloquial terms, at each step l we execute the cut $\bar{\gamma}^l$ in a subtree T_i, while maintaining clear all vertices from all other subtrees from the previous step $l-1$. Note that it is necessary to remove T_i from γ^{l-1} to keep only cleared vertices in other subtrees and have cleared vertices in T_i to be exactly $\bar{\gamma}^l$ (second case in the definition above). This is due to the fact that we have not ruled out recontamination yet. We can now introduce the main results of this subsection, namely that $\mathcal{S}_{v_y}$ is optimal.

Theorem 5 *$\mathcal{S}_{v_y}$ is an optimal cut sequence for T.*

Proof: We started assuming that v_y is the first vertex cleared by an optimal progressive contiguous strategy for T, so starting with $\gamma^1 = \{v_y\}$ does not compromise the possibility to get an optimal strategy. Let us consider $l \geq 2$. By construction, every cut γ^l has an associated cut $\bar{\gamma}^l$ (see Eq. 3.15). Such $\bar{\gamma}^l$ was

constructed from a certain $\gamma_i^j \in \bar{\mathcal{S}}_i$. We can therefore associate each cut γ^l with $l \geq 2$ with a couple of indexes i and j such that γ^l originated from γ_i^j.

Let us now describe the costs of $\mathcal{S}_{v_y}$ and relate it to the costs in the full cut sequences for each T_i. $b(\gamma^l)$ is the blocking cost in T after executing γ^l. The part of this blocking cost in T_i is given by $b_i^l = b(\gamma^l \cap T_i)$ and is equal to $b(\gamma_i^j)$ by construction. The cost of executing γ^l is given by $c^l = ag(\gamma^l, \nu^l)$. We can relate c^l to the costs inside each subtree T_i with the following relationship:

$$c^l = b^{l-1} - b_i^{l-1} + ag(\gamma_i^j, \nu_i^j). \tag{3.16}$$

Notice that ν_i^j of the cut sequence $\bar{\mathcal{S}}_i$ is identical to the state that ν^l of the cut sequence $\mathcal{S}_{v_y}$ induces on T_i. Eq. 3.16 follows simply by construction. It results from keeping $(\gamma^{l-1} \setminus T_i)$ blocked which costs $b^{l-1} - b_i^{l-1}$, and executing γ_i^j in T_i with cost $ag(\gamma_i^j, \nu_i^j)$. By lemma 5 and the observation that $b_i^{l-1} = b(\gamma_i^{j-1})$, i.e. the blocking cost from the cut in $\bar{\mathcal{S}}_i$ right before γ_i^j, we get (see Eq. 3.14):

$$c^l \;\; = \;\; b^{l-1} - b_i^{l-1} + ag(\gamma_i^j, \nu_i^2) = b^{l-1} + \rho_i^j \tag{3.17}$$

Here the significance of ρ values formerly defined becomes apparent. Notice the similarity to ordering subtrees in Section 3.5. In colloquial terms, the ordering with ρ increasing asks for the next cut that can reduce the blocking cost while not costing much to execute.

Now, let $\hat{\mathcal{S}}$ be an optimal contiguous strategy that is progressive and starts at v_y. We will adopt a similar notation as for $\mathcal{S}_{v_y}$ but adding $\hat{}$ to all terms. Due to lemma 1 we can assume that $\hat{\mathcal{S}}$ clears exactly one new vertex per step. Therefore, it can be written as a simple progressive cut sequence $\hat{\gamma}^1, \ldots, \hat{\gamma}^n$ with exactly n cuts. It follows that for every $l = 2, \ldots, n$ we have one and only one i s.t. $\hat{\gamma}^l \setminus \hat{\gamma}^{l-1} \subset T_i$. This allows us to consider, for each T_i, a cut sequence given by all non-empty $\hat{\gamma}^l \cap T_i$ for all l s.t. $\hat{\gamma}^l \setminus \hat{\gamma}^{l-1} \subset T_i$. To identify these cut sequences

restricted to each T_i we use $\hat{\gamma}_i^j = \hat{\gamma}^l \cap T_i$, again associating each step l in $\hat{\mathcal{S}}$ with an $\hat{\gamma}_i^j$. Hence, a similar analysis as above for $\mathcal{S}_{v_y}$ applies and we get:

$$\hat{c}^l \;=\; \hat{b}^{l-1} - \hat{b}_i^{l-1} + ag(\hat{\gamma}_i^j, \hat{\nu}_i^l) \tag{3.18}$$

where now $\hat{\nu}_i^l$ is simply the state of T_i induced by $\hat{\nu}^l$ and equal to $\hat{\nu}_i^j$, which is the state of T_i after execution of $\hat{\gamma}_i^1, \ldots, \hat{\gamma}_i^{j-1}$ in T_i. Now that we can describe $\mathcal{S}_{v_y}$ and $\hat{\mathcal{S}}$ let us compare the two and their costs. We will do so by replacing cuts in the cut sequences $\hat{\gamma}_i^j$ with cuts from the full cut sequences used to construct $\mathcal{S}_{v_y}$. In colloquial terms, we will show that cuts from the full cut sequences are not more costly than the optimal ones. For each i consider the cut sequence $\hat{\gamma}_i^j$ in T_i with associated clearing cost $ag(\hat{\gamma}_i^j, \hat{\nu}_i^j)$ and blocking cost $b(\hat{\gamma}_i^j)$. First, we want to remove all cuts $\hat{\gamma}_i^j$ which do not reduce b^l, i.e. have $b(\hat{\gamma}_i^j) \geq b(\hat{\gamma}_i^{j-1})$. It is evident from Eq. 3.18 that this removal does not increase b^l at any step. Hence, removal of such a cut $\hat{\gamma}_i^j$ can only lead to larger costs if it increases the cost for a subsequent cut, i.e. $ag(\hat{\gamma}_i^p, \hat{\nu}_i^p)$, for some $p > j$. But if after removal of $\hat{\gamma}_i^j$ we have $ag(\hat{\gamma}_i^p, \hat{\nu}_i^p)$ larger than before, then (through a similar argument as for the proof of lemma 5) we have $ag(\hat{\gamma}_i^p, \hat{\nu}_i^p) \leq ag(\hat{\gamma}_i^j, \hat{\nu}_i^j)$ and hence no overall larger cost. Therefore, we can remove all such cuts without increasing the overall cost, which leads to b^l becoming a strictly decreasing sequence.

With a similar argument we can modify the sequence $\hat{\gamma}_i^j$ to have $ag(\hat{\gamma}_i^j, \hat{\nu}_i^j)$ strictly increasing. Notice that if $ag(\hat{\gamma}_i^j, \hat{\nu}_i^j) \geq ag(\hat{\gamma}_i^{j+1}, \hat{\nu}_i^{j+1})$, then removal of $\hat{\gamma}_i^j$ and executing $\hat{\gamma}_i^{j+1}$ instead will not increase costs c^l, since $b(\hat{\gamma}_i^j) > b(\hat{\gamma}_i^{j+1})$ and $ag(\hat{\gamma}_i^{j+1}, \hat{\nu}_i^j) \leq ag(\hat{\gamma}_i^j, \hat{\nu}_i^j)$.

After these removals we are in a condition that satisfies the hypothesis of lemma 5 and obtain:

$$\hat{c}^l = \hat{b}^{l-1} - \hat{b}_i^{l-1} + ag(\hat{\gamma}_i^j, \hat{\nu}_i^2) = \hat{b}^{l-1} - \hat{\rho}_i^j \tag{3.19}$$

It is now clear to see (looking at ρ) that replacing every cut $\hat{\gamma}_i^j$ with a cut from the full cut sequence with equal or smaller ag and adding all remaining full cuts, ordered by ρ, leads to no increased cost. Hence we can turn $\hat{\mathcal{S}}$ into $\mathcal{S}_{v_y}$ without incurring larger cost and hence $\mathcal{S}_{v_y}$ is optimal. □

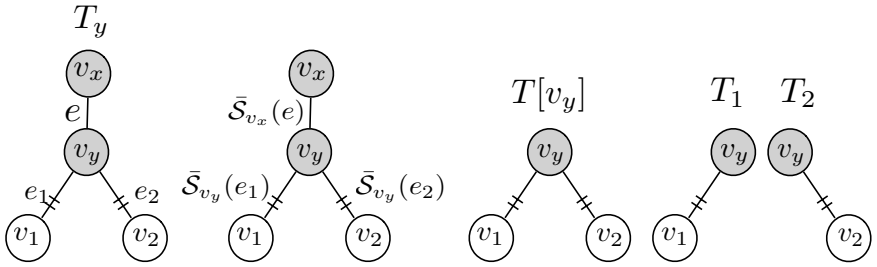
3.6.2 Constructing cut sets

Theorem 5 provides the basis for a recursive construction of optimal cut sequences starting at the leaves and associating them to edges, much like labels defined in Section 3.5. Similarly to the label-based algorithm, for each starting vertex the algorithm computes the best contiguous progressive strategy and its cost starting from that vertex. Then, looking at the costs of these these n strategies the optimal one can be retrieved. It is immediate to see that if all full cut sequences are progressive, then so is $\mathcal{S}_{v_y}$.

A brute force approach is not viable because we need to consider the set of all cuts for finding full cut sequences and this set grows exponentially in the number of vertices. However, we can construct full cut sequences more efficiently according to a bottom up paradigm. We first show how to start the recursive construction at the leaves and then show how to find full cut sequences efficiently.

We adopt the same perspective as in Section 3.5, i.e. let v_x and v_y neighbors, $e = [v_x, v_y]$, and $v_1, \ldots, v_k$ with $k = degree(v_y) - 1$ are all neighbors of v_y different from v_x (see Fig. 3.7 and Fig. 3.11). We now associate a full cut sequence $\bar{\mathcal{S}}_{v_x}(e)$ to e coming from direction v_x, similar to the label $\lambda_{v_x}(e)$. Note that $\bar{\mathcal{S}}_{v_x}(e)$ is a full cut sequence for v_x in the tree T_y given by removing all edges from v_x except e (see Fig. 3.11). If v_y is a leaf, then $k = 0$, i.e. its only neighbor is v_x. In this case it is immediate to compute the right cut sequence:

$$\bar{\mathcal{S}}_{v_x}(e) = \{v_x\}, \{v_x, v_y\}.$$



$$\bar{\mathcal{S}}_{v_y}(e_1) = \{v_y\}, \{v_y, v_1\} \qquad \bar{\mathcal{S}}_{v_y}(e_2) = \{v_y\}, \{v_y, v_2\}$$

$$\mathcal{S}_{v_y} = \{v_y\}, \{v_y, v_2\}, \{v_y, v_2, v_1\}$$

Possibilities for $\bar{\mathcal{S}}_{v_x}(e)$

1) $\bar{\mathcal{S}}_{v_x}(e) = \{v_x\}, V(T_y) = \{v_x, v_y, v_2, v_1\}$

2) $\bar{\mathcal{S}}_{v_x}(e) = \{v_x\}, \{v_x, v_y\}, V(T_y)$

3) $\bar{\mathcal{S}}_{v_x}(e) = \{v_x\}, \{v_x, v_y\}, \{v_x, v_y, v_2\}, V(T_y)$

Figure 3.11: Illustration of the cut sequences associated to edges and the subtrees involved in the construction. There are three possibilities for $\bar{\mathcal{S}}_{v_x}(e)$, depending on the costs of the cuts. E.g. if weights on the tree are s.t. executing $\{v_x, v_y\}$ costs as much as executing the full cut $V(T_y)$ right away, then the first possibility is the full cut sequence. Otherwise, if $\{v_x, v_y\}$ costs less and has smaller blocking cost than $\{v_x\}$, then the second possibility is the full cut sequence, and so on.

Otherwise, if $k > 0$ we consider $v_1, \ldots, v_k$ with edges $e_i = [v_y, v_i]$, $i = 1, \ldots, k$. Let $\bar{\mathcal{S}}_{v_y}(e_i)$ be the full cut sequence on edge e_i coming from direction v_y. By virtue of theorem 5 we can construct an optimal cut sequence $\mathcal{S}_{v_y}$, as defined by Eq. 3.15, which clears the subtree $T[v_y]$ rooted at v_y after removing e (see Fig. 3.11). If all v_i are leaves this corresponds to exactly the same local strategy that the algorithm from Section 3.5 produces. This is evident if one compares 3.8 and 3.14 and keeps in mind that cuts are sorted with ρ increasing.

We now need to find a full cut sequence for T_y, where T_y is the analogue of the trees T_i but from the perspective of v_x towards v_y instead of v_y towards

v_i (see Fig. 3.11). This full cut sequence can be associated to e and written as $\bar{\mathcal{S}}_{v_x}(e)$ to be able to continue the recursion. The first cut is trivially $\{v_x\}$. Next, we can obtain the remaining cuts from the already available $\mathcal{S}_{v_y}$ instead of looking at all possible cuts. One observation is crucial to this construction, namely that only cuts that correspond to a step in the execution of $\mathcal{S}_{v_y}$ need to be in $\bar{\mathcal{S}}_{v_x}(e)$. Clearly, one needs to add v_x to all cuts of $\mathcal{S}_{v_y}$, since $v_x \notin T[v_y]$. For the example with all $v_1, \ldots, v_k$ leaves this leads to cuts $\{v_x, v_y\}, \{v_x, v_y, v_k\}, \ldots,$ $\{v_x, v_y, v_k, \ldots, v_1\}$ (assuming that indices are ordered by ρ as in Section 3.5). We can now obtain a full cut sequence by selecting all cuts from this set that satisfy the criteria outlined in definition 20. The following argument validates this claim.

Theorem 6 *Let $\bar{\mathcal{S}}_{v_x}$ be a full cut sequence for v_x in T_y and $\mathcal{S}_{v_y} = \gamma^1, \ldots, \gamma^r$ constructed as before. If $\gamma \in \bar{\mathcal{S}}_{v_x}$ and $\gamma \neq \{v_x\}$, then $\exists l \in \{1, \ldots, r\}$ s.t. $\gamma^l \in \mathcal{S}_{v_y}$ has $ag(\gamma^l \cup \{v_x\}, \nu^1) \leq ag(\gamma, \nu^1)$ and $b(\gamma^l \cup \{v_x\}) \leq b(\gamma)$.*

Proof of Theorem 6. Let $\gamma \in \bar{\mathcal{S}}_{v_x}$. By definition $v_x \in \gamma$. If $\gamma = \{v_x, v_y\}$ the claim is trivially true by noting that $l = 1$ with $\gamma^1 = \{v_y\}$ and then $ag(\{v_y\} \cup \{v_x\}, \nu^1) = ag(\gamma, \nu^1)$.

Otherwise, γ has one or more vertices in some T_i, $i = 1, \ldots, k$. We can hence write the execution of γ as a new sequence of cuts starting at $\hat{\gamma}^1 = \{v_x\}, \hat{\gamma}^2 = \{v_x, v_y\}$ and continuing with cuts separated into T_i similar as for the proof of theorem 5. Write $\hat{\gamma}^3, \ldots, \hat{\gamma}^t = \gamma$ for these cuts. Note that this is not a cut sequence for T_y, but a only a sequence of cuts because the full cut is missing. Again, as for theorem 5, $\hat{\gamma}^3, \ldots, \hat{\gamma}^t$ induces sequences of cuts in the subtrees T_i, written $\hat{\gamma}_i^j$. The only difference to the proof for theorem 5 is that we had two steps $\hat{\gamma}^1$ and $\hat{\gamma}^2$ prior to considering the cuts in subtrees T_i, which means that v_x and v_y are both not going to be recontaminated. Therefore, we can also apply

the same replacement method as for theorem 5 and all cuts $\hat{\gamma}_i^j$ can be replaced with cuts from full cut sequences of T_i without incurring larger costs.

Now, all $\hat{\gamma}_i^j$ are cuts that also appear in the full cut sequences for T_i. The only significant difference to the proof of theorem 5 is that after this replacement there is a last cut in each sequence $\hat{\gamma}_i^j$ which is not necessarily a full cut for T_i, so the sequence is still not a cut sequence. Now, the last cut $\hat{\gamma}^t$ is associated to a last cut of some sequence $\hat{\gamma}_i^j$ in some T_i written $\hat{\gamma}_i^p$. Since $\hat{\gamma}_i^p \in \bar{\mathcal{S}}_i$ due to the replacement method we have a cut γ^l in $\mathcal{S}_{v_y}$ associated to it as well. By construction and from Eq. 3.17, $\mathcal{S}_{v_y}$ incurs no higher cost up until γ^l than the sequence $\hat{\gamma}^3, \ldots, \hat{\gamma}^t$. This is due to the fact that $\mathcal{S}_{v_y}$ is based on all cuts from the full cut sequences $\bar{\mathcal{S}}_i$ for each T_i while $\hat{\gamma}^3, \ldots, \hat{\gamma}^t$ may have some cuts omitted. From Eq. 3.17 it is clear that adding additional cuts from the full cut sequences $\bar{\mathcal{S}}_i$ can only improve the costs since those cuts with ρ lowest are executed first and after being executed can only decrease the overall blocking cost. It follows that $ag(\gamma^l \cup \{v_x\}, \nu^1) \leq ag(\gamma, \nu^1)$ and $b(\gamma^l \cup v_x) = b(\gamma^l) \leq b(\gamma)$. □

Theorem 6 implies that we can get a full cut sequence $\bar{\mathcal{S}}_{v_x}$ to associate to e by only considering the cuts arising from $\mathcal{S}_{v_y}$. Algorithm 2 shows how to use the results from theorems 5 and 6 to compute an optimal strategy. Just like we did with labels before, the algorithm recursively builds cut sequences on the edges of a surveillance tree T with two directions for each edge. To finally obtain $ag(T)$ we construct $\mathcal{S}_{v_y}$ on T for each vertex $v_y \in V(T)$, and then the vertex with the lowest cost is selected as the starting vertex. This is similar to the procedure in Section 3.5. Algorithm 2 presents this in pseudo-code and returns $ag(T)$. Once the best vertex v_y is found, translating the cut sequence $\mathcal{S}_{v_y}$ into a strategy is straightforward. In fact, by following the edges and using the associated cut sequences one can write $\mathcal{S}_{v_y}$ as a simple cut sequence which can be immediately

converted into a strategy.

The complexity of algorithm 2 is $O(n^2)$ and can be computed as follows. Throughout the analysis it is important to note that since we are dealing with a tree the number of edges is $m = n - 1$, so eventually we compute the complexity as a function of the number of vertices only. Clearly, line 11 is the most costly part in which we have to sort cuts when constructing $\mathcal{S}_{v_y}$ on $T[v_y]$. However, the number of cuts to be sorted is bounded by n due to the linear construction which leads to a contribution of at most one cut for a vertex in the tree. This is obvious from the fact that $|\bar{\mathcal{S}}_{v_x}| \leq 1 + |\mathcal{S}_{v_y}| = 2 + \sum_{i=1}^{k} |\bar{\mathcal{S}}_i| - 1$. But even better, each $\bar{\mathcal{S}}_i$ is already sorted by ρ, so for constructing $\mathcal{S}_{v_y}$ we have to merge $degree(v_y) - 1$ sequences which altogether are at most of length n which can be done in $O(\log(degree(v_y)) \cdot n)$. Line 11 is executed twice for each edge once in each direction, or in other words $degree(v_y)$ times for each vertex v_y. But, the only difference between two executions of line 11 at vertex v_y is that the full cut sequence from one edge e_i is replaced by the full cut sequence on another edge. More precisely, the edge e from v_x to v_y of a previous execution of line 11 becomes one of the edges e_i towards v_i in a subsequent execution while one of the previous e_i becomes e. Hence, we can reuse the sorted $\mathcal{S}_{v_y}$ of the first execution of line 11 for subsequent executions by just removing one of the cut sequences in a T_i and adding one. This can be done linearly in n. Therefore we only need to merge $degree(v_y)$ sorted sequences once for each vertex which leads to $\sum_{v_y \in V(T)} \log(degree(v_y)) \cdot n \leq 2m \cdot n$ and then reuse it for the $degree(v_y) - 1$ remaining executions which can be done in $\sum_{v_y \in V(T)} (degree(v_y) - 1) \cdot n \leq 2m \cdot n$. Hence the overall complexity is $O(n^2)$.

1: Set all $\bar{\mathcal{S}}_v(e)$ to $\emptyset$ and initialize empty queue Q

2: $Q.enqueue(leaves(T))$

3: **while not** $Q.empty()$ **do**

4: $\quad v_y \leftarrow Q.dequeue()$

5: $\quad$ **if** $degree(v_y) = 1$ **then**

6: $\quad\quad v_x \leftarrow neighbors(v_y)$

7: $\quad\quad \bar{\mathcal{S}}_{v_x}([v_x, v_y]) \leftarrow \{\{v_x\}, \{v_x, v_y\}\}$

8: $\quad$ **else if** $\exists$ neighbor v_x s.t. $\bar{\mathcal{S}}_{v_x}([v_x, v_y]) = \emptyset$ **then**

9: $\quad\quad k \leftarrow degree(v_y) - 1$

10: $\quad\quad$ Let $v_1, \ldots, v_k$ be neighbors s.t. $\bar{\mathcal{S}}_{v_y}([v_y, v_i]) \neq \emptyset$

11: $\quad\quad$ Construct $\mathcal{S}_{v_y}$ on $T[v_y]$

12: $\quad\quad$ Construct $\bar{\mathcal{S}}_{v_x}$ on T_y from $\mathcal{S}_{v_y}$

13: $\quad\quad \bar{\mathcal{S}}_{v_x}([v_x, v_y]) \leftarrow \bar{\mathcal{S}}_{v_x}$

14: $\quad a \leftarrow$ number of neighbors of v_x s.t. $\bar{\mathcal{S}}_{v_x}([v_x, v]) \neq \emptyset$

15: $\quad$ **if** $a = degree(v_x) - 1$ **then**

16: $\quad\quad Q.enqueue(v_x)$

17: $\quad$ **else if** $a = degree(v_x)$ **then**

18: $\quad\quad$ **for all** $v \in neighbors(v_x)$ s.t. $\bar{\mathcal{S}}_v([v, v_x]) = \emptyset$ **do**

19: $\quad\quad\quad Q.enqueue(v_x)$

20: **for all** $v \in V(T)$ **do**

21: $\quad$ Construct $\mathcal{S}_v$ on T

22: $\quad ag(v) \leftarrow c(\mathcal{S}_v)$

23: **return** $\min_{v \in V(T)}(ag(v))$

Algorithm 2: $Cut\_strategy(T)$

3.7 Improved Non-Contiguous Strategies: Hybrid Algorithm

In [KC07c] it was proposed to combine the two variants, contiguous and non-contiguous, of the label-based algorithm from Section 3.5 by separating the neighboring vertices into two sets and clearing one using the contiguous and one with the non-contiguous variant. More precisely, for v_y, coming from v_x, we seek to partition the neighbors $V := \{v_2, \ldots, v_m\}$ into two sets of vertices V_1 and V_2. The first set V_1 will be cleared with the non-contiguous procedure. Once all elements of V_1 are cleared the team clears v_y and then proceeds to clear V_2 with the contiguous procedure. We thereby divide the weight of the term $\sum_{2 \leq l < i} w(e_l)$ from equation 3.9 onto two sets. This can greatly reduce the total cost. Figure 3.12 illustrates how such a hybrid strategy would be executed.

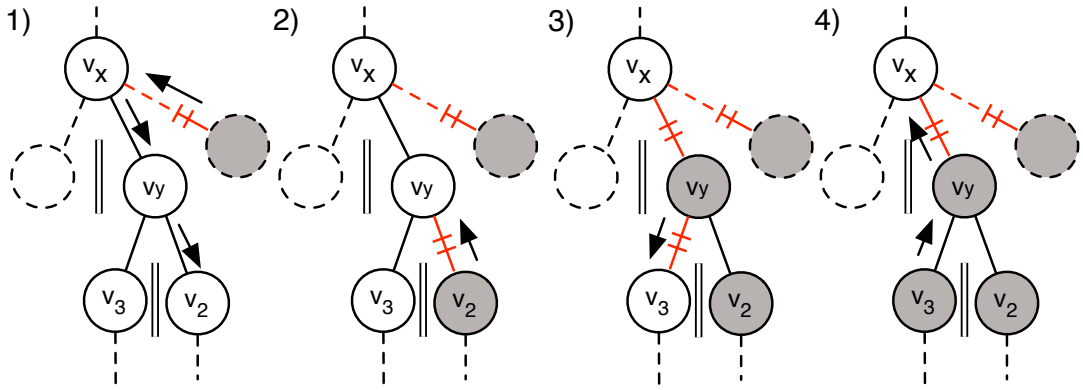


Figure 3.12: Execution of the hybrid strategy.

From fig. 3.12 one complication becomes apparent. Let V_1^x and V_2^x be the partitioning of the neighbors of v_x when coming from yet another vertex v_z. If $v_y \in V_1^x$, then e is not blocked when the team enters v_y, as seen in fig. 3.12. Once we clear v_y we have to add a block on e which increases the total cost while clearing V_2, as seen in steps 3 to 5 in fig. 3.12. If $v_y \in V_2^x$, then the situation

is reversed and we have to add a block on $w(e)$ only while we clear V_1 and not while clearing V_2.

Let us denote the case when $v \in V_1^x$ as case 1 and $v \in V_2^x$ as case 2. We can compute a label for both cases, using the superscripts ^{1} and ^{2}. So the labels on edge e become:

$$h_u^1(V_1, V_2) = \max\left\{\max_{v_i \in V_1}\{c^1(v_i)\}, \max_{v_i \in V_2}\{c^2(v_i) + w(e)\}\right\}$$
$$h_u^2(V_1, V_2) = \max\left\{\max_{v_i \in V_1}\{c^1(v_i) + w(e)\}, \max_{v_i \in V_2}\{c^2(v_i)\}\right\}$$
$$\lambda_{v_x}^1(e) = \max\left\{s(v_y), min_{V_1,V_2}\{h_u^1(V_1, V_2)\}\right\} \tag{3.20}$$
$$\lambda_{v_x}^2(e) = \max\left\{s(v_y), min_{V_1,V_2}\{h_u^2(V_1, V_2)\}\right\} \tag{3.21}$$

where $c(v_i)^j = \lambda_{v_y}^j(e_i) + \sum_{v_l \in V_j, 2 \leq l < i} w(e_l)$ for $j = 1, 2$. It is easy to see, however, that $h_u^1(V_1, V_2) = h_u^2(V_2, V_1)$ given that $\lambda_{v_y}^1(e_i) = \lambda_{v_y}^2(e_i)$, which is the case since we compute the labels from the leaves upward and these equations are identical. It is however, important to note that the partition still has take into account the penalty term $w(e)$, i.e. only to which side it is assigned is not relevant. Hence, to simplify notation, we will drop superscripts ^{1} and ^{2}. The problem now states as follows:

Definition 21 (Hybrid algorithm: optimal partition) *Given v_x, v_y and neighbors $V = \{v_2, \ldots, v_m\}$ as before find a partition of V into V_1 and V_2 s.t. $h_u(V_1, V_2)$ is minimal.*

The proposed algorithm to find partitions will be based on theoretical framework of the next two subsections. First we introduce the concept of batches which cluster vertices and then proceed by developing criteria for optimal partitions into

V_1 and V_2 in Section 3.7.2. On the basis of this we will develop an algorithm in Section 3.7.3.

3.7.1 Batches

The following will be useful to describe at which vertex within a set V the cost is maximal. We shall call a set of all vertices with $\rho_i = a - p$ a batch B_p, where $a := \max\{\lambda_{v_y}(e_i)\}$. The set V can have at most $a - 1$ batches, i.e. $B_1, B_2, \ldots, B_{a-1}$. During the execution of a strategy S in the non-contiguous variant we clear the batches in sequence $B_1, B_2, \ldots, B_{a-1}$ and then clear v. For the contiguous variant the order of clearing is reversed. Define the weight of a batch as $w(B_p) := \sum_{v_i \in B_p} w(e_i)$ and write $w(B_{p<k}) := \sum_{p<k} w(B_p)$. Define the maximum cost within V to be $h := \max_{2 \leq i \leq m}\{c(v_i)\}$ and let v_q be a vertex that assumes this maximum, i.e. $h = c(v_q)$, s.t. $v_q \in B_k$ with k being the largest such possible batch index. Using this notation we can rewrite the maximum cost to be:

$$\begin{aligned} h &= w(B_{i<k}) + w(B_k) - w(e_q) + \lambda(e_q) \\ &= w(B_{i \leq k}) + \rho_q = w(B_{i \leq k}) + a - k. \end{aligned} \tag{3.22}$$

Furthermore, we can define the maximum cost within a batch:

$$h_j := \begin{cases} w(B_{i \leq j}) + a - j & if B_j \neq \emptyset \\ 0 & otherwise \end{cases} \tag{3.23}$$

Clearly $h = \max_{1 \leq j \leq a-1}\{h_j\}$. The following lemma will be relevant for our further results.

Lemma 6 *Let v_q and B_k be as before. Consider any non-empty $B_{k'}$ s.t. $k \neq k'$. If $k > k'$, then $k - k' \leq w(B_{k'<i \leq k})$. Otherwise if $k < k'$, then $w(B_{k<i \leq k'}) \leq k' - k$.*

Proof: First assume $k > k'$. Given h as above, consider the last vertex v_r of another batch $B_{k'}$, i.e. $v_r = v^e_{k'}$ and define $h' := c(v_r) = w(B_{i\leq k'}) + \rho_r$. Recall that $\rho_q = a - k$ and $\rho_r = a - k'$. By assumption $h' \leq h$. This implies

$$\begin{aligned} w(B_{i\leq k'}) + \rho_r &\leq w(B_{i\leq k}) + \rho_q \\ w(B_{i\leq k'}) + \rho_r - \rho_q &\leq w(B_{i\leq k}) \\ k - k' &\leq w(B_{i\leq k}) - w(B_{i\leq k'}) \\ k - k' &\leq w(B_{k'<i\leq k}) \end{aligned} \tag{3.24}$$

which concludes the proof for $k > k'$. The result for $k < k'$ is analogue. □

All classes of vertices in a batch B_p can be listed as:

ρ_i	$a-p$	$a-p$	$\dots$	$a-p$
$\lambda(e_i)$	$a-p+1$	$a-p+2$	$\dots$	a
$w(e_i)$	1	2	$\dots$	p

Using this we can write down all batches and their possible edge types as:

$Batch$	B_1	B_2		$\dots$	B_{a-1}			
ρ_i	$a-1$	$a-2$	$a-2$	$\dots$	1	1	$\dots$	1
$\lambda(e_i)$	a	$a-1$	a	$\dots$	2	3	$\dots$	a
$w(e_i)$	1	1	2	$\dots$	1	2	$\dots$	$a-1$

Table 3.2 shows a set of vertices $V = \{v_2, v_3, \dots, v_{10}\}$ with $a = 10$ and $v_q = v_9$ with maximum cost $c(v_q) = 19$.

3.7.2 Criteria for optimal partitions

All vertices in batches B_i, for $i > k$, do not contribute to the maximum, i.e. a removal of these vertices does not change the maximum cost. We shall call

B_p	B_2	B_3			B_5	B_6	B_7		B_9
v_i	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}
ρ_i	8	7	7	7	5	4	3	3	1
$\lambda(e_i)$	10	8	8	10	7	7	5	5	2
$w(e_i)$	2	1	1	3	2	3	2	2	1
$c(v_i)$	10	10	11	14	14	16	17	19	18

Table 3.2: A simple example of a set of vertices and their assignment into batches.

such vertices the *tail* $T := \bigcup_{i>k} B_i$ of V. Their joint weight shall be denoted by $w(T) = \sum_{v_i \in T} w(e_i)$. As a consequence of lemma 6 we have $w(T_t) < a - k$.

When partitioning V into V_1 and V_2 we shall write $B_{i,1}$, $B_{i,2}$ for the batches of V_1 and V_2, k_1, k_2 for k, $v_{q,1}$, $v_{q,2}$ for v_q, h_{V_1}, h_{V_2} for h and T_1 and T_2 for T. For notational simplicity we will ignore the penalty term in this section and discuss it thereafter when presenting the partitioning algorithm. Finally, for a partition V_1 and V_2 we define a maximization criterion as:

$$c(V_1, V_2) := k_1 + k_2 + w(T_1) + w(T_2) - |h_1 - h_2|. \tag{3.25}$$

Definition 22 (Balanced and full partitions) *Let V be a set of vertices as before. A partitioning of V into V_1 and V_2 is called*

- full *if* $k = k_1 = k_2$,
- balanced *if* $w(B_{i \leq k_1,1}) - k_1 = w(B_{i \leq k_2,2}) - k_2$,
- maximal *if for any other partition V_1', V_2' we get that* $c(V_1, V_2) \geq c(V_1', V_2')$.

It is easy to see that a partition that is full and balanced will minimize h_u and is therefore optimal. Also any full and balanced partition will be maximal.

To show that any maximal partition is optimal we need the following lemma to show that $h_b := w(B_{i\leq k})/2 + a - k$ is a lower bound on h_u.

Lemma 7 *Given V, with a and k as before and any partition V_1 and V_2 we have that:*

$$h_u \geq w(B_{i\leq k})/2 + a - k = h_b. \tag{3.26}$$

Proof: W.l.o.g. assume that $h_{V_1} \geq h_{V_2}$, i.e. $w(B_{i\leq k_1,1}) + a - k_1 \geq w(B_{i\leq k_2,2}) + a - k_2$. So $h_u = h_{V_1}$. Since V has no tail we have that $w(T_1) \leq k - k_1$ and $w(T_2) \leq k - k_2$. Assume that $h_{V_1} < w(B_{i\leq k})/2 + a - k$ which leads to:

$$2 \cdot h_{V_1} < w(B_{i\leq k_1,1}) + w(B_{i\leq k_2,2}) \tag{3.27}$$

$$w(T_1) + w(T_2) + 2a - 2k$$

$$2 \cdot h_{V_1} < w(B_{i\leq k_1,1}) + w(B_{i\leq k_2,2}) \tag{3.28}$$

$$a - k_1 + a - k_2$$

$$h_{V_1} < h_{V_2} \tag{3.29}$$

Which is a contradiction to $h_{V_1} \geq h_{V_2}$ and concludes the proof.□

For full and balanced partitions we have $h_u = h_b$. But a full and balanced partition may not exist and hence we have to consider maximal partitions.

Lemma 8 *If V_1, V_2 is a maximal partition of V, then h_u is minimal, i.e. the partition is optimal.*

Proof: W.l.o.g. assume that $k_2 \leq k_1$. As before we have $w(B_{i\leq k}) = w(B_{i\leq k_1,1}) + w(T_1) + w(B_{i\leq k_2,2}) + w(T_2)$ and $|h_1 - h_2| = |(w(B_{i\leq k_1,1}) + k_1) - (w(B_{i\leq k_2,2}) + k_2)|$. This leads to the following cases:
Case 1: assume $h_u = h_1 > h_2$ and we get

$$h_2 - h_1 + k_2 - k_1 = w(B_{i\leq k_2,2}) - w(B_{i\leq k_1,1}). \tag{3.30}$$

Now:

$$\begin{aligned} h_b - h_u &= w(B_{i\leq k})/2 - w(B_{i\leq k_1,1}) + k_1 - k \\ &= \frac{1}{2}\left(h_2 - h_1 + k_2 + k_1 + w(T_2) + w(T_1)\right) - k \end{aligned} \tag{3.31}$$

By the maximal property we get that $h_b - h_1$ is maximal and the partition is therefore optimal.

Case 2: assume $h_u = h_2 > h_1$ and analogue to the previous case this results in:

$$h_b - h_2 = \frac{1}{2}\left(h_1 - h_2 + k_2 + k_1 + w(T_2) + w(T_2)\right) - k \tag{3.32}$$

which is again maximal by the maximal property of the partition. Hence a maximal partition is optimal.□

In colloquial terms, we have to find a partition with the largest k_1, k_2 and large tails T_1, T_2 and with $w(B_{i\leq k_1,1})$ roughly equal to $w(B_{i\leq k_2,2})$.

3.7.3 The partitioning algorithm

The algorithm is based on a dynamic programming approach motivated by the relation of the maximization criterion to the subset sum problem, one of the early NP-complete problems [GJ79]. In short, the subset sum problem is to determine whether a set of integer values contains a subset whose values sum up to some given integer z. A dynamic programming algorithm to solve it runs in pseudo-polynomial time $O(Cn)$ where C is the sum of all members of the set and n is the number of elements. Translated to our case this becomes the problem to determine whether V contains a set of vertices V_2 s.t. the sum of the weight of their respective edges $w(V_2)$ sums up to $z = \lceil w(V)/2 - w(e)/2 \rceil$. Here $w(e)$ is the penalty term from equation 3.20. A solution V_2 would minimize h_u given that $V_1 = V \setminus V_2, V_2$ is a full partition, i.e. it satisfies $k_1 = k_2 = k$. Obviously,

using the dynamic programming approach for solving the subset sum problem gives no guarantee that $k_1 = k_2 = k$. In fact, such a partition may not even exist. The following will be concerned with an algorithm that guarantees to find a full partition if one exists.

Let A be a table with $m-1$ rows and $z = \lceil w(V)/2 - w(e)/2 \rceil$ columns. Set $A(0,j) := 0, \forall j$ and $A(i,0) := 0, \forall i$. Each row represents a vertex and they shall be ordered as $v_m, \ldots, v_2$, i.e. v_m corresponds to row one, v_{m-1} to row two and so on. Write c_i for $w(e_{m-i+1})$, i.e. the cost added to V_2 by adding the vertex in row i. If $c_i > j$, then $A(i,j) = A(i-1,j)$, otherwise $A(i,j) = \max\{A(i-1,j), A(i-1, j-c_i) + c_i)\}$. An entry $A(i,j)$ in the table is then the maximal weight for V_2 achievable using vertices $v_m, \ldots, v_{m-i+1}$. The table is filled as usual for the subset sum problem. If an entry in A exists s.t. $A(i,j) = \lceil w(V)/2 - w(e)/2 \rceil$, then we have a partition that is optimal with respect to to the distribution of the edge weights onto V_1 and V_2. This is, however, only one part in the optimization. An entry in A represents possibly multiple partitions, some of which do not satisfy that $k_1 = k_2 = k$. A particular partition can be thought of as a path within the table. Finding an optimal partition is hence the problem of finding an entry with $A(i,j) = \lceil w(V)/2 - w(e)/2 \rceil$ for which we have a path that represents a maximal partition. We will show how to compute whether such a path exists for the case of full partitions.

Since we ordered the vertices in reverse order we can view the problem from the perspective of adding vertex by vertex with decreasing index to V_2 as we proceed through the rows of A. For V_1 we can view it as if we are removing vertices with decreasing index from V_1. The main question is what happens to k_1 for V_1 and k_2 for V_2 as we remove and add vertices. When we add a vertex $v \in B_{u,1}$ from V_1 to V_2 we know that all other vertices in V_2 are in batches $B_{j \geq u,2}$.

Write $V_1' = V_1 \setminus \{v\}$ and $V_2' = V_2 \cup \{v\}$. Define $S(V_2) := \sum_{1 \leq i \leq k_2} w(B_i, 2)$ to be the support of V_2. Now if $k_2 - u > S(V_2)$, then $v = v_q'$ will be the new maximum for V_2'. Otherwise, if $k_2 - u \leq S(V_2)$, then $v_q = v_q'$. To illustrate this with our example set of vertices simply choose $V_2 = \{v_9\}$. Clearly $v_{q,2} = v_9$ and $S(V_2) = 2$ and adding v_5 will lead to $v_{q,2}' = v_5$. Similarly for V_1', when removing v with associated edge e_v, the support will be reduced to $S(V_1') = S(V_1) - w(e_v)$. Now the maximum $v_{q,1}'$ may shift to a vertex of a lower batch if $\exists B_b$ s.t. $k_1 - b > S(V_1)$, otherwise it will remain at it former vertex s.t. $v_{q,1}' = v_{q,1}$.

As long as $k_1 = k_2 = k$ we know that $S(V_1) = w(V_1), S(V_2) = w(V_2), w(T_1) = w(T_2) = 0$ and we do not need to keep track of these values. Once we add a vertex $v \in B_{u,1}$ from V_1 to V_2 with $k_2 - u > S(V_2)$ we will have $k_2' < k$ and the path will not be a valid solution. Let us define two further tables $K_1(i,j)$ and $K_2(i,j)$ in which we will keep track of k_1 and k_2. For our case the computation of $K_1(i,j)$ and $K_2(i,j)$ involves only a simple check, whether upon addition/removal of the vertex the current K_1 and K_2 can be maintained. If this is not the case we discard the solution path by setting $K_1(i,j) = 0$ or $K_2(i,j) = 0$. The pseudo code in 3 shows how to compute A, K_1 and K_2. Initially we set $K_1(0,j) = K_2(0,j) = k$. It is obvious that k_1, k_2 are monotonically decreasing with respect to to growing i, j, except for the special case for V_1 if we remove the first vertex v_2 in the last row of the table and at this point have $v_{q,1} = v_2$ and $B_{b_1,1} = \{v_2\}$, i.e. there is no other vertex in its batch. Dealing with this special case merely complicates notation without changing the methodology and we will therefore ignore it. Now, an entry $A(i,j) = z$ with $K_1(i,j) = K_2(i,j) = k$ has a path that represents a full and balanced partition which is therefore optimal. If no such entry exists, then neither does a full and balanced partition. The algorithm for this case of full and balanced partitions illustrates how to use the theoretical results of this paper to obtain an improved algorithm for the Graph-Clear problem on trees.

if $c_i > j$ **then**

$A(i,j) \leftarrow A(i-1,j)$

$K_1(i,j) \leftarrow K_1(i-1,j)$

$K_2(i,j) \leftarrow K_2(i-1,j)$

else

$A(i,j) = \max\{A(i-1,j), A(i-1,j-c_i) + c_i)\}$

if $A(i,j) = A(i-1,j-c_i) + c_i$ **then**

if $\rho_2 < K_1(i-1,j-c_i) - (w(V) - A(i,j))$ **then**

$K_1(i,j) \leftarrow 0$

else

$K_1(i,j) \leftarrow K_1(i-1,j-c_i)$

if $a - \rho_{m-i} < K_2(i-1,j-c_i) - A(i-1,j-c_i)$ **then**

$K_2(i,j) \leftarrow 0$

else

$K_2(i,j) \leftarrow K_2(i-1,j-c_i)$

if $A(i,j) = A(i-1,j)$ **then**

if $K_2(i-1,j) \geq K_2(i,j)$ and $K_1(i-1,j) \geq K_1(i,j)$ **then**

$K_1(i,j) \leftarrow K_1(i-1,j)$

$K_2(i,j) \leftarrow K_2(i-1,j)$

Algorithm 3: *Compute_table_entry*(i,j)

Trying to obtain an algorithm for the general case entails more complications. In essence, a compromise between obtaining balanced edge weight and large k_1, k_2 has to be sought. More precisely, we have to consider all parts of the maximization criteria $c(V_1, V_2)$ to identify optimal partitions. Extending the previous dynamic programming approach with brute force would mean to evaluate all possible paths in the table leading to any entry $A(i, j)$ and choosing one for which the maximization criteria is largest. Obviously, this is not efficient. To see how we could arrive at a more efficient method let us define $C(i, j)$ to be the largest value of $c(V_1, V_2)$ across all partitions that lead to a path to $A(i, j)$. More precisely, $C(i, j)$ is the largest $c(V_1, V_2)$ for all partitions V_1, V_2 s.t. $V_2 \subset \{v_m, \ldots, v_{m-i+1}\}$ with $w(V_2) = A(i, j)$. Computing C involves keeping track not only of k_1 and k_2 in tables K_1, K_2, but also of the tails T_1 and T_2 for whose weight we also need tables $T_1(i, j)$ and $T_2(i, j)$ and requires therefore some more bookkeeping. Using equation 3.25 for $c(V_1, V_2)$ we get that:

$$C(i,j) = K_1(i,j) + K_2(i,j) + T_1(i,j) + T_2(i,j) \tag{3.33}$$

$$- |H_1(i,j) - H_2(i,j) - w(e)| \tag{3.34}$$

$$H_1(i,j) = w(V) - A(i,j) - T_1(i,j) - K_1(i,j), \tag{3.35}$$

$$H_2(i,j) = A(i,j) - T_2(i,j) - K_2(i,j). \tag{3.36}$$

where K_1, K_2, T_2, T_1 now describe a partition that minimizes $C(i, j)$. Since the support of a set V_1 is s.t. $w(V_1) = S(V_1) + w(T_1)$ we also know about the size of the support of V_1 and analogue for V_2.

The key problem for finding maximal partitions efficiently is to find a way to compute $C(i, j)$ from the entries for A, K_1, K_2, T_1, T_2 at $(i-1, j)$ and $(i-1, j-c_i)$. We do already know how K_1, K_2, T_1 and T_2 evolve when adding a vertex v_{m-i+1}. Hence we can identify whether the path from $(i-1, j)$ or from $(i-1, j-c_i)$ leads to a better partition w.r.t to $C(i, j)$. One problem, however, is that it not known

whether it is possible that a partition at $(i-1,j)$ or $(i-1,j-c_i)$ leads to an optimal partition at (i,j) while not being optimal for $C(i-1,j)$ or $C(i-1,j-c_i)$ respectively.

To see this more clearly consider a partition for $A(i,j)$ that maximizes $C(i,j)$. Now the vertex for row i, i.e. v_{m-i+1}, is either in V_2 or V_1. Now if $v_{m-i+1} \in V_2$ for this partition, then the partition $V_2' = V_2 \setminus \{v_{m-i+1}\}, V_1' = V \setminus V_2'$ is a partition that leads to a path to $A(i-1,j-c_i)$. There is no proof yet that this partition V_2', V_1' maximizes $C(i-1,j-c_i)$. If this is not the case, then a path leading to a maximal partition for (i,j) passing through $(i-1,j-c_i)$ can be different from the optimal path to entry $(i-1,j-c_i)$. This has dramatic consequences, as we cannot build our solution path row by row but would have to reconsider all possible paths to an entry. Let us illustrate this with an example in table 3.3 and 3.13. Table 3.14 shows some partitions represented by paths that lead to entry $A(5,4)$. These three partitions from 3.14 $V_2' = \{v_3, v_6\}$, $V_2'' = \{v_3, v_4, v_5\}$ and $V_3''' = \{v_3, v_4, v_7\}$. If we assume that the penalty term $w(e) = 1$, then the minimum h^u for these partitions is

$$h_u(V \setminus V_2', V_2') = \max \left\{ \max_{v_i \in V \setminus V_2'} \{c(v_i)\}, \max_{v_i \in V_2'} \{c(v_i) + w(e)\} \right\} = 7 \tag{3.37}$$

for V_2', while V_2'' and V_2''' have $h_u(V \setminus V_2'', V_2'') = h_u(V \setminus V_2''', V_2''') = 8$. Evidently V_2' is also a maximal partition and hence a solution for the example. The key problem, however, is to distinguish the partition V_2' already at entry $A(4,3)$ as the number of possible partitions grows exponentially and we seek a way to compute $C(i,j)$ efficiently. At entry $A(4,3)$ we have already three possible partitions, i.e. $V_2' \setminus v_3, V_2'' \setminus v_3$ and $V_2''' \setminus v_3$. At this point we should already be able to make a choice which partition can lead to maximal partitions at later entries such as $A(5,4)$. Currently, we only have the maximization criteria $C(i,j)$ to evaluate partitions which does not necessarily lead to future optimal partitions as we add

V	v_2	v_3	v_4	v_5	v_6	v_7
ρ_i	6	5	4	3	2	1
$\lambda(e_i)$	7	6	6	4	5	2
$w(e_i)$	1	1	2	1	3	1
$c(v_i)$	7	7	8	8	10	10

Table 3.3: Another example of vertices.

vertices. For the case in the example, using the maximization criteria also leads to $V_2' \setminus v_3$ being the optimal partition for $A(4,3)$, but this need not be the case in general.

Column		1	2	3	4	5	6	7
v_i	$w(e_i)$							
v_7	1	1	1	1	1	1	1	1
v_6	3	1	1	3	4	4	4	4
v_5	1	1	2	3	4	5	5	5
v_4	2	1	2	3	4	5	6	7
v_3	1	1	2	3	4	5	6	7
v_2	1	1	2	3	4	5	6	7

Figure 3.13: The dynamic programming table for the example from table 3.3.

Assuming we solve the previously mentioned issue, i.e. we can compute the best partition based on the previous partitions efficiently, the results on how k_1, k_2 change upon addition or removal of vertices can be used for an attempt to find general solutions. For a set of vertices V_1 we have that $w(V_1) = w(T_1) + S(V_1)$ and hence we can chose to either keep track of the support $S_1(i,j)$ or the tail $T_1(i,j)$ as each can be computed from $A(i,j)$ and the other. For the following we choose to compute $S_1(i,j)$ explicitly. The entries for table $C(i,j)$ are also implicitly known through $A(i,j), w(V), S_1(i,j), S_2(i,j), K_1(i,j)$ and $K_2(i,j)$.

It is clearly visible in the pseudo-code from 4 at which point we assume that we have a criteria for the evaluation of partitions at $(i-1,j)$ and (i,j) which allows

if $c_i > j$ **then**

$[K_1(i,j), S_1(i,j)] \leftarrow [K_1(i-1,j), S_1(i-1,j)]$

$[K_2(i,j), S_2(i,j)] \leftarrow [K_2(i-1,j), S_2(i-1,j)]$

else

if $A(i-1, j-c_i) + c_i = A(i,j)$ **then**

if $a - \rho_2 < K_1(i-1, j-c_i) - S(i-1, j-c_i) + c_i$ **then**

$K_1(i,j) \leftarrow \max\{k_{q_i} | k_{q_i} < K_1(i-1, j-c_i) - S(i-1, j-c_i)\}$

$S_1(i,j) \leftarrow w(B_{i \leq K_1(i,j)})$

else

$K_1(i,j) \leftarrow K_1(i-1, j-c_i)$

$S_1(i,j) \leftarrow S_1(i-1, j-c_i) - c_i$

if $a - \rho_i < K_2(i-1, j-c_i) - S(i-1, j-c_i)$ **then**

$K_2(i,j) \leftarrow a - \rho_i$

$S_2(i,j) \leftarrow c_i$

else

$K_2(i,j) \leftarrow K_2(i-1, j-c_i)$

$S_2(i,j) \leftarrow S_2(i-1, j-c_i) + c_i$

if $A(i-1,j) = A(i,j)$ and partition at $(i-1,j)$ better than at (i,j) **then**

$[K_1(i,j), S_1(i,j)] \leftarrow [K_1(i-1,j), S_1(i-1,j)]$

$[K_2(i,j), S_2(i,j)] \leftarrow [K_2(i-1,j), S_2(i-1,j)]$

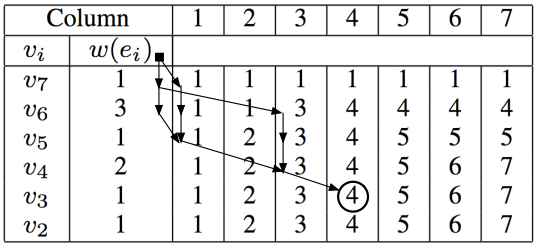
else

$[K_1(i,j), S_1(i,j)] \leftarrow [K_1(i-1,j), S_1(i-1,j)]$

$[K_2(i,j), S_2(i,j)] \leftarrow [K_2(i-1,j), S_2(i-1,j)]$

return $K_2(i,j)$

Algorithm 4: *Compute_table_entries_at*(i,j)



Column		1	2	3	4	5	6	7
v_i	$w(e_i)$							
v_7	1	1	1	1	1	1	1	1
v_6	3	1	1	3	4	4	4	4
v_5	1	1	2	3	4	5	5	5
v_4	2	1	2	3	4	5	6	7
v_3	1	1	2	3	(4)	5	6	7
v_2	1	1	2	3	4	5	6	7

Figure 3.14: Possible partitions resulting from the dynamic programming table for the example from 3.3. A partition is represented by a sequence of arrows where a diagonal arrow means that the vertex of the row to which the arrow is pointing is in V_2 while a horizontal arrow indicates that the vertex is in V_1.

us to compute future partitions maximizing $C(i, j)$. One could use the current value of the maximization $C(i, j)$ as a heuristic. It is not clear whether $C(i, j)$ actually satisfies the aforementioned assumptions, so it is not guaranteed that a maximal partition is found. Another minor detail in the algorithm is that the assignment of the vertex in the last row v_2 may lead to K_2 increasing and needs to be dealt with separately in practical implementations. One way around this, however, is to adopt a different perspective on the problem than before. It was useful to consider vertices to be added to V_2 and removed from V_1 when searching for full and balanced partitions, but for the general case another variant seems more elegant. Instead of V we consider a subset of vertices $V^i = \{v_m, \ldots, v_{m-i+1}\}$ to be partitioned into V_1^i, V_2^i at entry $A(i, j)$. The pseudo-code from 5 shows how the formulas change when we take this approach that consider $V^i := \{v_m, \ldots, v_i\}$ to be partitioned into V_1^i, V_2^i at entry $A(i, j)$. This means when going to $A(i+1, j)$ we either add v_i to V_1^i or to V_2^i which leads to similar formulas for both of these sets and resolves the details for assigning v_2. A solution to the problem can now be found only in the last row $m - 1$ of the dynamic programming table, namely in column j with $C(m - 1, j)$ maximal.

if $c_i > j$ **then**

$[K_1(i,j), S_1(i,j)] \leftarrow [K_1(i-1,j), S_1(i-1,j)]$

$[K_2(i,j), S_2(i,j)] \leftarrow [K_2(i-1,j), S_2(i-1,j)]$

else

if $A(i-1, j-c_i) + c_i = A(i,j)$ **then**

if $a - \rho_i < K_2(i-1, j-c_i) - S(i-1, j-c_i)$ **then**

$K_2(i,j) \leftarrow a - \rho_i,\ S_2(i,j) \leftarrow c_i$

else

$K_2(i,j) \leftarrow K_2(i-1, j-c_i)$

$S_2(i,j) \leftarrow S_2(i-1, j-c_i) + c_i$

$[K_1(i,j), S_1(i,j)] \leftarrow [K_1(i-1, j-c_i), S_1(i-1, j-c_i)]$

if $A(i-1,j) = A(i,j)$ **then**

Compare partitions when coming from $A(i-1,j)$ with $A(i-1, j-c_i)$

if $A(i-1,j)$ leads to a better partition **then**

if $a - \rho_i < K_1(i-1,j) - S(i-1,j)$ **then**

$K_1(i,j) \leftarrow a - \rho_i,\ S_1(i,j) \leftarrow c_i$

else

$K_1(i,j) \leftarrow K_1(i-1,j),\ S_1(i,j) \leftarrow S_1(i-1,j) + c_i$

$[K_2(i,j), S_2(i,j)] \leftarrow [K_2(i-1,j), S_2(i-1,j)]$

else

if $a - \rho_i < K_1(i-1,j) - S(i-1,j)$ **then**

$K_1(i,j) \leftarrow a - \rho_i,\ S_1(i,j) \leftarrow c_i$

else

$K_1(i,j) \leftarrow K_1(i-1, j-c_i),\ S_1(i,j) \leftarrow S_1(i-1, j-c_i) + c_i$

$[K_2(i,j), S_2(i,j)] \leftarrow [K_2(i-1,j), S_2(i-1,j)]$

return $K_2(i,j)$

Algorithm 5: *Compute_table_entries_at*(i,j)

3.7.4 Discussion and Conclusion

We presented a new approach for finding strategies for Graph-Clear in a tree which requires solving a partitioning problem. We presented criteria for optimal partitions and based on these we presented an algorithm that computes optimal partitions given that a full and balanced partition exist. We also presented an approach that can compute maximal partitions, but whose complexity depends on whether one can find a method to evaluate partitions at an entry (i, j) for their potential to maximize later entries in the dynamic programming table. Even without resolving this problem the algorithm always returns better strategies on trees than the algorithm from Section 3.5 since in the worst case it degrades to either the contiguous or non-contiguous variant. On the other hand, in a realistic applications of Graph-Clear a reduction by a few robots is already significant decrease in costs and therefore an optimal solution for the partitioning is of interest. Furthermore, from a graph-theoretical perspective the investigation whether optimal solutions for general strategies on trees exist motivates further analysis of the hybrid method in combination with the optimal contiguous algorithm from Section 3.6. For all practical purposes, however, we believe the current progress for finding Graph-Clear strategies already gives a good basis for using it in applications. Therefore, we shall conclude the investigation of deterministic strategies and extend Graph-Clear to accommodate probabilistic sensor failure models in the next section.

3.8 Probabilistic Graph-Clear

This section introduces a probabilistic extension for Graph-Clear to accommodate uncertainty in sensing. Sensors are described with a footprint and a false negative

probability, i.e. the probability of failing to report a target within their sensing range. This extension leads to a new algorithm that allows to answer new design and performance questions, namely 1) how many robots are needed to obtain a certain confidence that the environment is free from intruders, and 2) given a certain number of robots, how should they coordinate their actions to minimize their failure rate.

One of the limits of our previous study has been the *deterministic* assumption. To be precise, in the previous sections we have assumed an error-free sensing process. In fact, it was hypothesized that whenever an intruder was within the sensing range of a robot its presence was always reported. In practice, however, no sensor is error-free. Hence, we extend our previous deterministic formulation to include possibly faulty sensors. In particular, we here account for sensors that may give false negatives, i.e. with a certain known probability they may report that no intruder is within their sensing range even if there is one or more. The use of robots equipped with these faulty sensors naturally leads to a number of design and performance questions, like for example the following:

- if after having cleared an environment the robot team reports that no intruders were found, what is the probability that instead n intruders (with n being a positive natural number) successfully managed to remain undetected due to errors in the sensing process?

- given r robots, what is the clearing strategy that minimizes the probability that one or more intruders remain undetected?

- given a certain environment and a target probability p, how many robots are needed in order to be sure that if the team reports no detection, then with probability at least p there are indeed no intruders in the environment?

The main contribution of this section is twofold. First, the deterministic model is extended into a probabilistic one in order to account for faulty sensors. This modification entails a number of updates in the model components that are fully worked out in this dissertation. Moreover, an appropriate model for faulty sensors is presented. Secondly, in the light of the new model, one of the algorithms we formerly developed for the deterministic case is extended in order to answer the above questions.

3.8.1 Probabilistic Model

In order to extend the Graph-Clear formalism from a deterministic to a probabilistic scenario, various concepts need to be accordingly updated or introduced. Before getting into the details we clarify one important point concerning the remainder of the discussion. From now on we concentrate on the case of robots not reporting intruders. We hereby assume that when a robot reports an intruder this event is separately handled, e.g. a human operator is dispatched, a tracking behavior is triggered or the like. Also, while we assumed the possibility of false negatives, we do not assume false positives, i.e. an intruder is detected only when it is really present. For this reason the event *intruder detected* never occurs in the discussion. Our interest in this paper is in drawing conclusions upon a sequence of negative observations reported by the robots.

3.8.1.1 Worst case adversary

As previously mentioned, the deterministic Graph-Clear framework has been formulated under a worst case scenario. More precisely, in the deterministic scenario this hypothesis implies targets moving with unbounded speed and with complete knowledge of the environment and of the actions of all robots. Hence, when-

ever a strategy leaves room for recontamination, it will certainly happen. To maintain this idea, the worst case adversary has to be differently defined when faulty sensors are used. In the probabilistic scenario the worst case adversary still has complete knowledge of the robots' positions, as well as of their sensors error rates. Each of the intruders will try to maintain their *undiscovered* status by crossing blocks or sweeps where the highest error rate occurs. As anticipated, these crossing events must occur, since ultimately all elements in the graph will be swept or blocked. This concept will be clearer after the probabilistic sensor model will be formally specified.

3.8.1.2 Environment

A probabilistic surveillance graph is similar to a deterministic surveillance graph, with the important difference that instead of $w : V \cup E \to \mathbb{N}^+$ we introduce $w : V \cup E \to \mathcal{F}$, where the set $\mathcal{F}$ is defined as follows:

$$\mathcal{F} := \{\ f \mid f : \mathbb{N} \to [0,1], f(0) = 1, \forall r, r' \in \mathbb{N}\ r \geq r'$$

$$\Rightarrow f(r) \leq f(r')\}.$$

That is, in the probabilistic case each graph element is not associated with a constant cost, but rather with a monotonically decreasing function mapping the natural numbers to the interval $[0, 1]$. Throughout the paper, with a slight abuse of notation, we will write $w_x(r)$ for $w(x)(r)$ for some $x \in V \cup E$ and $r \in \mathbb{N}$. Also, in order to ease the discussion, whenever we write x we mean either a vertex or an edge. What was previously understood as the weight, namely the number of robots needed for a block or sweep, now becomes a function defining the probability of a false negative (i.e. no intruder reported even if there was at least one passing through the sensor footprint during the block or sweep) while

executing a block or a sweep using a certain number of robots. According to the intuition, for every x, $w_x(0) = 1$, i.e. if no robot is used then the probability of not reporting any intruder is always 1. Using more robots hence leads to an improvement in the detection capabilities.

3.8.1.3 Probabilistic Actions and Probabilistic Strategies

As a consequence of the new definition of w, the notion of a strategy also needs to be extended. While it was formerly defined as a sequence of actions which essentially determines which block and sweep operations are executed at which time step, a probabilistic strategy now describes the number of robots allocated to a each sweep or block of an action.

Definition 23 (Probabilistic action) *The* probabilistic action set *of a probabilistic surveillance graph G is $\mathbb{N}^{n+m}$ where each element $a = \{a_1, \ldots, a_{n+m}\}$ (called* probabilistic action*) has an associated cost $c(a) = \sum_{i=1}^{n+m} a_i$.*

The reader should observe the fundamental difference with the deterministic case, where actions are elements of $\{0,1\}^{n+m}$. In the deterministic scenario a block or a sweep is either executed or not. In the probabilistic case these operations are instead executed using a certain number of robots. Also, due to the way the set $\mathcal{F}$ was defined, we relax the explicit requirement that edge blocks are executed concurrently to vertex sweeps. With the new definition of probabilistic action, if these blocks are not executed than the corresponding function yields a probability of non detection equal to 1, rendering such a strategy useless. The functions w_x are the essential probabilistic element in the graph G.

3.8.1.4 Undetected intruders and false negatives

We describe the number of undetected intruders in the environment with the discrete random variable T, and we write $p(T = i), i \in \mathbb{N}$, for its mass distribution. For each edge or vertex x we will write $p_x(N|t, r)$ for $w_x(r)$, i.e. the probability of a false negative. A negative observation is written as N and the event of a target crossing is written as t. The number of robots that are executing the corresponding sweep or block operation on x is given by r. For notational convenience we will drop the r and write $p_x(N|t)$ assuming that there is a set number of robots. Keeping in mind that each x is blocked or swept once during the execution of a fixed strategy S, let $\bar{N}$ denote a sequence consisting of only negative observations during the execution of S. We are interested in studying the following probability, that is here written using Bayes rule:

$$p(T = i|\bar{N}) = \frac{p(\bar{N}|T = i) \cdot p(T = i)}{p(\bar{N})} \tag{3.38}$$

where for a fixed strategy $p(\bar{N})$ is a normalization constant. Now, $p(T = i)$ is simply the prior target distribution of the number of targets, and the most important part is $p(\bar{N}|T = i)$ which we intend to relate to w_x. It is in this term that the smart target assumptions has significant consequences. We restrict the collection of observations to robots engaged in a sweep or a block. Recall that an undetected target after the entire strategy is executed must have crossed either through a block or been in a vertex during a sweep. The undetected target then crosses from the contaminated part to the cleared part at some x. An individual smart target will choose its crossing point x so that $p_x(N|t)$ is largest. A smart group of i targets acting in a cooperative way will each choose an x s.t. $p(\bar{N}|T = i)$ is maximized. This is an important distinction since the probability for detections may be different for sensors that are more sensitive when multiple targets cross at once. In the general case, for each x we should then specify $p_x(N|T_c = i)$,

i.e. the probability of a negative observation given that i targets cross x during its block or sweep. Notice that T_c is not T but a random variable associated to x describing the number of targets crossing. However, it is not practical to specify $p_x(N|T_c = i)$ for all i and doing so also leads to more complications. It may be the case that i targets in the environment achieve the lowest likelihood of detection if split into $i_1 + i_2 = i$ with some crossing on $x_1 \in G$ and some on $x_2 \neq x_1 \in G$ if $p_x(N|T_c = i) < p_{x_1}(N|T_c = i_1) \cdot p_{x_2}(N|T_c = i_2), \forall x \in G$. For simplicity we prefer to describe the sensing in terms of $p_x(N|t) = p_x(N|T_c = 1)$. Therefore we assume that all targets crossing an element x are detected independently. This assumption leads to $p_x(N|T_c = i) = p_x(N|t)^i$. In this case a group of smart targets would all choose to cross at the same x with $p_x(N|t)$ largest, but not necessarily at the same time. Alternatively, we could have chosen e.g. $p_x(N|T_c = i) = p_x(N|t), \forall i$, then a group of smart targets will all choose to cross at x with $p_x(N|t) = p_x(N|T_c = 1)$ largest and they will choose to cross at once. This choice has different implications and the choice should ideally coincide with the actual sensor properties. To summarize, all this taken together we now get that $p(\bar{N}|T = i) = p_{x_{max}}(N|t)^i$ where $x_{max} = argmax_{x \in G}\{p_x(N|t)\}$ from the assumption that targets are detected independently, choose the worst case path and have complete knowledge about actions of the robot team. Before we proceed with an algorithm that computes probabilistic strategies let us shortly discuss how to obtain p_x, or equivalently w_x, from basic sensing on the grid.

3.8.2 Modeling faulty sensors

The actual sensors of the robot team are described by their footprint, i.e. their coverage of part of the grid g representing the environment, and their probability for misses which may differ for each cell the sensor covers. They produce obser-

vations in discrete time intervals and may at any given interval either return a positive observation, i.e. a flagged target detection or a negative observation, i.e. no target detection. Fig. 3.15 shows an example of a sensor placed in a grid and shows its coverage. In general, the sensor may have any number of cells covered and any probability associated with any cell, so there is no restriction on the type of sensor except that its covered area is discretized on a grid.

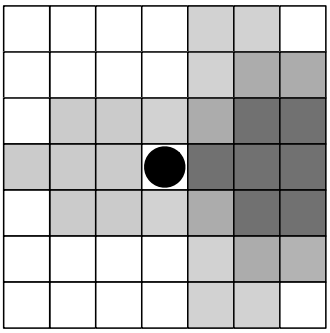


Figure 3.15: A grid with a sensor placed in its center as a black circle and with the cells observed by the sensor in grey. A darker grey tone denotes a smaller false negative probability.

From the sensing probabilities on individual cells, we need to obtain probabilities for the blocks and sweeps. The details on how to execute these actions, however, are only given when they are actually implemented in a specific application. In the deterministic model the requirement for the implementations on a block are that it has to guarantee that an intruder attempting to cross the block will be detected. In the probabilistic scenario we can modify this into having the intruder cross through at least one grid cell covered by a sensor so that the probability of detecting it is non-zero. The main difference, however, is that we can increase the probability of detecting the intruder if we utilize more robots than the minimum necessary, hence the monotonically decreasing trend for the function w_x for $x \in G$. Given that we assumed worst case targets, we assume

that a target chooses the path with the smallest probability of being detected. The probability for a miss on the edge will then become the probability of a miss of a target on this path. Fig. 3.16 shows two robots blocking a hallway and the worst case target path as well as four robots blocking and leading to a higher probability of detecting the target.

For an edge e let $C_t = \{g_1, \ldots, g_{n_p}\}$ be a set of grid cells that are covered by the sensors of the block on e and that are traversed by the target on its worst case path through the block. Now the probability of a miss on the block of e becomes $p_e(N|t) = \Pi_{i=1}^{n_p} p_{g_i}(N|t)$, where $p_{g_i}(N|t)$ is the probability of a miss on grid cell g_i. Hence a target can only pass undetected if it is undetected on each cell. Note that the equation assumes independence in the detections. Here $p_{g_i}(N|t)$ is given by $p_{g_i}(N|t) = \Pi_{j=1}^{n_{g_i}} p_{s_j}$ where $\{s_1, \ldots, s_{n_{g_i}}\}$ are the n_{g_i} sensors covering cell g_i. One may increase the probability of detecting the target by increasing the number of robots for the block action. This may not always be possible, depending on the constraints of the environment. We may e.g. not be able to add additional robots to a block due lack of space or we may need many more robots to yield an improvement. The latter case is seen in fig. 3.17. This is the motivation for having w_x as a general function that is only required to be monotonically decreasing and starting at $w_x(0) = 1$ and otherwise unrestricted, since it can then also capture the above cases.

An analogue case can be made for a sweeping routine, even though its derivation is slightly more tedious to describe and hereby omitted. Either way, we assume a sweep routine also gives a final w_x, i.e. in this case the probability that a negative observation is made when targets are present in the corresponding vertex v_i and r robots are used.

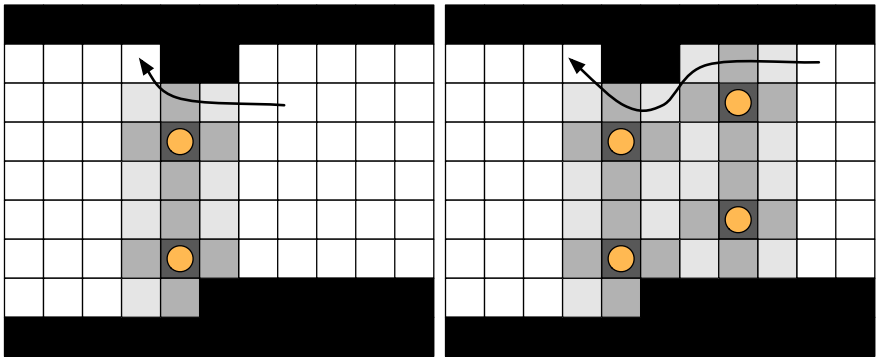


Figure 3.16: An illustration of the computation of detection probabilities for blocks through the worst case path a target can take through a block.

3.8.3 Probabilistic Extensions to Graph-Clear

In this part the label-based algorithm from Section 3.5 is extended to a variant that considers the probabilistic nature of the sensors. We focus on the conservative scenario with worst case adversarial targets and the model introduced in the previous section. Recall that for each edge and vertex we are given a monotonically decreasing function $w_x : \mathbb{N} \to [0, 1]$, which gives us a miss probability $p_x(N|t, r)$ when using r robots for the block or sweep on x. In the light of the model just developed, the last two questions raised in the introduction can then be reformulated as follows:

1. Given a desired $p(T = 0|\bar{N}) \in [0, 1]$ how many robots are needed?

2. Given r robots what is the strategy producing the highest $p(T = 0|\bar{N})$?

Recall the computation in the deterministic algorithm for Graph-Clear of a label $\lambda_{v_x}(e)$ on an edge $e = [v_x, v_y]$. The label $\lambda_{v_x}(e)$ is the number of robots needed to clear all vertices beyond e when coming from v_x towards v_y. At this point we are considering the neighboring vertices $v_1, \ldots, v_m$ of v_y different from v_x and compute $\lambda_{v_x}(e) = \max\{s(v_y), \max_{i=1,\ldots,m}\{c(v_i)\}\}$.

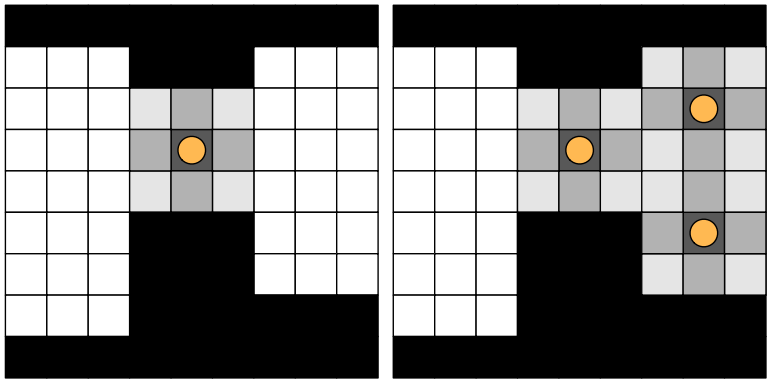


Figure 3.17: The basic block in this figure only needs one robot, while the first reinforcement leading to an improvement in the detection capability needs two additional robots. In order to get this fact the reader should consider that intruders may also move diagonally on the grid.

For the probabilistic variant we can employ a similar reasoning, except that we now need to get a function $\lambda^e_{v_x}(r)$ instead of just a label, analogue to having a function w_x instead of just a constant weight on a vertex or edge. $\lambda^e_{v_x}(r)$ will return the probability of failing to report one or more targets for all clearing steps when moving from v_x towards v_y and clearing all neighboring subtrees with r robots. Due to the assumption about target movement the overall probability for a miss will be the maximum of all probabilities of misses during any of the steps. Let us illustrate how to construct $\lambda^e_{v_x}(r)$. For each w_x we have a minimum number of robots required to get a miss probability of less than 1. Formally we have for each $x \in G$ a r_{min,w_x} with

$$r_{min,w_x} = \min\{r \in \mathbb{N} \mid w_x(r) < 1\}.$$

Using this r_{min,w_x} as the deterministic weight $w(x)$ on each $x \in G$ for the deterministic Graph-Clear algorithm we can compute $\lambda_{v_x}(e)$ as before using equation 3.10. Now, obviously $\lambda^e_{v_x}(r) = 1, \forall r < \lambda_{v_x}(e)$, i.e. if we use less robots than

$\lambda_{v_x}(e)$ we have at least one block or sweep that does not have sufficiently many robots, and the miss probability will be 1.

Let us now describe the procedure that computes $\lambda^e_{v_x}(r)$. We will consider

$$F_{v_x,e} := \{\lambda^{e_1}_{v_y}, \ldots, \lambda^{e_m}_{v_y}, w_{e_1}, \ldots, w_{e_m}, w_{v_y}\}$$

a set of functions instead of fixed weights and labels, the key difference to the deterministic case. For each of these $f \in F$ we will have an auxiliary term r_f which denotes the current argument for f, i.e. how many robots are allocated to the respective edge for blocking the subtree or for clearing it. Also, let *deterministic_contiguous_label*(G, v_x, e) be a function that computes the deterministic label $\lambda_{v_x}(e)$ on G using the current r_f. Hence, it computes $\lambda_{v_x}(e)$ according to equation 3.10 but with $w(e_i) = r_{w_{e_i}}$ and similarly for $\lambda_{v_y}(e_i) = r_{\lambda^{e_i}_{v_y}}$. Intuitively, this computes the current cost for clearing the subtree rooted at v_y given a certain cost assignment for each f.

Algorithms 6 and 7 shows the entire procedure to compute $\lambda^e_{v_x}(r)$. Algorithm 6 can be used to compute all label functions by first considering all leaves and then moving upwards through the tree and process all non-leaves that have all neighbors but one with label functions already computed. This part of the procedure identical to the computation of deterministic labels on a tree. Algorithm 7 shows this in detail. Its complexity is $O(e \cdot B \cdot d^2 \cdot \log(d))$. Here $e = |E|$, B is a bound given as input and d is the maximum vertex degree. The complexity results from line 14 in algorithm 7 which is itself $O(d \log(d))$ and executed within the while loop $O(d \cdot B)$ times. Finally, algorithm 7 is called $O(e)$ times in the queuing in algorithm 7.

To illustrate the algorithm let us consider two examples. First, let v_y be a leaf. In this case $F_{v_x,e} = \{w_{v_y}\}$ and $\lambda^e_{v_x}(r) = w_{v_y}(r)$. Secondly, consider the slightly more complicated example in fig. 3.18. All $\lambda^{e_i}_{v_y}$ are available since v_1, v_2, v_3 are

1: Set all label functions to 0 and initialize empty queue O
2: $O.enqueue(leaves(G))$
3: **while not** $O.empty()$ **do**
4: $v_y \leftarrow O.dequeue()$
5: $Compute\_lambda\_function(v_x, v_y, B)$
6: $a \leftarrow$ number of neighbors of v_x s.t. $\lambda_{v_x}^{[v_x,v]}$ is computed
7: **if** $a = degree(v_x) - 1$ **then**
8: $O.enqueue(v_x)$
9: **else if** $a = degree(v_x)$ **then**
10: **for all** $v \in neighbors(v_x)$ s.t. $\lambda_v^{[v,v_x]}$ not computed **do**
11: $O.enqueue(v_x)$

Algorithm 6: $Compute\_all\_lambda\_functions(T, B)$

just leaves. The algorithm then computes $\lambda_{v_x}^{e}(r)$ by first trying the smallest possible values $r_{f,min}$ for each function involved and then allocates additional robots to those functions that determine the overall maximum to reduce the probability that a target can cross undetected until the overall cost is larger than allowed. The necessity for checking whether we can improve the functions that determine the maximum is that the maximum cost during the clearing may occur when clearing v_1, v_2, v_3 or v_y, since each vertex is cleared at a different step in the strategy. Let us assume the maximum cost that determines the deterministic label occurs when clearing vertex v_2. The function that determines the maximum for the probability of letting a target through may, however, be w_{v_3}. If we use less robots than the maximum cost at v_2 while clearing v_3 then we could try to use more to decrease $w_{v_3}(r_{w_{v_3}})$ without increasing the overall number of robots needed as long as it is still below what we need during the clearing step for v_2.

1: **if** $degree(v_y) == 1$ **then**

2: $\quad \lambda^e_{v_x} \leftarrow w_{v_y}$

3: **else**

4: $\quad$ Locate all neighbors $v_1, \ldots, v_m$ and create $F_{v_x,e}$.

5: $\quad$ Initialize r_f to $r_{f,min}$ for all $f \in F$

6: $\quad deterministic\_contiguous\_label(v_x, e)$

7: $\quad$ Set $\lambda^e_{v_x}(r) = 1, \forall r < \lambda_{v_x}(e)$

8: $\quad$ **for** $r \leftarrow \lambda_{v_x}(e)$ to B **do**

9: $\quad\quad repeatable \leftarrow true$

10: $\quad\quad$ **while** $repeatable$ **do**

11: $\quad\quad\quad f_{max} \leftarrow argmax_{f \in F_{v_x,e}} f(r_f)$

12: $\quad\quad\quad addbots \leftarrow argmin_{i \in \mathbb{N}}(f_{max}(r_{f_{max}} + i) < f_{max}(r_{f_{max}}))$

13: $\quad\quad\quad r_{f_{max}} \leftarrow r_{f_{max}} + addbots$

14: $\quad\quad\quad deterministic\_contiguous\_label(v_x, e)$

15: $\quad\quad\quad$ **if** $\lambda_{v_x}(e) > r$ **then**

16: $\quad\quad\quad\quad r_{f_{max}} \leftarrow r_{f_{max}} - addbots$

17: $\quad\quad\quad\quad repeatable \leftarrow false$

18: $\quad\quad\quad\quad \lambda^e_{v_x}(r) \leftarrow f_{max}(r_{f_{max}})$

Algorithm 7: $Compute\_lambda\_function(v_x, v_y, B)$

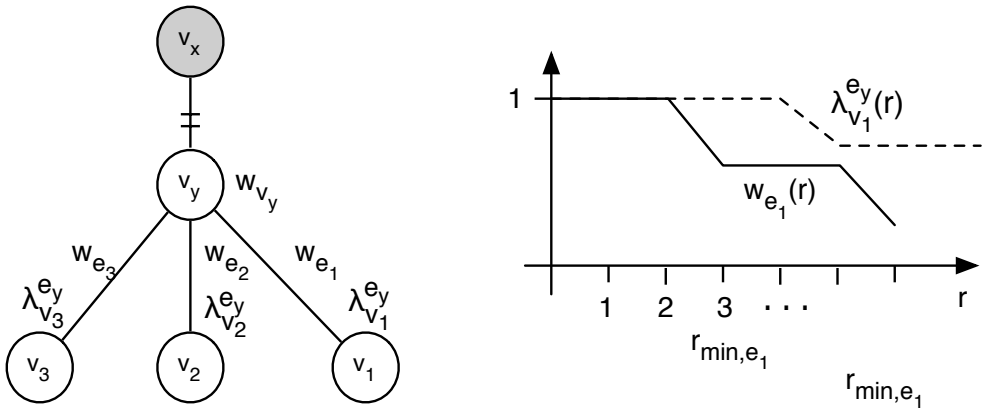


Figure 3.18: An example of algorithm 7

Once all lambda functions are computed up to using B robots, both questions we formulated can be answered. For each possible starting vertex $v \in G$ we can compute a probabilistic strategy by only considering the lambda functions computed for the neighbors. To determine the number of robots needed for a certain $p(T = 0)$ we set all auxiliary terms r_f for the lambda functions, w_v and all w_x on the edges of v s.t. each function achieves a value small enough s.t. we get the desired $P(T = 0)$ from equation 3.38 by having the appropriate $P(\bar{N}|T = i)$. Note that we may not obtain $p(T = 0)$ exactly since the lambda functions are discrete, but we get the next possible value smaller or equal. Finally, to obtain the strategy we proceed exactly as for the deterministic variant utilizing equation 3.11 to obtain the final cost for a strategy starting at v. Now we do this for every starting vertex and select the best as a desired starting point and return the needed number of robots. To determine the best $p(T = 0)$ reachable with a certain number of robots we simply compute the lambda function of a virtual edge from a new vertex v to a vertex $v_i \in G$. Fig. 3.19 illustrates this graphically. This lambda function can now be read and the value for the available number of robots can be determined. Doing this for every $v_i \in G$ and selecting the best

starting vertex yields the answer to our the last question.

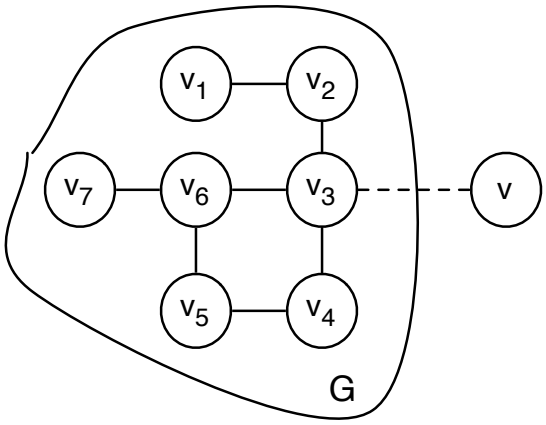


Figure 3.19: Adding a virtual edge to compute the cost of clearing G starting from v_3. The lambda function on the virtual edge, represented as a dotted line will represent the probability of clearing everything beyond that edge.

3.8.4 Discussion and Conclusion

In this section we presented a probabilistic extension to the Graph-Clear formalism we formerly introduced under deterministic assumptions. The probabilistic extension allows to model faulty sensors that may return false negatives, i.e. they may fail to detect an intruder crossing their sensing range. This failure rate is characterized by a probability distribution that can be easily computed once specific sensors are considered. The introduction of this function is the main change to the deterministic model and entails a different concept of a solution strategy for the problem at hand. Building upon this change in the model, a formerly developed algorithm for the deterministic case has been extended to answer performance and design questions introduced in the introduction. It is worth outlining that while extending the formalism towards the probabilistic scenario we have retained the hypothesis of worse case adversaries, i.e. intruders

that have complete knowledge of the environment, of the position of the robots and of their error rates. The results presented in this dissertation have to be interpreted accordingly, namely they describe the system performance when facing the smartest possible set of intruders. The benefit of the probabilistic extension is a more accurate reasoning about the inherent uncertainty in sensing and reflects the reality of robotic applications more accurately than the deterministic approach. More work in this direction is envisioned and we could also consider uncertainty in the robots motion when computing the miss rates for blocks and sweeps, e.g. robots may with a small probability not execute a block accurately due to faulty information about their location and give a target the opportunity to pass through undetected.

3.9 Modified Graph-Clear: Sweeps Prevent Recontamination

There is another extension that can be added to Graph-Clear, this time with slightly less modifications than in Section 3.8. This modification presented here mainly arises from consideration on Line-Clear in Chapter 4 and it will be used therein.

Recall the discussion in Section 3.1 in which we compared weighted edge-searching with Graph-Clear. One of the main differences highlighted was that we relaxed the requirement for a vertex action, our vertex sweeps, so that it does not need to prevent recontamination paths through the vertex. The reasoning for this is to enable a broader range of algorithms to become implementations for vertex sweeps. But in some applications it may well be that the sweeping implementation can additionally guarantee that no intruder crosses through the actual

vertex. This alleviates the need for the edge blocks of a vertex while the sweep action is applied, since the vertex sweep would prevent recontamination. At first sight this modification seems to lead to a problem more similar to edge-searching. Yet, this is not the case since after the sweep it is still more economical to block the edges instead of leaving all robots inside the vertex. And surprisingly, it does not change the core ideas of the algorithms to compute strategies. Let us now outline the proposed modification in more detail and show how the contiguous label-based algorithm from Section 3.5 can be modified to accomodate the modification.

First, it is useful to extend the weight function to have directional weights. More precisely, if vertex sweeps prevent recontamination it can matter from which direction the vertex is being swept since it can lead to different costs for avoiding contamination during the sweep. We shall see examples of this in Chapter 4. Notice that in a contiguous strategy on a tree all vertices, except the first, are always entered from exactly one blocked edge while no other edges are blocked. After the sweep all other edges are blocked while the first edge is not anymore. Hence, we extend the weight function to describe a directional weight. Let us redefine $w : (V \times E) \cup E \to \mathbb{N}$ and write $w(v_y, e)$ for the cost of clearing v_y entering from edge e. We also drop the requirement for the blocking while sweeping, but instead assume that $w(v_y, e) \geq w(e)$. Now for all v_y that are leaves the label definition changes to:

$$\lambda_{v_x}(e) := w(v_y, e) \tag{3.39}$$

Once the labels towards leaves are computed we can consider vertices which are not leaves but for which all edges, except one, have an outgoing label. More precisely let v_x, v_y be neighbors and $m = degree(v_y) - 1$. Write neighbors of v_y different from v_x as $v_1, \ldots, v_m$. When coming from v_x the first step is always to

clear v_y, since the strategy has to be contiguous and v_x is assumed cleared. Then vertices of the contaminated subtrees rooted at the neighbors $v_1, \ldots, v_m$ can be cleared. To simplify matters we assume that once a subtree v_i, $i = 1, \ldots, m$ is entered the robot team clears all its vertices, comes back and then clears another subtree. Following this simple procedure the goal is now to clear the subtrees in an order that is least costly. Write $e_i := [v_y, v_i]$. It turns out that ordering $v_1, \ldots, v_m$ s.t. $\rho_i = \lambda_{v_y}(e_i) - w(e_i)$ is descending and then clearing $v_m, \ldots, v_1$ in this order has minimum overall cost. The cost at step i is given by the cost of clearing subtree v_i and blocking all e_i towards subtrees that are still contaminated. Assuming that we order indices by ρ_i we can write this as:

$$c(v_i) := \lambda_{v_y}(e_i) + \sum_{l=1}^{i-1} w(e_l). \tag{3.40}$$

The new label for v_x then becomes the maximum of clearing v_y itself and the maximum cost occurred while clearing any one of the subtrees.

$$\lambda_{v_x}(e) = \max\{w(v_y, e), \max_{i=1,\ldots,m}\{c(v_i)\}\}. \tag{3.41}$$

Given these definitions, computing all labels in a tree is straightforward. Once these are computed the overall cost of clearing the tree when starting at vertex v is determined by:

$$ag(v) = \max\{w(v), \max_{1 \le i \le m}\{\lambda_v e_i + \sum_{l=1}^{i} w(e_i)\}\}, \tag{3.42}$$

where now $v_1, \ldots, v_m$ are all neighbors of v, i.e. $m = degree(v)$ and

$$w(v) := \min_{e \in Edges(v)}\{w(v, e) + w(e)\}. \tag{3.43}$$

Since we redefined w we need this to denote the cost of clearing v coming from no edge. It emulates blocking e and then clearing v from there while keeping it blocked. Finding the best starting vertex leads to the best possible strategy that

clears each subtree in a depth-first manner. The key parts of the algorithm are very similar to the one in Section 3.5. The other algorithms are expected to be extendable in a similar manner. The only key difference to note is that when clearing a vertex is a tree, one enters it exactly from one previously blocked edge, apart from the very first vertex, and leaves blocks at all other edges after clearing it. We shall see in Chapter 4 that this modification has practical relevance for two dimensional pursuit-evasion scenarios.

3.10 Applying Strategies to Graphs

For most environments a surveillance graph of an environment in a realistic application is usually a graph. Since the general problem on graphs is NP-hard, we need to develop heuristics or approximations in order to obtain practical solutions. One convenient method is to calculate the minimum spanning tree (MST) on a graph and then block all edges s.t. only MST edges remain. Let us denote all edges not belonging to the MST as cycle edges. The cycle edges would be blocked throughout the execution of the strategy for the MST. Clearly, we can do a lot better. Once the MST is computed we only need to block cycle edges that connect a contaminated and a cleared vertex during the clearing process. We shall call this approach dynamic cycle blocking and the former constant cycle blocking. To quantify the improvement we carried out the following simple experiments.

Random graphs with a given number of vertices and edges are created with random weights in the range 1 to 12 for vertices and 1 to 6 for edges. Once the graph is constructed we compute the MST and pick the root in the center of the longest path in the tree and compute all labels into the direction of the root vertex. The labels and the total weight of all cycle edges give the cost for the

Table 3.4: Reduction of the number of robots needed when using dynamic cycle blocking expressed in terms of the percentage of the number of agents needed to block all cycles at once.

Edges per vertex	1	1.5	3
20 Vertices	47.74%	41.69%	40.40%
30 Vertices	55.49%	45.09%	42.85%
40 Vertices	62.24%	47.00%	45.32%

constant cycle blocking. To determine the costs for the dynamic cycle blocking we execute the strategy and add and remove cycle blocks as necessary. We defined 9 sets of parameters, 3 sets with 20 vertices with 20, 30 and 40 edges, 3 sets with 30 vertices with 30, 45, and 60 edges and 3 sets 40 vertices with 40, 60 and 80 edges. For the experiments we constructed 1000 graphs for each set of parameters.

Figure 3.20 shows the number of robots needed for clearing the graph for each set of parameters. It is apparent that the costs for constant cycle blocking are considerably higher and that the difference between constant and dynamic cycle blocking becomes greater when more edges are present. Table 3.10 shows what percentage of the total costs for all cycle blocks we are saving when blocking cycles dynamically. While these experiments are very limited in scope and validity they still serve as a strong indicator that the dynamic cycle blocking can lead to a significant improvement, in particular as the number of vertices grows.

Another approach for computing better strategies for graphs is found in [HSK09] and [HKS08]. Therein an anytime algorithm that iteratively tries out new spanning trees is given. In practice the algorithm will be terminate when a strategy is required by the robot to start clearing. Their particular algorithm

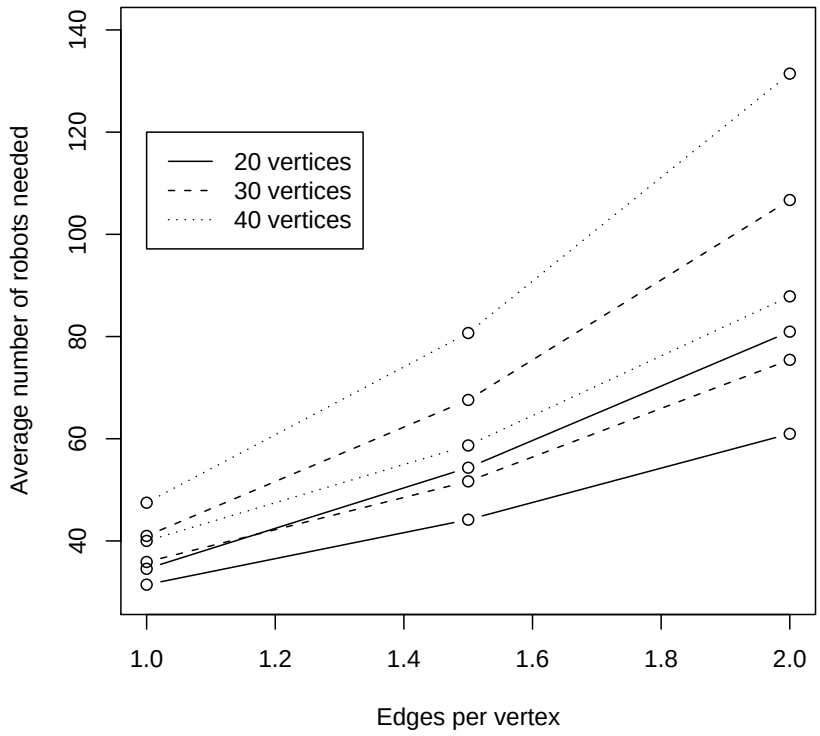


Figure 3.20: Results of the experiments with 9 sets of parameters. The upper lines for each number of vertices is always the cost for the constant cycle blocking strategy and the lower for the dynamic cycle blocking strategy.

is concerned with edge-searching, but the idea of trying multiple spanning trees extends trivially to Graph-Clear.

3.11 Discussion and Conclusion

In this paper we presented a novel theoretical framework to model surveillance tasks performed by multiple robots with limited sensing capabilities. The approach we presented has two major advantages. First, it produces coordination

plans, called strategies, that abstract from low level sensing details. Limited sensing capabilities of robots in the team are accounted for by assuming that multiple robots are needed in order to perform the basic operations, i.e. blocking a connection between two areas, or sweeping an area. Secondly, by formalizing it into a well characterized graph optimization problem we were able to leverage a significant amount of former graph-related literature and gain significant insights into its computational structure. After having established the formal framework and determined its computational complexity, we turned our attention to the tractable case of trees. To make these algorithms useful in practice a conversion of surveillance graphs into surveillance trees is discussed as well. We presented an algorithm for the special case of trees and contiguous strategies that is optimal. Contiguous strategies are more restricted, so in general one can expect strategies that are not required to be contiguous to need less robots. The existence of a polynomial algorithm capable of producing an optimal non-contiguous strategy for trees is an open question. As progress towards this question we presented a hybrid method in Section 3.7 that combines previous contiguous and non-contiguous labeling methods and produces non-contiguous strategies that in certain situations can outperform contiguous strategies. This hybrid method, however, is based on a dynamic programming approach and its complexity is pseudopolynomial and it is not complete. This tradeoff therefore justifies the use of the optimal algorithm for contiguous strategies presented in this paper with polynomial complexity. Contiguous strategies are not only interesting from a mere theoretical point of view. For example, in certain situations recontamination of a cleared room should be avoided because it could be used to deploy infrastructures that would be negatively impacted by intruders. Yet, it may be an interesting venue for future work to consider a combination of the hybrid algorithm that uses the optimal contiguous algorithm instead of the simpler label-based algorithm as a

basis. This may lead to the development of a pseudopolynomial but optimal algorithm for non-contiguous strategies on trees.

We have defined Graph-Clear with the objective of minimizing the number of robots needed to detect all intruders, but one could aim for the optimization of different parameters. For example, if one considers the motion model of the robots being used, then one could look for strategies that are fast to execute, or minimize energy consumption, etc. Finally, one issue is the detailed implementation of sweep operations for a given class of robots and sensors and the automated extraction of graphs from robot maps. We shall turn to both of these problems in the remaining chapters of this dissertation.

CHAPTER 4

Line-Clear: Multi-Robot Pursuit-Evasion in 2d

In this chapter we introduce a novel multi-robot pursuit-evasion problem for two dimensional environments. This problem can be seen as the natural extension of visibility-based pursuit-evasion to sensors with limited range, since for both problems the environment, targets and robots are otherwise the same. Yet, to make sense of the limited range our formalization differs quite significantly and the algorithms bear little to no resemblance. This is expected, however, since our interpretation of limited range, particularly by associating a cost to sensing on large distances, changes the problem drastically. From another perspective, the Line-Clear problem can be considered an analogue to Graph-Clear for two dimensional environments. In fact, there is a practical relationship between Graph-Clear and Line-Clear in as much as the former can be used to find solutions to the latter. The development of Line-Clear started when carrying out the first experiments with Graph-Clear on real robots, presented in Section 6.1. Therein only two robots were available and it was desirable to have a frontier moving forward to avoid having to block edges. This frontier became the concept of a sweep line.

Recall that in pursuit-evasion scenarios targets are generally assumed to be worst-case adversaries and hence capable to exploit any weakness the robot team may exhibit. Hence the robot team needs to restrict all possible target movements until targets cannot possibly escape, which then guarantees a detection. In two dimensional environments the fundamental ability a robot team needs for this is

to restrict target movement between obstacles. Hence, the key idea for Line-Clear is to abstract the capabilities of the robot team to *sensing on a line*. This sensing on a line presupposes that the robot team is capable of coordinating in order to completely cover a line in the environment with its sensors. The actual footprint of the sensors is not directly relevant. Only the cost in terms of the number of robots needed to cover a line of a given length is needed.

It is important to note that the *sensing on a line* abstraction has serious implications with respect to applicability and usefulness of the algorithm presented in this manuscript. In particular, it is suitable for limited sensing ranges, i.e. when the range is shorter than most distances between disjoint obstacles. With large sensing ranges a single robot could potentially cover the area of two separate sweep lines, which our model does not take into consideration and it would then compute a cost of at least two robots. But given a scenario in which a single robot is generally not capable of covering multiple lines simultaneously, the presented approach is viable.

We shall first proceed with the basic definitions of Line-Clear and then show how to compute so called sweep schedules using Voronoi Diagrams and Graph-Clear strategies. These sweep schedules are, however, not optimal in all cases and hence not, in the strict sense, solutions to the Line-Clear problem. Yet, it is a practical method to compute sweep schedule that in some cases will not perform worse. It also serves as a good introduction to the problem of computing optimal sweep schedules for simply connected environment which we shall address in Section 4.3 and Section 4.4. The question whether it is possible to compute optimal sweep schedules is still open and we present an approach that clearly shows at what part one could either prove NP-hardness or a property that will guarantee polynomial complexity. For this we also focus on contiguous

and progressive sweep schedules. Many more open questions arise when considering non-contiguous or non-progressive sweep schedules. Another open question is also whether recontamination can improve sweep schedules. Future work on these questions may well use the relationship between Graph-Clear and Line-Clear that we will encounter within this chapter. We shall also revisit Line-Clear in Chapter 5 as a method to obtain surveillance graphs from an environment.

4.1 Line-Clear: Definitions

We assume that free space of the environment is given as a bounded and connected set $\mathcal{E} \subset \mathbb{R}^2$, endowed with the Euclidean metric $\|\cdot\|$. Obstacles are given as a finite collection of convex sets $C_1, \ldots, C_{n_o} \subset \mathbb{R}^2$. We write $C = \bigcup_{i=1}^{n_o} C_i$ for all obstacles. Let δA be the boundary of set A. We further assume that $\delta\mathcal{E} \subset \delta C$ and $interior(\mathcal{E}) \cap interior(C) = \emptyset$, i.e. $\mathcal{E}$ is bounded by obstacles. The environment $\mathcal{E}$ is possibly multiply connected from a topological point of view.

It should be noted that many non-convex sets can be represented as the union of multiple convex sets, therefore our assumptions about the shape of the obstacles are not too demanding. Only obstacles with a boundary that is a non-convex and non-linear curve cannot be represented in this manner. In this case a finite union of convex obstacles can only be an approximation. Fig. 4.1 illustrates a typical environment considered in this paper, where free space is white and obstacles are gray. Nonetheless, the convexity requirement is consistent with the literature on Voronoi Diagrams and the Generalized Voronoi Graph [CB94] and causes no problems in practice, in particular since robot generated maps are themselves often discretized approximations. We now introduce a formal definition of sweep lines and further definitions to compose them to sweep $\mathcal{E}$.

Definition 24 (Sweep line) *A sweep line l is an ordered set of $n > 1$ points in $\mathbb{R}^2$ written $l = [x_1, \ldots, x_n]$ with $x_1 \in \delta C_i$ and $x_n \in \delta C_j$ for some $j \neq i$. Furthermore none of the line segments $[x_k, x_{k+1}], 1 \leq k < n$ intersects interior(C) nor with each other within interior$(\mathcal{E})$[1]. $S(n)$ denotes the set of all sweep lines with n points. Additionally let a simple sweep line be a sweep line that satisfies $x_k \in$ interior$(\mathcal{E}), 1 < k < n$.*

The above definition is both formal and operative, i.e. a sweep line is introduced as a finite collection of concatenated segments, and therefore it can be easily represented by the sets of its endpoints. The reader should note that it is convenient to allow (non simple) sweep lines to have middle points on obstacles in order to have a sweep line that can then be split into two sweep lines (see for example sweep line a in Fig. 4.1). A sweep line $[x_1, x_2, x_3]$ can then be split into two sweep lines $[x_1, x_2]$ and $[x_2, x_3]$ if x_2 is on an obstacle. The latter two sweep lines cover the same points as the first. Hence, for the purpose of removing contamination these can be considered identical. To make this precise we now define *point sets* for a set of sweep lines which follow naturally from the interpretation of sweep lines as chains of line segments. This is then used to define equivalence of sweep lines.

Definition 25 (Sets of sweep lines and point sets) *Define $\mathcal{S} = \bigcup_{i=2}^{\infty} S(i)$ to be the set of all sweep lines, and let $S \subset \mathcal{S}$ be a set of sweep lines. For $l \in S$, let $n(l)$ be the number of points of $l \in S$. Then we define,*

$$\bar{P}(S) = \bigcup_{l \in S} \bigcup_{i=1}^{n(l)-1} [x_i, x_{i+1}]$$

as the point set of S.

[1] *Intersections of line segments are allowed only on δC.*

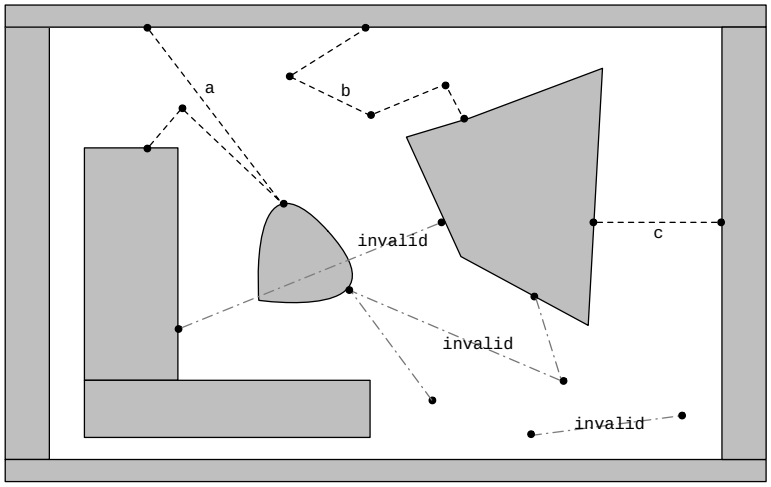


Figure 4.1: Examples of points that are valid sweep lines a), b) and c). The remaining lines are invalid and not sweep lines.

In plain words, $\bar{P}(S)$ is the set of points covered by sweep lines in S. This simple definition is needed in order to establish equivalence between sweep lines and sets of sweep lines.

Definition 26 (Equivalent sweep lines) *Two sweep lines $l_1, l_2 \in \mathcal{S}$ are equivalent if $\bar{P}(\{l_1\}) = \bar{P}(\{l_2\})$. Similarly, two sets of sweep lines S, S' are equivalent if $\bar{P}(S) = \bar{P}(S')$.*

From now on we consider two equivalent sweep lines equal. The next definition introduces the concept of time by moving the points of a sweep line continuously. Such *moving sweep lines* will become the basis of sweep schedules.

Definition 27 (Moving sweep line) *A moving sweep line is a function $l : \mathbb{R}^+ \rightarrow S(n)$, i.e. $l(t) = [x_1(t), \ldots, x_n(t)]$ such that $x_i(t)$ is continuous $\forall i \in \{1, \ldots, n\}$.*

Next, we introduce the concept of *sweep schedule*, i.e. a set of moving sweep lines whose cardinality can grow or shrink at given points in time.

Definition 28 (Sweep schedule) *A sweep schedule is a function* $\tau : [0, T] \rightarrow \mathcal{P}(\mathcal{S})$*, where* $\mathcal{P}(\mathcal{S})$ *is the power set of* $\mathcal{S}$*,* $T \in \mathbb{R}^+$ *and* $|\tau(t)|$ *is finite* $\forall t \in [0, T]$*. Additionally, we require that at time* t*:*

1. $\tau(t) = \{l_1(t), \ldots, l_{s(t)}(t)\}$*, where* $s(t) \in \mathbb{N}^+$ *and each* $l_i(t)$ *is a moving sweep line.*

2. *for every* $l_i(t)$ *there* $\exists\ t_s, t_e$ *with* $t_s \leq t < t_e$ *s.t.*

   (a) $l_i(t') \in \tau(t'),\ \forall t' \in [t_s, t_e)$

   (b) $\exists\ \delta > 0$ *s.t.* $l_i(t') \notin \tau(t'),\ \forall t' \in [t_s - \delta, t_s) \cup [t_e, t_e + \delta]$

The second part of the definition of a sweep schedule may not seem necessary at first. It essentially requires a sweep schedule to be a set of moving sweep lines with each moving sweep line first appearing at time t_s and not being present anymore at t_e. Without this addition it would be possible to construct a sweep schedule τ such that at some time t_1 we have $\tau(t_1) = \{l_1\}$ and $\tau(t_1 + \epsilon) = \{l_1, l_2\}, \forall \epsilon \in \mathbb{R}^+ \setminus \{0\}$. This means that l_2 does not appear at any time t but in the limit of $\epsilon \rightarrow 0$, which is not desirable nor useful for clearing contamination. The additional requirement also allows us to immediately describe an sweep schedule as moving sweep lines and let it be continuous. The alternative would be to use continuity and optimality, as defined below, to show that there always exists an optimal sweep schedule that is continuous and then show that all continuous sweep schedules can be written in terms of appearing and disappearing moving sweep lines. This may be desirable from a theoretical standpoint and leads to a cleaner definition of sweep schedules, but it serves no practical purpose for the robotic application. Hence we impose the restriction rather than deriving it and can directly create sweep schedules from sets of moving sweep lines by simply stating when a new sweep line is added and when existing sweep lines are

removed. Next, we define continuity of a sweep schedule from a topological point of view.

Definition 29 (Continuity of sweep schedules) *A sweep schedule $\tau : [0,T] \to \mathcal{P}(\mathcal{S})$ is continuous if the composite map $P \circ \tau : [0,T] \to \mathcal{P}(\mathcal{E})$[2] satisfies the following condition: $\forall t \in [0,T]$ and any neighborhood[3] N of $\bar{P}(\tau(t))$ in $\mathcal{E}$, there exists a neighborhood U of t such that $\forall t' \in U$ $\bar{P}(\tau(t')) \subseteq N$.*

Having formally defined sweep lines and sweep schedules, we can now connect back to our original problem setting. Given that we are interested in robots with restricted sensing capabilities, it is evident that if a sweep line represents a moving sensor arrangement, a certain number of robots will be needed to implement it. The next definition models this number as the *cost* to implement a sweep line.

Definition 30 (Cost of sweep lines) *To each sweep line $l = [x_1, \ldots, x_n] \in \mathcal{S}$ we associate a cost $c(l)$ defined as $\sum_{i=1}^{n-1} R(\|x_i - x_{i+1}\|)$. Here R is a non-decreasing function $R : \mathbb{R}_0^+ \to \mathbb{N}^+$ with $R(0) = 0$ which determines the number of robots needed to cover a segment of a particular length. For a finite set of sweep lines S we define the cost $c(S)$ as the sum of the costs of all sweep lines in S. For a sweep schedule τ define the cost $c(\tau) = \max_{t\in[0,T]}\{c(\tau(t))\}$.*

The definition of $c(\tau)$ outlines that the cost of a sweep schedule is defined in a worst case scenario, i.e. it is the minimum number of robots needed in order to implement it. An alternative definition for the cost could be $c_{alt}(l) = R(\sum_{i=1}^{n-1} \|x_i - x_{i+1}\|)$ which can generally lead to a lower cost for sweep lines that have midpoints. In colloquial terms, the definition of cost for c_{alt} makes

[2] *Notice that the power set of $\mathcal{E}$ is much larger than the range of $P \circ \tau$ since $|\tau(t)|$ is finite and is chosen for notational convenience.*

[3] *neighborhoods are open sets*

sense only if sensors can sense across multiple straight line segments up until a certain distance, such as omnidirectional sensors or generally sensors with a large field of view. In contrast, a sensor that can only sense on a straight line, such as a single laser, cannot cover any two adjacent line segments if they are not collinear. Even though the difference seems small at first, the choice of cost function has some implications which we will point out whenever they are relevant. Before concluding this section with the precise problem definition, we formalize the concept of recontamination path formerly introduced in colloquial terms.

Definition 31 (Path) *A path between two points $x_1, x_2 \in \mathcal{E}$ is a continuous curve between x_1 and x_2 not intersecting C.*

Definition 32 (Cleared and contaminated points) *A point x is cleared at time t if $x \in \bar{P}(\tau(t))$. Furthermore, a point x cleared at time t is also cleared at time $t' > t$ if $\nexists$ a path from x to a contaminated point y in $\mathcal{E}$ at any time $t'' \in [t, t']$ that does not intersect $\bar{P}(\tau(t''))$. If x is not clear it is called contaminated. At $t = 0$ all points in $\mathcal{E} \setminus (\bar{P}(\tau(0)))$ are contaminated. Write $\mathcal{R}(t)$ for all cleared points and $\mathcal{C}(t)$ for all contaminated points at time t.*

We can now distinguish a special type of sweep schedule with the additional requirement that the set of cleared points is connected at all times. Such sweep schedules are denoted as *contiguous*, analogue to the formulation of contiguity for graphs in Graph-Clear but extended to point sets. Similarly, it is also useful to define progressiveness for sweep schedules.

Definition 33 (Contiguous Environment Sweeps) *Let τ be a sweep schedule. If $\mathcal{R}(t)$ is a connected set $\forall t \in [0, T]$, then τ is contiguous.*

Definition 34 (Progressive Sweep Schedules) *Let τ be a sweep schedule. If $\mathcal{R}(t) \subseteq \mathcal{R}(t') \leftrightarrow t \leq t'$, then τ is progressive.*

Finally, for Line-Clear the goal of a sweep schedule is to remove all contamination from an initially fully contaminated environment at minimal cost.

Definition 35 (The (contiguous) Line-Clear problem) *Given a contaminated $\mathcal{E}$ with $C_1, \ldots, C_{n_o}$ as above, the Line-Clear problem is to find a sweep schedule τ_{opt} with minimal cost $c(\tau_{opt})$ that removes all contamination. We shall call τ_{opt} an optimal sweep schedule. The contiguous Line-Clear problem additionally requires τ_{opt} to be contiguous.*

4.1.1 Covering Sweep Lines with Sensors

Let us go through an example for the function R and assume that we have robots that can sense on a line of length r, e.g. r could be the diameter of a disk if the robot senses with an omnidirectional limited range sensor. Given that we may have noise in the control or localization of a robots we would like to have the sweep line covered with overlapping sensors. Let the parameter δ describe the extent of the overlap. The function R now becomes:

$$R(d) = \left\lceil \frac{d}{r - \delta} \right\rceil$$

If we have a sensor footprint that is a disk we can also consider using $c_{alt}(l)$ instead of $c(l)$ as a cost function. Fig. 4.2 shows how multiple robots cover a sweep line being oriented towards the left obstacle site, i.e. robots are placed uniformly on a sweep line starting from the left site and placed at distance $r - \delta$ from each other with the first robot having distance $\frac{r-\delta}{2}$ from the left site. As the distance between the obstacles grows another robot will have to be added.

Due to the left bias this is rather easy. If the robots can localize themselves and are synchronized they could plan their paths independently and then simply follow them as seen in fig. 4.2. Here errors in the synchronization of the paths can also be compensated by δ. But we shall also see approaches in which robots cannot localize themselves nor plan their paths ahead of time in Chapter 6 when presenting applications of Line-Clear.

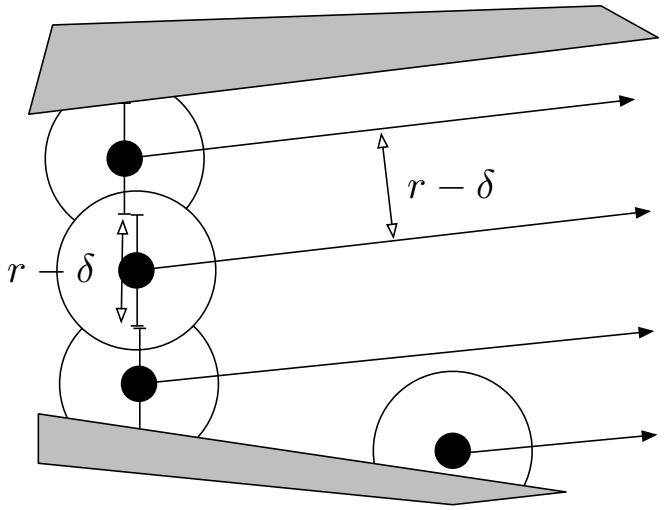


Figure 4.2: Multiple robots covering a sweep line between two obstacles. As the distance between the obstacles grows another robot is added at the appropriate location.

4.2 Sweep Schedules through Graph-Clear

In this section we explore the connection between Line-Clear, Voronoi Diagrams and Graph-Clear and show how to construct sweep schedules. Voronoi Diagrams are useful to detect narrow parts of an environment. Such narrow parts are intuitively good positions for blocks in Graph-Clear and hence they also play a role in extracting surveillance graphs to create Graph-Clear instances from a grid

map as we shall see in Chapter 5. We proceed by introducing the generalized definitions for Voronoi Diagrams from [CB95] and then show how these relate to Line-Clear and Graph-Clear.

4.2.1 Voronoi Diagrams

A rigorous formalization and generalization of Voronoi Diagrams to Generalized Voronoi Graphs (GVG) is found in [CB95] and we here shortly review its notation for two dimensions. Recall the definitions for the environment as given in Section 4.1. The following functions define a distance function towards obstacles which is the basis for the GVG:

$$\begin{aligned} d_i(x) &= min_{c_0 \in C_i} \|x - c_0\| \\ \nabla d_i(x) &= \frac{x - c_0}{\|x - c_0\|}. \end{aligned}$$

With these one can construct *equidistant surfaces* and *2-equidistance surjective surfaces* via respectively:

$$\begin{aligned} \mathcal{S}_{ij} &= \{x \in \mathbb{R}^2 : \ d_i(x) - d_j(x) = 0\} \\ \mathcal{SS}_{ij} &= \{x \in \mathcal{S}_{ij} : \ \nabla d_i(x) \neq \nabla d_j(x)\} \end{aligned}$$

Subsets of these then make up *2-equidistant faces* which are further restricted to *3-equidistance faces*:

$$\begin{aligned} \mathcal{F}_{ij} &= \{x \in \mathcal{SS}_{ij} : \ d_i(x) \leq d_k(x) \ \ \forall k \neq i, j\} \\ \mathcal{F}_{ijk} &= \mathcal{F}_{ij} \cap \mathcal{F}_{ik} \end{aligned}$$

Fig. 4.3 illustrates what these definitions.

The 2-equidistant faces and 3-equidistant faces become the edges and vertices of the GVG, respectively. More precisely the GVG for two dimension is $G_{gvg} :=$

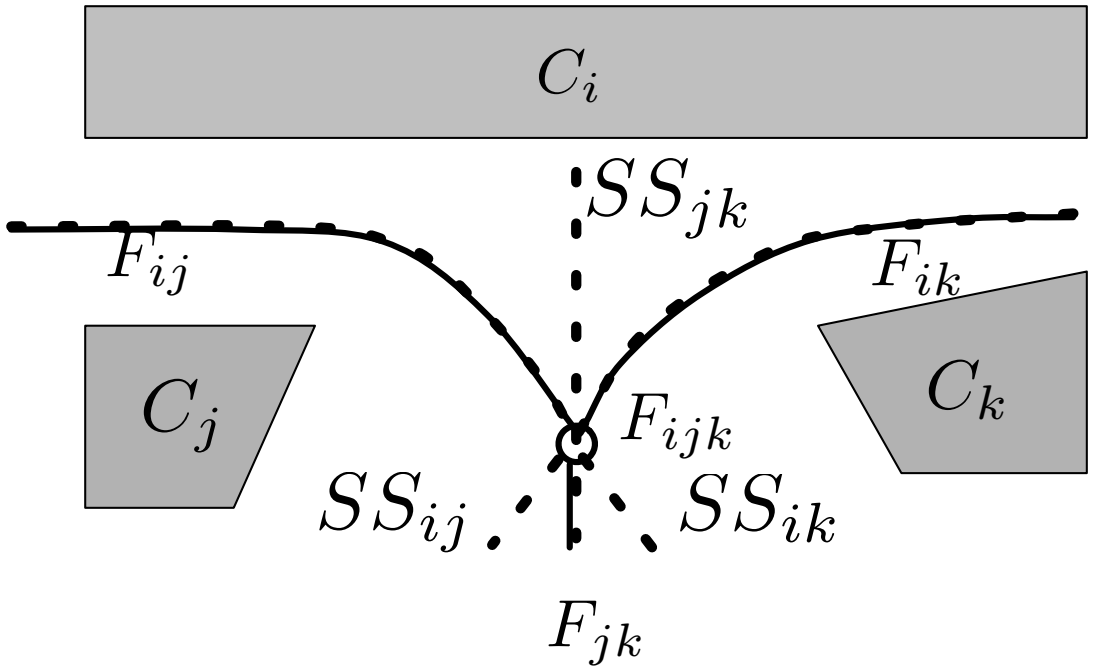


Figure 4.3: An illustration of the definitions of surjective surfaces and equidistant faces. The surjective surfaces are drawn with thin lines and the equidistant faces, a subset of the surjective surfaces, are drawn in thick and dashed lines.

$(\mathcal{F}^3, \mathcal{F}^2)$ where:

$$\mathcal{F}^2 = \bigcup_{i=1}^{n_o-1} \bigcup_{j=i+1}^{n_o} \mathcal{F}_{ij}$$

$$\mathcal{F}^3 = \bigcup_{i=1}^{n_o-2} \bigcup_{j=i+1}^{n_o-1} \bigcup_{k=j+1}^{n_o} \mathcal{F}_{ijk}$$

We will also make use of the function

$$q_i(x) := argmin_{c_0 \in C_i} \|x - c_0\|$$

which returns the closest point to x from obstacle C_i.

There are some complications with the GVG that arise when there exists a non-empty 4-equidistant surface. This corresponds to four points lying on the

circumference of a circle. To avoid this we shall also assume, as in [CB94], that no four obstacle points are co-circular. From a practical standpoint this assumption is not too demanding, since the case of four or more co-circular points can be dealt with by adding a small random perturbation, a technique often used in computational geometry (see for example [BKO00]). Another complication is that there may be multiple points in any $\mathcal{F}_{ijk}$. Each of these points is then a vertex in the GVG, which will cause complications in our construction in Section 4.2.2. We can, however, simply partition one of the obstacles into multiple parts s.t. each will be responsible for one vertex. This simplifies the notation without losing generality. From now on we can hence assume that there is at most one vertex for distinct C_i, C_j, C_k. Finally, it has to be noted that G_{gvg} is not strictly speaking a graph, unless one introduces an additional vertex at infinity. This is due to the fact that there are edges that go from a vertex to the intersection between two obstacles and continue unbounded, as seen in fig. 4.4. This will however not be a problem since we can ignore these edges in the construction presented in the following section. Note that this does not occur for all intersections since $\mathcal{SS}_{ij}$ has the additional constraint that $\nabla d_i(x) \neq \nabla d_j(x)$.

Since in two dimensions the GVG is a Voronoi Diagram, we shall use the term Voronoi Diagram throughout this dissertation. The definitions for surjective surfaces that the GVG uses will be useful later on. Another advantage of the GVG notation is a potential generalization to higher dimensions, although within the scope of this dissertation we shall remain focused on the two dimensional case. The Voronoi Diagram computed from a set of convex obstacles has useful properties to detect parts of the environment with small clearance and naturally provides a topological map. This fact has often been exploited for robot navigation [Thr98]. Also Graph-Clear benefits from surveillance graphs with small edge weights and hence areas with small clearance provide good candidates

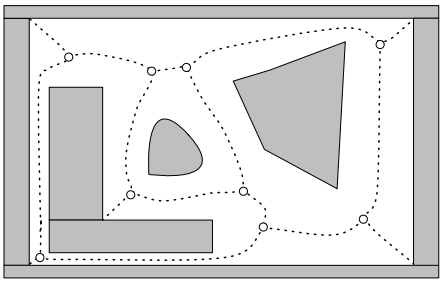


Figure 4.4: The GVG vertices and edges in the environment marked as circles and dashed lines, respectively. Note that some edges are continuing on the intersection of two obstacle boundaries, but not all intersections lead to an edge as seen in the corridor to the left. To form a proper graph an additional vertex at infinity could be introduced and connected to these edges.

for borders between regions leading to edges with small weights. For this reason we will revisit Voronoi Diagrams in Chapter 5 for the extraction of surveillance graphs from robot maps.

In practice, when considering a map given by polygonal obstacles it is often useful to consider each line segment of the polygons and each of their endpoints to be a separate obstacle. This automatically satisfies convexity without having to partition obstacles into convex sets, but it does increase the size of the Voronoi Diagram. Figure 4.6 shows a Voronoi Diagram that would be generated by considering each segment and its endpoints as separate obstacles.

4.2.2 Constructing Sweep Schedules from Voronoi Diagrams

We now outline a construction that creates a surveillance graph from a Voronoi Diagram and then associates moving sweep lines to its vertices and edges in order to compute their weights. A strategy on this graph can then be converted to a sweep schedule. The main idea is easily illustrated in fig. 4.5 in which a moving

sweep line clears the environment by moving between two obstacles and then splits into two new moving sweep lines at a new obstacle. The purpose of the surveillance graph is to capture the cost of such movement and subsequent splits.

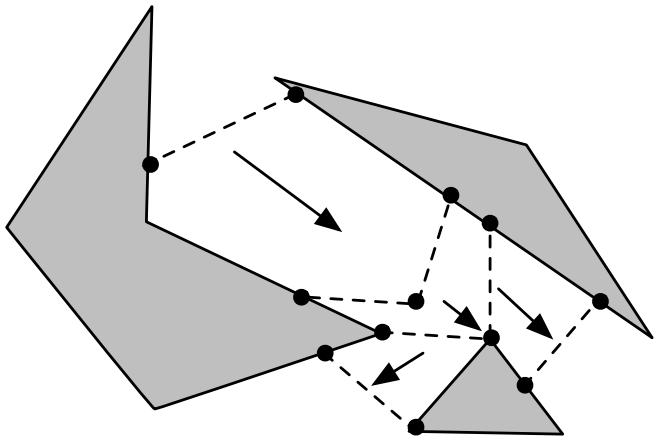


Figure 4.5: Illustration of the concept of moving and splitting sweep lines. The arrows indicate the direction of movement of the sweep line on the left side until it splits into two sweep lines which continue independently.

Given an environment $\mathcal{E}$ we first compute the Voronoi Diagram given by $(\mathcal{F}^2, \mathcal{F}^3)$. For each vertex in $\mathcal{F}^3$ which is in free space $interior(\mathcal{E})$ we create a vertex for a surveillance graph $G_{sg} = (V, E, w)$ and add all edges from $\mathcal{F}^2$ between these vertices that are also entirely in free space to E. Recall that we can assume that any vertex of $\mathcal{F}^3$ has degree 3 and hence no vertex of G_{sg} will have degree larger than 3. The main part of the construction is now to associate weights and moving sweep lines to every vertex and edge of G_{sg}. We shall use the modified variant of Graph-Clear from Section 3.9 to allow directional vertex weights. Figure 4.6 shows an example of G_{sg} and a few lines we will associate to its vertices. With this figure in mind let us make the construction more precise.

Every vertex $v \in V$ has exactly three defining obstacles, which we shall call sites. Write C_i, C_j and C_k for the sites of v. Notice that v hence originated from

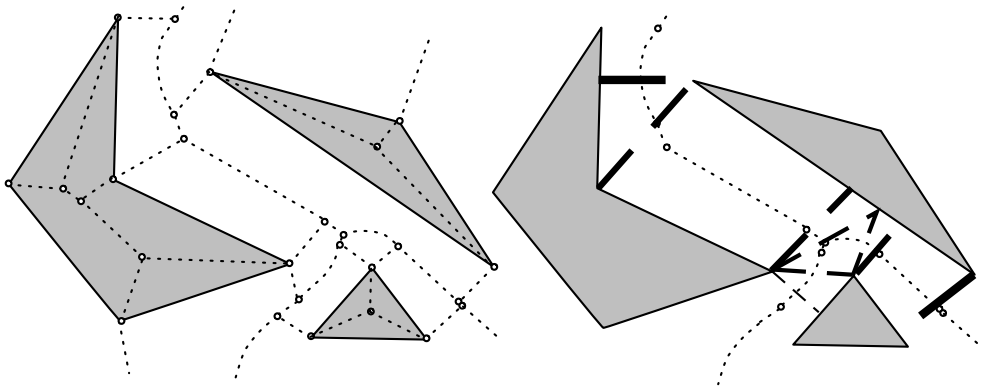


Figure 4.6: Left: A Voronoi Diagram resulting from line segments of multiple polygonal obstacles by considering each open segment and their endpoints as independent obstacles. Note vertices and edges inside the polygons are also drawn and result from the fact that the line segments are considered obstacles for the purpose of construction the Voronoi Diagram. Right: Conversion of the Voronoi Diagram into a surveillance graph. Dashed lines indicate lines that are associated to vertices and edges and represent blocks and sweeps. The movement of lines is represented by their thickness, i.e. thin lines move towards thicker lines.

$\mathcal{F}_{ijk}$ and with a slight abuse of notation we can write $v \in \mathcal{F}_{ijk}$. We now create a moving sweep line that we associate to v to obtain its weights. Note that we need a weight for every direction that the vertex can be entered from, i.e. for each of its edges. For each pair of its sites there can be an edge for v, depending on whether the neighboring vertex from $\mathcal{F}^3$ is in free space. Let e_{ij} be one of the edges of v, namely the one that originated from $\mathcal{F}_{ij}$ and is hence between C_i and C_j. Consider the scenario in which we approach v with a sweep line spanned between C_i and C_j written l_{ij}. Call $l_{ij}(t_{block})$ the sweep line at blocking position for e_{ij} at some time t_{block}. Now, sweeping v from direction e_{ij} means to move $l_{ij}(t)$ of e_{ij} onto the third site C_k. Fig. 4.7 illustrates this graphically. From this point onwards the sweep line can split into two sweep lines between C_i, C_k and

C_j, C_k. The cost of this split, i.e. the joint cost of the two sweep lines right after splitting becomes the weight of v.

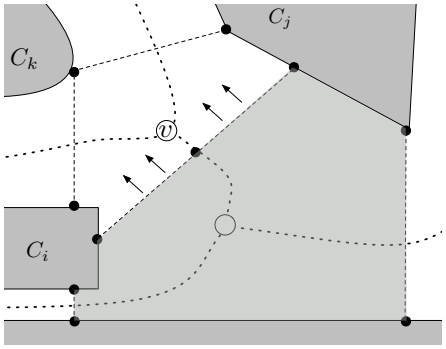


Figure 4.7: A sweep line moving along an edge e_{ij} and crossing vertex v. The grey area is the cleared part $\mathcal{R}(t)$ bounded by obstacles and sweep lines.

To determine the cost of the split we need to find the point on C_k that allows us to split the sweep line at lowest cost. For this purpose we define the split point $p \in C_k$ as follows. First, we need to define a set of points on C_k which lead to valid sweep lines from C_i and C_j to C_k. Given an $x \in \delta C_k$ write $l_{i,k}(x) = [q_i(x), x]$ for a sweep line between C_i and C_k and $l_{j,k}(x) = [q_j(x), x]$ for a sweep line between C_j and C_k.

$$X := \{\ x \in \delta C_k \mid \exists l_1, l_2 \in \mathcal{S},\ l_1 = l_{i,k}(x), l_2 = l_{j,k}(x)\}. \tag{4.1}$$

The split point p is now any point satisfying:

$$p = argmin_{x \in X}\{c(l_{i,k}(x)) + c(l_{j,k}(x))\}. \tag{4.2}$$

Now that we have p we can describe the moving sweep line $l_{ij}(t)$ in more detail, in particular how it moves from its initial blocking position towards its split on C_k. Formally, let $l_{ij} : \mathbb{R}^+ \to S(3)$ so that it has three points[4]. Write

[4]At the blocking position we only need two points, the two endpoints on the obstacles, but

$l_{ij}(t_{block})$ for the initial blocking sweep line at time t_{block} and $l_{ij}(t_{split})$ for the final split sweep line. It is convenient to describe the movement of l_{ij} in reverse, i.e. from t_{split} to t_{block}. Fig. 4.8 illustrates the construction that follows.

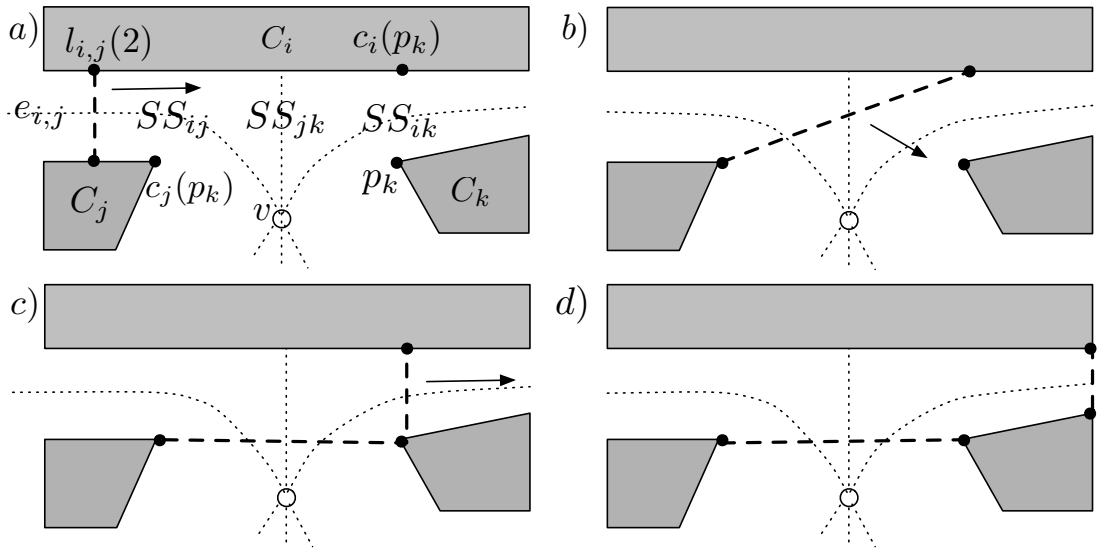


Figure 4.8: Illustrating the forward movement of a sweep line across a vertex v. Part a) shows l_{ij} at time t_{block}, part b) at time t_{swap}, part c) at time t_{split} and part d) shows the two new moving sweep lines l_{ik} and l_{jk} at time t_{end}.

We set $l_{ij}(t_{split}) := [q_i(p), p, q_j(p)]$. Clearly, $c(l_{ij}(t_{split})) \leq c(l_{i,k}(p)) + c(l_{j,k}(p))$. We then move backwards to time $t_{swap} < t_{split}$ and sweep line $l_{ij}(t_{swap}) = [q_i(p), x_{mid}, q_j(p)]$ where $x_{mid} \in [q_i(p), q_j(p)]$ is chosen s.t. $c([q_i(p), x_{mid}]) \leq c([q_i(p), p])$ and $c([x_{mid}, q_j(p)]) \leq c([p, q_j(p)])$ which is clearly possible due to the triangle inequality (see fig. 4.8). At this point we could remove l_{ij} and instead add an equivalent sweep line that has only two points, since x_{mid} is on the straight line segment between the two endpoints. But to simplify the notation we

it is more convenient for the notation to already require three points since at the split we need at least three points. Note that a change in the number of points of a moving sweep lines is formally always a removal of the old sweep line and the addition of a new moving sweep line to τ.

will instead choose x_{mid} so that $c(l_{ij}) = R(|l_{ij}|)$, in other words so that the cost is identical to the cost of the sweep line with only two endpoints. Note that this consideration is only necessary when using the cost function $c(l)$ and not for $c_{alt}(l)$ since additional midpoints in a sweep line cannot change the cost for c_{alt}. This slight technicality allows us to continue to use the same notation for the moving moving sweep line l_{ij} and move it further backwards towards $t_{block} < t_{swap}$. This last movement to the blocking position to moves from $l_{ij}(t_{swap})$ towards the direction into which the distance between the obstacles is shrinking until the minimum distance is reached. Here it is worth noting that the function $d_i(x) = d_j(x)$ with $x \in SS_{ij}$ has exactly one minimum [Sie99]. This does not, however, mean that $l_{ij}(t_{block})$ finally reaches that minimum as seen in fig. 4.9, but it does mean that it eventually stops at the position with lowest possible cost. To complete the construction we now do the same with the sweep lines after the split. At time t_{split} remove $l_{ij}(t_{split})$ from τ and add two new sweep lines $l_{ik}(t_{split}) = [q_i(p), y_{mid}, p]$ and $l_{jk}(t_{split}) = [q_i(p), z_{mid}, p]$ where y_{mid} and z_{mid} are chosen just as x_{mid} before to emulate a sweep line with only two endpoints. Both lines l_{jk} and l_{ik} continue to move into the direction of decreasing length until a minimum is reached, just as l_{ij} moved backwards. The position reached is then the blocking position for edges e_{jk} and e_{ik}. Let us denote the time at which both sweep lines reach this point by $t_{end} > t_{split}$.

Finally, the movement of the sweep line l_{ij} from t_{block} to t_{split} and subsequently the movement of l_{ik} and l_{jk} to t_{end} is a sweep for vertex v from direction e_{ij}. The maximum cost occurring during this process occurs right after the split and hence $w(v) = c(l_{ik}(t_{split})) + c(l_{jk}(t_{split}))$. The weights for the edges e_{ij}, e_{ik} and e_{jk} simply become the cost of the respective sweep line at the blocking position.

Now let us expand our perspective from v and e_{ij} to the entire graph. Due to

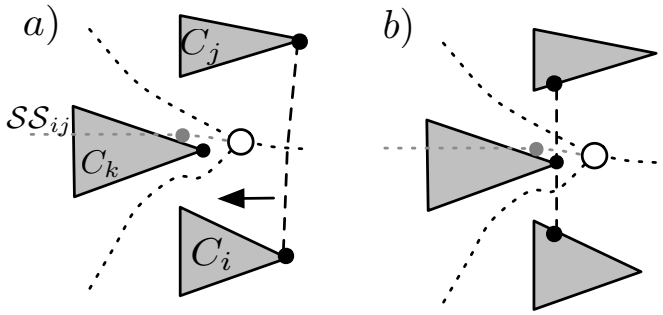


Figure 4.9: An example in which the minimum of d_i on $\mathcal{SS}_{ij}$, marked as a grey dot on a grey dashed line in a), does not lead to a valid sweep line. The blocking position is then earlier on $\mathcal{SS}_{ij}$ as seen in b).

the fact that we only have one minimum of $d_i(x)$ on $x \in \mathcal{SS}_{ij}$ for all i, j it is clear that we can reach the blocking position of l_{ik} and l_{jk} from which the sweep of the neighboring vertex starts without incurring larger cost. Hence we can apply this procedure to get weights for every vertex v for the sweeps from an incoming edge and we can concatenate these neighboring vertex sweeps. After a vertex is swept, the new sweep lines after the split simply become the blocking positions of the edges for the neighboring vertices. So for every vertex with degree three we now have weights $w(v, e_1)$, $w(v, e_2)$,$w(v, e_3)$ for all its three edges e_1, e_2, e_3. Note that we have not yet clarified how to arrive at the blocking position for the first vertex when concatenating these sweeps.

Let us now consider vertices with degree two or one. Note that the originating vertices from the Voronoi Diagram, i.e. in $\mathcal{F}^3$, all have degree 3. Hence the missing edges from for degree two and one vertices in G_{sg} are the result of either C_i, C_j or C_k being adjacent to each other such that the edge in $\mathcal{F}^2$ for the originating vertex goes towards the intersection of two obstacles and possibly towards the infinity vertex. This observation simplifies the consideration of vertices with degree two and one. One can simply think of them as degree three vertices by

adding virtual edges with weight zero to G_{sg}. These virtual edges coincide with edges in $\mathcal{F}^3$ that are on going towards intersections of adjacent obstacles. We cannot directly add these edges to G_{sg} because we only allowed non-zero and positive integer weights. This simplifies the treatise significantly if we consider all vertices principally of degree three, since the number of cases to discuss is reduced significantly. Computing the weight $w(v, e_1)$ for a degree two vertex v with edges e_1, e_2 in G_{sg} is then simply a split from the blocking position of e_1 into two sweep lines, one with length 0 and one that moves to the blocking position for e_2. Computing $w(v, e_2)$ is symmetric. For a vertex v with only one edge e computing $w(v, e_1)$ is then again just a split into two sweep lines that approach length 0.

So now we can compute weights for all vertices $v \in V$ for every direction. Note that for the modified variant of Graph-Clear from Section 3.9 these weights already suffice if we consider only trees. Furthermore, we have additional sweeps when entering a degree two or one vertex from a virtual edge which is essentially a sweep starting at a sweep line with length 0. Such a sweep from a virtual edge is only possible if the vertex is the first to be cleared. So let us now clarify the process of clearing the first vertex in G_{sg}, which obviously has no edge from G_{sg} to enter from (recall eq. 3.43 in Section 3.9 where this problem is shortly mentioned). This simply works by setting up two sweep lines at one edge (including virtual edges with length zero sweep lines) at the blocking position and then moving of them to sweep the vertex. There are three choices of edges to start from and we simply chose the one that has the lowest overall cost. The cost is simply $w(v, e_i) + w(e_i)$ for every edge e_i of v, i.e. cost of the blocking position for the chosen edge and the sweep cost from that direction. This overall cost then becomes the weight $w(v)$, i.e. the weight for sweeping v without a given direction.

To consider graphs or non-contiguous sweep schedules, we also have to describe a sweep procedure for the case when a vertex is cleared starting from two edges and three edges. These cases do not occur in a tree that is cleared contiguously. But fortunately, they are analogue to the case when one edge is blocked before the sweep. Fig. 4.10 shows these different cases that are best illustrated in a figure. To see the connection between them let us look at how to sweep a vertex when starting at edges e_{ik} and e_{jk} and sweeping v to finally arrive at a blocking position on e_{ij}. For this we can use the same sweep lines constructed previously, but moving them backwards. Hence the cost for this procedure is the same as given by $w(v, e_{ij})$ and this weight represents the cost of two sweeps. Similarly, the cost of clearing by starting from all three edges blocked is identical to clearing with no edge blocked. Therefore, the weight function $w(v, e)$ suffices to represent all costs, even when considering cyclic graphs.

Fig. 4.11 shows an example of a surveillance graph created from a Voronoi Diagram and blocking positions for edges. Let us now shortly describe how to construct a sweeping schedule τ from a strategy $\mathcal{S}$ represented by a sequence of vertices written $v_1, \ldots, v_n$. Start with two identical sweep lines l_1, l_2 at one of the edges in the Voronoi Diagram of v_1, namely the one with 0 or lowest cost. Add l_1, l_2 to τ at time $t = 0$. If $degree(v) = 3$ there is no edge in the Voronoi Diagram with a sweep line with cost 0 and all edges for the same vertex in $v_{voronoi} \in \mathcal{F}^3$ are also edges in G_{sg}. If $degree(v) < 3$, then at least one of the edges in $\mathcal{F}^2$ for $v_{voronoi}$ is not in G_{sg} and hence has a blocking sweep line with length and cost 0. Now, $l_1(t)$ remains at its blocking position while we move $l_2(t)$ from $t = 0$ to $t = t_{split}$ following the sweeping procedure for vertex v coming from the respective edge that we chose previously. Then we remove l_2 from τ at time t_{split} and add the new sweep lines resulting from the split to τ and move them towards their blocking position at time t_{end}. Note that there is at least one such new blocking

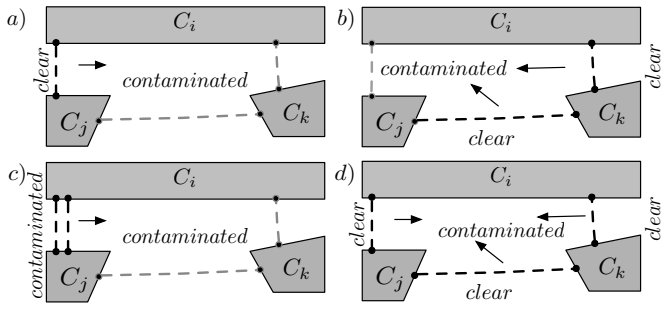


Figure 4.10: Four cases for a vertex with degree three. Current sweep lines are black while future sweep lines that are to be reached are grey. Contaminated and cleared sides of current sweep lines are marked. In a) one sweep line splits into two sweep lines. In b) two sweep lines merge into one sweep line, the converse of a). In c) all points are still contaminated and two sweep lines are established to being clearing. In d) all points outside the figure are cleared and all sweep lines will disappear, the converse of c). Part a) represents three cases, one for each choice of direction for the current sweep line, i.e. either starting between C_i and C_j as seen or between C_i,C_k or C_j,C_k. Similarly, part b) also represents three cases. For c) and d) there is only one choice of directions, leading to overall eight possible sweeps for the Voronoi vertex associated to C_i,C_j,C_k represented by four weights, since each case has an associated inverse at identical cost.

line. Then we choose one of these blocking lines, namely the one leading to vertex v_2 and start sweeping vertex v_2 as described before. Continuing this leads to an sweep schedule τ that clears the entire environment. It is slightly more difficult to assess whether such a sweep schedule can be optimal and hence a solution to Line-Clear. This we will discuss in the next subsection. In Chapters 6 and 5 we shall revisit this procedure and see how it works in practice and compares to other approaches.

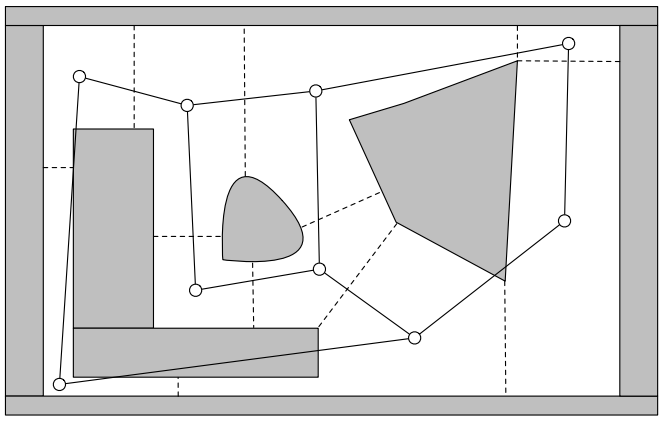


Figure 4.11: The blocking positions of sweep lines in the environment are marked as dashed lines each creating an edge between two vertices which now correspond to a region as partitioned by the blocks. The Voronoi Diagram of the environment is presented in fig. 4.4.

4.2.3 Optimality Considerations

The construction of an sweep schedule from a Voronoi Diagram via a strategy on its dual surveillance graph gives a convenient way to construct sweep schedules. In colloquial terms the strategy computed on the surveillance graph tells us in which sequence we should consider the obstacles C_i to become an endpoint of a sweep line. The Voronoi Diagram in turn restricts the number of obstacles we have to consider locally to those that are sites for a common vertex. This restriction of which obstacles to consider for a split, which is the process by which new obstacles become endpoints of sweep lines, is significant for the optimality considerations. In fact, it leads to the construction of a counterexample in which we can show that this restriction of choice leads to a suboptimal sweep schedule.

Consider the simple star-like environment as seen on the left side in fig. 4.12. Its six-way intersection would lead to a Voronoi vertex with degree six, but through a slight perturbation this is resolved into four vertices, each of de-

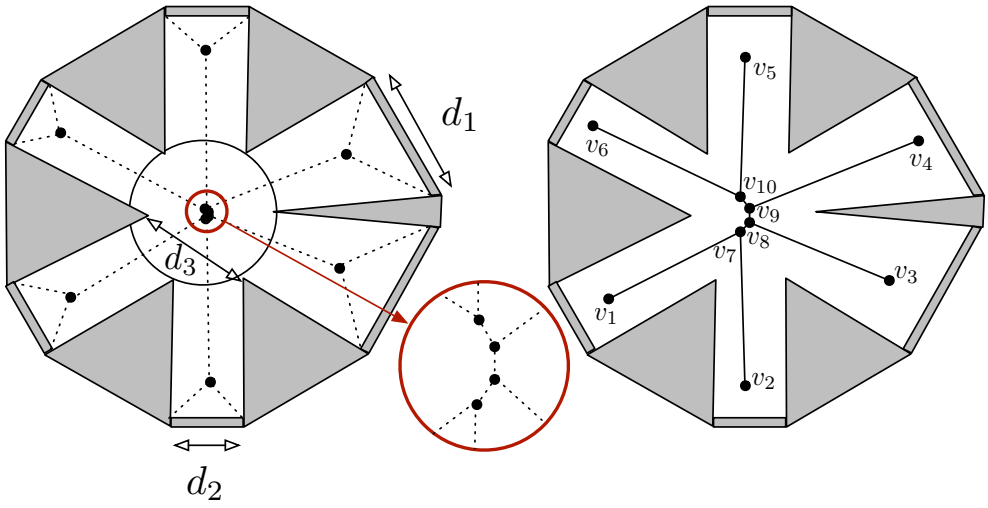


Figure 4.12: A six-way intersection constructed around a circle with diameter d and larger corridors given by parameter d_1. Values for d_3 and d_2 follow from the circles diameter. All center obstacles are aligned around the circle with the exception of the leftmost obstacle which is moved towards the center by $\epsilon > 0$.

gree three. All six obstacles in the center are placed around a circle with diameter d shown in fig. 4.12 up to an arbitrary small perturbation by some $\epsilon > 0$. In fig. 4.12 this is done by moving the leftmost obstacle by ϵ, closer to the center of the circle than other obstacles. We shall now show that for an appropriate choice of d and d_1 this leads to a surveillance graph whose strategies produce sweep schedules that are not optimal. Each corridor of the six-way intersection in the example is either of constant width d_2 or grows in width from d_2 to d_1. A constant width corridor can be swept at the same cost as its block, while for a growing corridor the cost increases as it becomes wider. If the growing corridors are very costly relative to the blocks towards corridors, i.e. d_1 is much larger than d_2, more precisely $d_1 > d_2 + d$, then an optimal sweep schedule will have to clear these growing corridors when no other sweep lines are at their blocking positions, i.e at the beginning and end of the sweep schedule. This is shown in

fig. 4.13 as the optimal sweep schedule which can clear the environment with cost d_1 given that we choose $d_1 > d_2 + d$. Let us now focus on the surveillance graph that results from the Voronoi Diagram. Consider a strategy on the graph that sweeps both of the growing corridors, represented vertex v_3 and v_4, at a time when no other blocks are active in the environment, i.e. at the beginning and at the end of the strategy. From the surveillance graph it follows that such a strategy necessarily has a split on v_8 (or v_9 due to the symmetry) right after clearing v_3 (or v_4). This split has a cost of $d - \epsilon + d_3$ shown in fig. 4.14. If we now choose d and d_1 so that we have $d - \epsilon + d_3 > d_1 > d_2 + d$ the sweep schedule from any strategy will never be able to clear the environment at cost d_1 or lower and can hence not be optimal. This choice is clearly possible by simply choosing any d which automatically determines d_3 and d_2 since they are dependent on the circle and then choosing $\epsilon < (d_3 - d_2)/2$ and $d_1 = d_2 + d + \epsilon$.

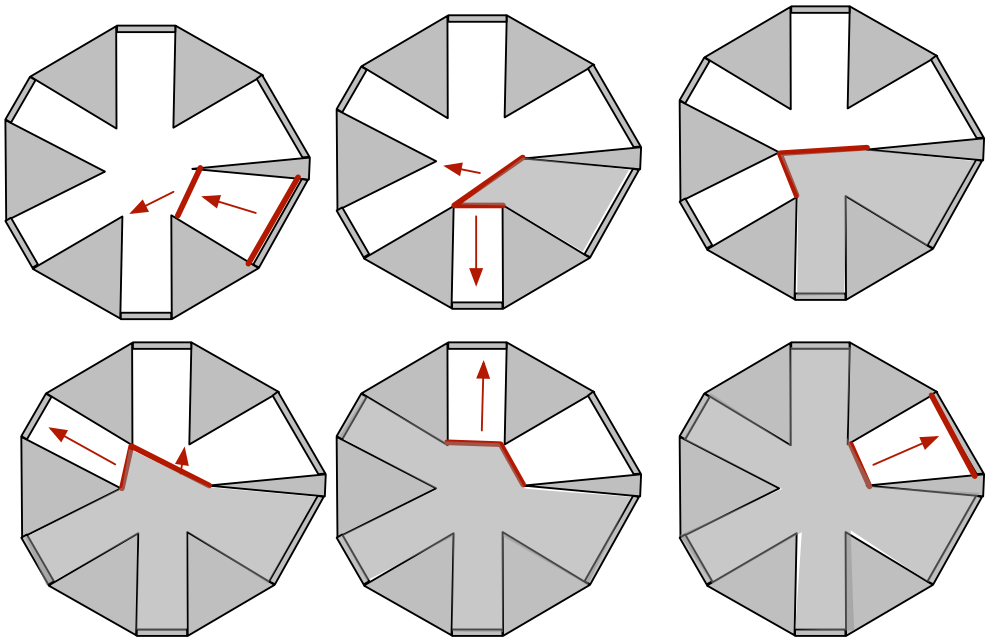


Figure 4.13: An illustration of an optimal sweep schedule for the environment from fig. 4.12 given that $d_1 > d_2 + d$.

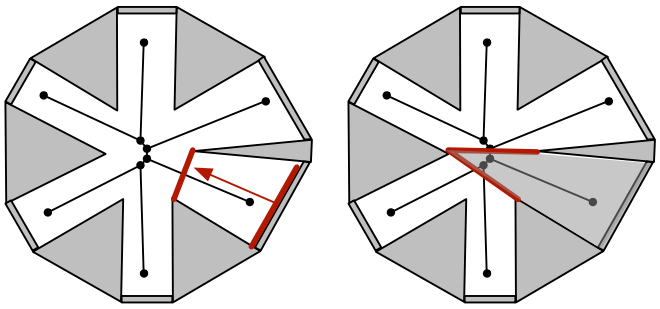


Figure 4.14: An illustration of the beginning of a sweep schedule created from the Voronoi Diagram of the environment from fig. 4.12 starting at v_8. Given that v_8 is the starting vertex, the next split is necessarily at v_3 as shown in the figure.

From the counterexample it becomes clear that the reduction for the choices for obstacles onto which to split is the reason for losing optimality. This motivates the investigations in the following sections in which we seek to represent all choices and their respective cost. It may well be that under certain conditions the restriction of choices through the Voronoi Diagram can lead to optimal solutions. For example when no fourth obstacle is close to a Voronoi vertex $v \in \mathcal{F}_{ijk}$ it is unlikely that any other split than the one onto C_K should be considered. Further work into this direction may lead to strict criteria, but to attempt this we first need to understand the cost structure of choosing obstacles for splits.

4.3 Reduction to a Combinatorial Problem

Given that the construction of sweep schedules from Voronoi Diagrams is not optimal we now investigate the construction of optimal progressive sweep schedules at least for simply-connected environments. Ideally this would be done in polynomial time, although the question whether this is possible still remains open. The previous construction would not have been able to compute optimal strate-

gies on graphs in polynomial time since Graph-Clear is NP-hard on graphs. As a consequence, it would also not have been able to tackle multiply-connected environments in polynomial time. In this section we are explicitly focusing on progressive sweep schedules for this section and thereby rule out recontamination. It is not yet proven that recontamination does not matter for Line-Clear. But considering its relationship with Graph-Clear which we shall continue to elaborate on throughout this section make this a reasonable conjecture. In what follows we develop an alternative perspective for the construction of sweep schedules that does not rely on the Voronoi Diagram for choosing which obstacle to initiate a split for.

For this we look at the boundary of $\mathcal{R}(t)$ and shall show that in simply connected environments Line-Clear is in fact a combinatorial problem. So let us restrict the environment to be simply-connected. For convenience we shall assume that the indices of all obstacles are ordered clockwise on the boundary of $\mathcal{E}$ and that in a full traversal of this boundary no index appears twice. Note that this does not impose any further restrictions on the environment but, as we shall see, simplifies the notation. The basis for the combinatorial perspective is to describe $\mathcal{R}(t)$ by means of a sequence of obstacle indices defined as follows:

Definition 36 (Boundary State) *Given a sweep schedule $\tau(t)$ at time t and a resulting $\mathcal{R}(t)$ let the state of its boundary be the sequence of obstacle indices on the boundary of $\mathcal{R}(t)$ starting at the lowest index in clockwise order. Write $B(t)$ for this sequence*[5].

Since $\mathcal{R}(t)$ is connected this is well defined. Fig. 4.15 shows a simply con-

[5] *Formally, a sequence $B(t)$ is a function $B(t) : N \to \{1, \ldots, n_o\}$ where $N = \{1, \ldots, b\} \subset \mathbb{N}$ if $B(t)$ has b elements. Note that $B(t)$ is not simply an ordered set since the same index can appear multiple times. But for notational convenience we shall still write $B(t) = \{B(t)(1), B(t)(2), \ldots, B(t)(b)\}$.*

nected environment with a few examples for $\mathcal{R}(t)$. It immediately follows that at the end of a sweep schedule we have $B(t) = \{1, \ldots, n_o\}$, i.e. a full traversal of obstacles on the boundary of $\mathcal{E}$.

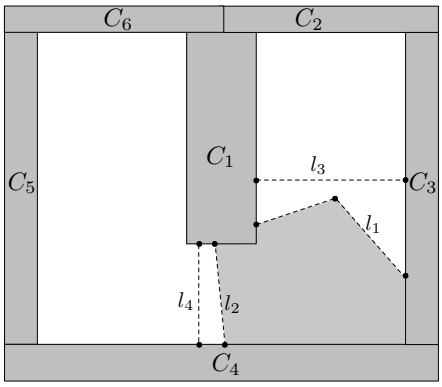


Figure 4.15: A simply connected environment with two sets of two sweep lines $\{l_1, l_2\}$ and $\{l_3, l_4\}$. Sweep lines l_1 and l_2 together with parts of obstacles C_1, C_3 and C_4 form the boundary of a cleared region for a sweep schedule. This boundary can be traversed by following C_1, C_3, C_4, C_1 and hence $B(t) = \{1, 3, 4\}$. The same traversal applies to a possible sweep schedule represented by $\{l_3, l_4\}$ and these represent the set of sweep lines with lowest cost that have this traversal sequence.

It is now straightforward to describe the evolution of $B(t)$ in terms of the obstacle indices that are added and removed at the appropriate location in $B(t)$ as τ proceeds to clear the entire environment until finally $B(t) = \{1, \ldots, n_o\}$, for some time t at the end of the execution of the schedule. In fact, we can show that obstacle indices are only added and never removed from $B(t)$.

Lemma 9 (Obstacle indices added exactly once) *For any progressive sweep schedule τ each obstacle index is added exactly once to $B(t)$.*

Proof: Let us assume that during the execution of a sweep schedule $\tau(t)$ we have an obstacle index $o \in \{1, \ldots, n_o\}$ that is added to $B(t)$ twice. First note,

that the removal of an obstacle index from $B(t)$ implies recontamination if it is unique in $B(t)$. Hence, at the time t_{add} when o is added to $B(t)$ for the second time we already have o in $B(t)$ and it occurs twice in $B(t)$. Now we have at least one obstacle index j s.t. $B(t) = \{\ldots, o, \ldots, j, \ldots, o, \ldots\}$. Fig. 4.16 illustrates this. Therefore, while traversing the boundary of $\mathcal{R}(t)$ from the first occurrence of o to the second occurrence of o we move along at least two sweep lines. The first sweep line encountered on this traversal necessarily starts at some point $x_1 \in C_o$ while the last sweep line encountered before reaching C_o again ends at some point $x_2 \in C_i$. Clearly, the segment of C_o between points x_1 and x_2 is contaminated. Hence the first sweep line has $\mathcal{R}(t)$ to its right side, with respect to to the direction of the traversal, while the last sweep line has it to its right side. This implies that $\mathcal{R}(t)$ is not connected, violating contiguity of τ. Hence no index can be added more than once. □

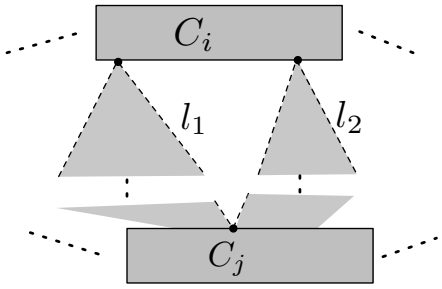


Figure 4.16: Adding an obstacle index twice to $B(t)$ at different times violates contiguity of the cleared part.

As a consequence of lemma 9 the evolution of $B(t)$ is simply the addition of all obstacle indices in some sequence. Furthermore, all indices in $B(t)$ are ordered, i.e. if $B = \{\ldots, i, j, \ldots\}$ then $i < j$. Suppose that we had a procedure that can construct the lowest cost sweep schedule from a sequence of obstacle indices $o_1, o_2, \ldots, o_{n_o}$. Then we can find the optimal sweep schedule by considering all

sequences of which we have $n_o!$ many. Obviously, we want to further reduce the number of sequences we have to consider.

Our goal is two-fold. First we seek to find a procedure that constructs the lowest cost sweep given a sequence of obstacle indices, and second, in the next section, we investigate how to compute a sequence of obstacle indices that leads to an optimal sweep schedule.

4.3.0.1 Constructing Minimal Cost Sweep Schedules

Let us begin with the construction of a minimum cost sweep schedule for a given sequence of indices. This construction is the basis of the reduction of the geometric Line-Clear problem to a combinatorial problem. We write $o_1, o_2, \ldots, o_{n_o}$ for the sequence in which obstacle indices are added to $B(t)$ and t_i when the i-th obstacle index is added to B. We may for example have $B(t_1) = \{o_1\} = \{4\}, B(t_2) = \{o_2, o_1\} = \{2, 4\}$ and $B(t_{n_o}) = \{o_5, o_2, o_3, o_1, \ldots, o_4\} = \{1, 2, 3, 4 \cdots, n_o\}$. For each $i \in \{1, \ldots, n_o\}$, at t_i when o_i is added we can compute a lower bound on the cost for any sweep schedule that satisfies the sequence $o_1, \ldots, o_{n_s}$.

First notice that the addition of index o_i with $i > 2$ always involves moving the sweep line between the two indices $i_l, i_r \in B(t_i)$ where $i_l = \max_{o \in B(t_i)}\{o < o_i\}$ is to the left of o_i and $i_r = \min_{o \in B(t_i)}\{o > o_i\}$ is to the right of o_i. Since the boundary is circular we need to clarify that if $o_i \geq o, \forall o \in B(t_i)$, then $i_r = \min_{o \in B(t_i)}\{o\}$ and similarly if $o_i \leq o, \forall o \in B(t_i)$, then $i_l = \max_{o \in B(t_i)}\{o\}$. The cost of going from $B(t_{i-1})$ to $B(t_i)$ can be determined by the cost of extending the sweep line between C_{i_r} and C_{i_l} towards C_{o_i} and splitting it. It boils down to finding the point on $p \in C_{o_i}$ so that the joint cost of a sweep line from p to C_{i_l} and a sweep line from p to C_{o_i} is lowest. This is similar to the construction in Section 4.2.2 with the only exception that we consider splits to any obstacle instead of only

one close obstacle. The definition of the minimum split point p ensures that the cost of adding o_i to B is in fact minimal. Fig. 4.17 shows examples of sweep lines starting at some point on C_{o_i}.

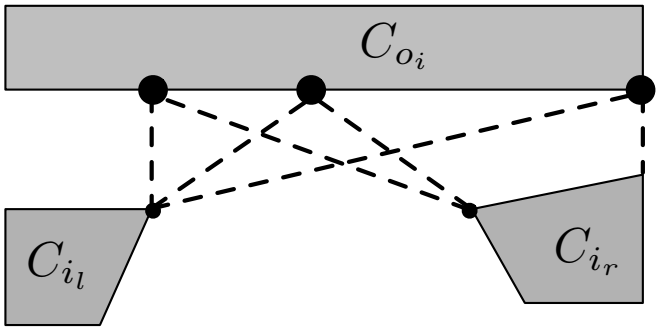


Figure 4.17: Three points on C_{o_i} and the sweep lines they form to C_{i_l} and C_{i_r}.

Once such a point p on C_{o_i} is found the sweep line between C_{i_l} and C_{i_r} is moved towards p and then splits into two sweep lines between C_{i_l}, C_{o_i} and C_{i_r}, C_{o_i}. These then continue to move until they reach a local minimum for their cost. Fig. 4.18 shows examples of obstacle index sequences and how they relate to sweep schedules. We are going to discuss some details on the split point p in Section 4.4.1. For now, we assume access to an oracle that provides p.

Given p we can determine the cost of adding o_i to $B(t)$. It is simply the cost of all sweep lines that maintain $B(t_{i-1})$, before addition of o_i, and the additional cost of moving the sweep line between C_{i_l} and C_{i_r} towards p and splitting it. Just as in Section 4.2.2 the maximum cost is occurs right after the split. Once the sweep line is split, the cost of maintaining $B(t_i)$ can be computed by moving the two new sweep lines towards the next minimum length between its two obstacle endpoints (recall the the movement towards blocking positions in Section 4.2.2).

Initially the cost of maintaining $B(t_1)$ is zero which bootstraps the recursion. Adding o_1 costs 0 by adding a sweep line starting and ending at a point on C_{o_1}.

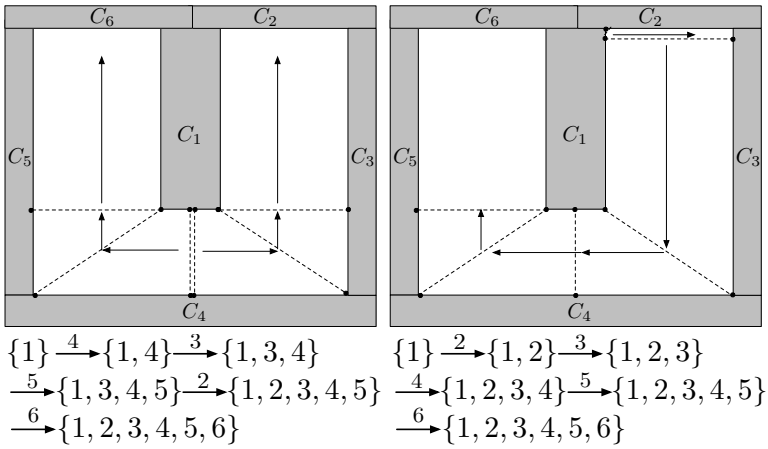


Figure 4.18: Example of two sweep schedules. The sweep on the left is suboptimal while the one on the right is the optimal solution. On the bottom of each side the evolution of the set $B(t)$ is shown.

Now, adding o_2 can cost 0 if C_{o_1} and C_{o_2} are adjacent or it will cost as much as the split from the first zero length sweep line from C_{o_1} to C_{o_2} and so on.

4.4 Finding Optimal Sweep Schedules for Simply-Connected Environments

We now turn to the problem of determining a sequence $o_1, \ldots, o_{n_o}$ that will correspond to an optimal sweep schedule. To gain an understanding of the problem we first go through a representation of the available choices. At first one may choose any of the obstacle indices for o_1. In the example in fig. 4.21 that will be discussed later on we set $o_1 = 1$. From there we can build a tree-like structure that represents all further choices, i.e. the remaining $\{1, \ldots, n_o\} \setminus \{o_1\}$ indices which can be chosen for o_2. Every choice for o_2 will lead to a separation of the remaining choices $\{1, \ldots, n_o\} \setminus \{o_1, o_2\}$ into two sides. On the left are all indices $o \in L_{o_1,o_2}$ s.t. $o_1 < o < o_2$ and right are all indices $o \in R_{o_1,o_2}$ s.t. $o_2 < o < o_1$.

Note that this is again a circular ordering, i.e. $5 < o < 2$ can mean indices $o = 6, 7, 1$. The left and right choices correspond to two disconnected contaminated parts in $\mathcal{E}$ that are separated by the cleared part. Each further choice, first continuing with o_3, separates either the left or the right indices further. Let us write $T_k^i = \{i, i+1, \ldots, i+k-1\}$ for a set of k consecutive obstacle indices starting at i and call it a choice set. To represent the circular order we identify indices $i + n_o$ with i. We also write $T_0^i = T_0 = \emptyset$. We can use this notation to represent the remaining choices on left and right, since all indices on each side are consecutive. Fig. 4.19 shows this recursion. Note that the purpose of every set of choices T_k^i is to represent a connected contaminated area in $\mathcal{E}$ bounded by exactly one sweep line and obstacles $C_{i-1}, C_i, \ldots, C_{i+k}$. This is shown in fig. 4.20.

After choosing the first index o_1 we can write all further choices as $T_{n_o-1}^{o_1+1}$ which contains all remaining indices $o_1 + 1, \ldots, o_1 + n_0 - 1$. Then the choice of the j-th element, which is an index in $T_{n_o-1}^{o_1+1}$ for index $o_2 = o_1 + j$ will lead to choices $T_{j-1}^{o_1+1}$ on the left side and choices $T_{n_o-1-j}^{o_1+j+1}$ on the right side. A further choice of obstacle index for o_3 will then do the same for one of the sides and so on. Fig. 4.21 shows an example with six indices that follows this recursive construction. Each choice is represented by an edge with the obstacle index written on it. It is straightforward to see that this recursive construction leads to a structure that has a very large number of edges. More precisely the number of edges in the tree starting at T_k^i is given recursively through $Edges(T_k^i) := k + \sum_{j=1}^{k}(Edges(T_{k-j}^i) + Edges(T_{j-1}^i))$ with $Edges(T_1^i) := 1$ and $Edges(T_0^i) := 0$. But this is only because we rewrite multiple occurrences of the same choice sets T_k^i for identical k and i as marked in fig. 4.21 with a dashed circle. If o_1 is chosen, then for every $k \in \{1, \ldots, n_o - 1\}$ we have $n_o - k$ distinct T_k^i, i.e. $i \in \{o_1 + 1, \ldots, o_1 + n_o - k\}$. This leads to a

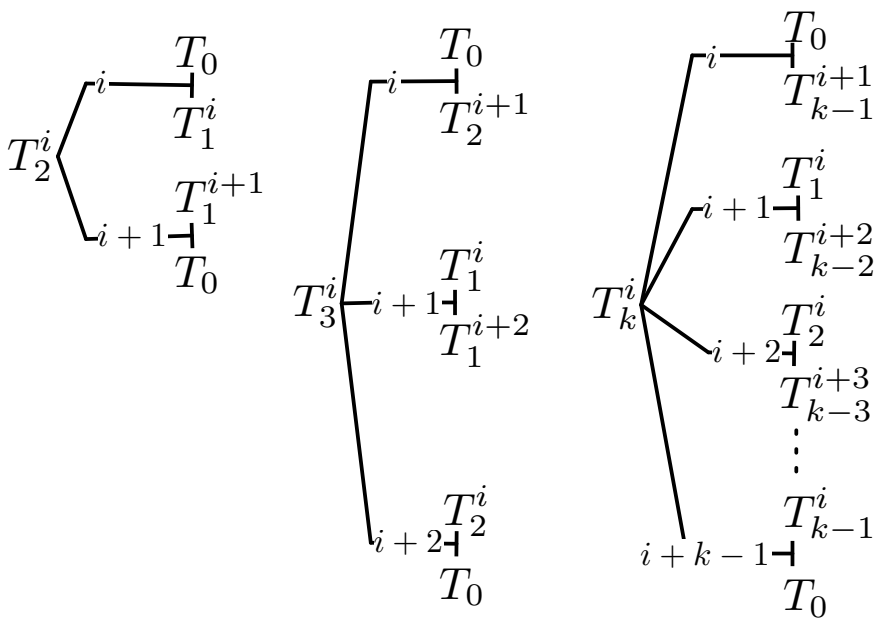


Figure 4.19: An illustration of how a set of choices T_k^i splits into left and right sides, depending on which obstacle index in T_k^i is chosen. A choice is represented by an edge towards a left and right side with the chosen index written on the edge.

total of $\sum_{k=1}^{n_o-1} n_o - k = \frac{n_o^2}{2} - \frac{n_0}{2}$ distinct choice sets. Every choice set T_k^i itself has exactly k edges and hence the total number of distinct edges leaving all sets is:

$$\sum_{k=1}^{n_o-1} (n_o - k) \cdot k = \frac{n_o^3 - n_o}{6}. \tag{4.3}$$

Fig. 4.22 shows this compressed polynomial size tree-like structure which now represents all choices.

Now that we can describe how a choice of an obstacle index influences further choices we can start describing the cost of each choice. One key observation is that the costs of all splits occurring on a left side are independent of the choices made on the right side and vice versa. Given a set of choices $T_k^i = \{i, i+1, \ldots, i+k-1\}$ let us consider the j-th element of T_k^i, i.e. obstacle index $o = i+j-1$ and compute

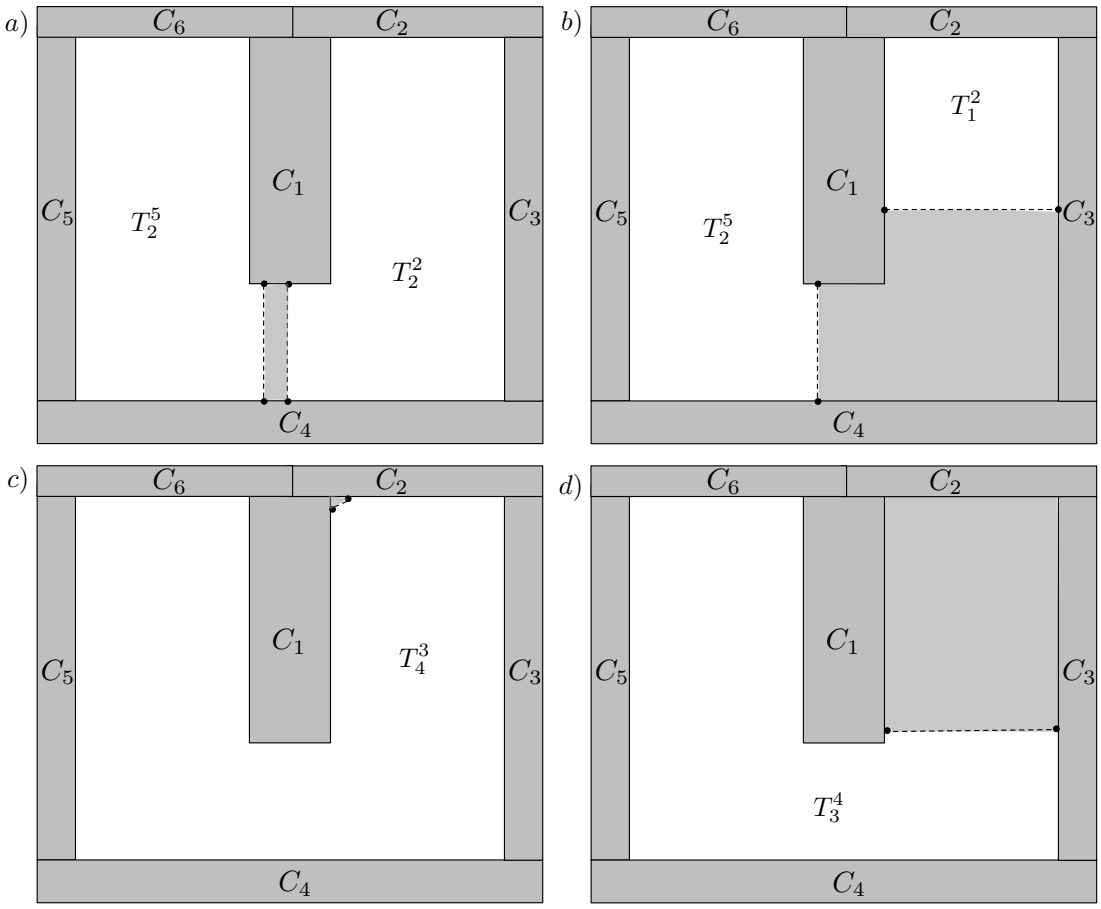


Figure 4.20: Part a),b),c) and d) show the environment in four different states with the respective sets of choice that exist in the respective state.

the cost of a split towards obstacle C_o. Since we are considering the set T_k^i we know that the sweep line that is being split at C_o is $l_{i-1,i+k}$. Consulting our oracle to obtain the lowest cost split point $p \in C_o$ we can compute the cost of extending $l_{i-1,i+k}$ to split on C_o. We can compute this cost for every outgoing edge of a choice set. Write $c(o|T_k^i)$ for this cost (the cost of choosing o out of T_k^i). If the oracle returns p in polynomial time, then all edges can get a cost also in polynomial time since the number of edges is $O(n_o^3)$ due to Eq. 4.3.

Up until now we have ignored another fundamental aspect of this representa-

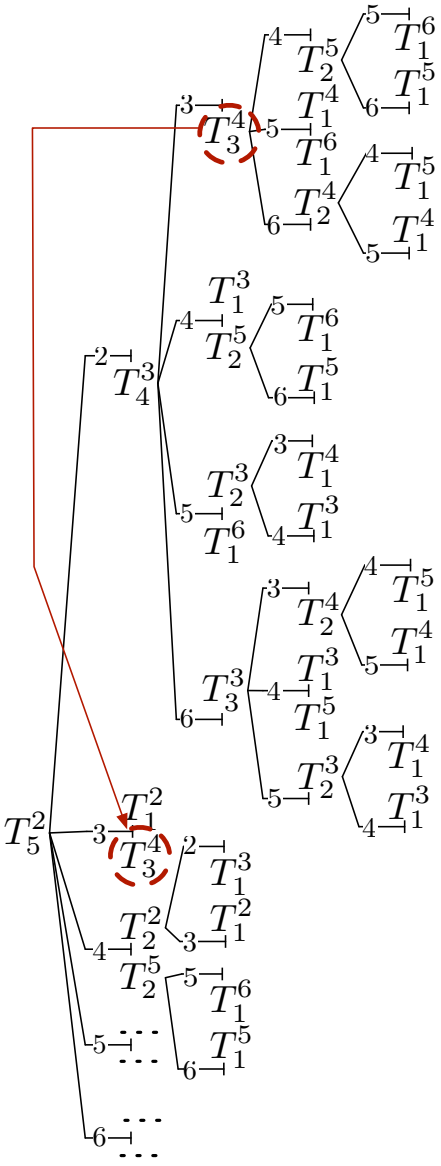


Figure 4.21: An example of a choice tree for $o_1 = 1$. The empty set T_0 is not drawn. The recursive construction is given in fig. 4.19

tion of choices. If we separate the choices into left and right sides the split costs are independent of each other, but we have not considered the cost for the sweep lines at their blocking positions, i.e. the cost of all sweep lines that are not being moved with the next choice. Let us write the cost of the blocking sweep line for

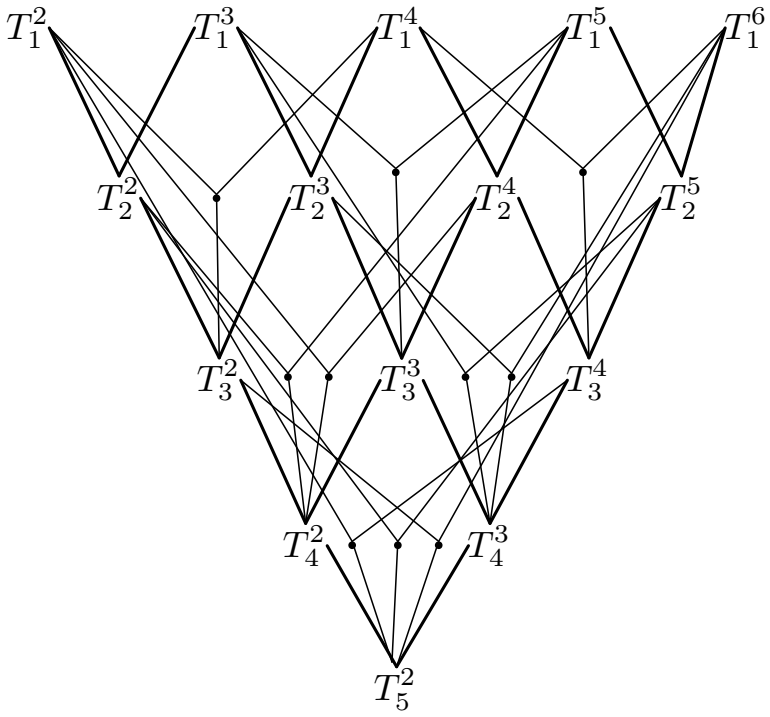


Figure 4.22: A compressed version of the tree structure from fig. 4.21. Every unique T_k^i is drawn exactly once. The number of edges grows as a polynomial with degree 3 in the number of obstacle indices.

choice set T_k^i as $b(T_k^i) = c(l_{i-1,i+k})$. More precisely the problem is that once the choices are split left and right we have to decide in which set to make the next choice. This is equivalent to the problem of deciding which vertex to clear next in a surveillance graph, since the other choice set still has a blocking sweep line that can add to the total cost, just like a block in Graph-Clear. In fact, we will now cast this problem into a surveillance graph.

A sequence of choices starting with choice set T_k^i is essentially a special subtree in the tree-like structure from fig. 4.22. In fig. 4.23 we show two sequences of choices. Such subtrees are trivial to create by starting at the first choice set and setting it as a root vertex and then just following exactly one outgoing edge for

each choice set. Notice that an outgoing edge from a choice set can that split into two new choice sets in which case both sides have to be followed. One can think of these subtrees as paths that can bifurcate at every choice. This is best understood visually with the example from fig. 4.23 which shows two special subtrees. Note that since there is a separation into left and right sides a sequence of choices in this tree-like structure does not directly correspond to a sequence of all obstacle indices. For example, the choices leading to one of the subtrees in fig. 4.23 can correspond to obstacle index sequences $4, 2, 3, 5, 6$ and $4, 2, 5, 3, 6$ and $4, 5, 2, 6, 3$ and so on. It merely determines that 4 is first, 2 is before 3 and 5 is before 6. Clearly, these different obstacle index sequences can have different cost. To find out which one has lower cost we construct a surveillance graph from this subtree. This surveillance graph is actually quite simple. We already know that every choice out of a set T_k^i has a cost to reach the split. Hence every outgoing edge of a choice set can become a vertex of the surveillance graph. In fig. 4.25 we show all the each vertices for every possible choices in T_k^i. The chosen vertex will receive one edge for each non-empty new choice set, the left and the right side. The weight of the vertex is $c(o|T_k^i)$, the cost of choosing obstacle index o from T_k^i. Let o be the j-th index in T_k^i, then the weight of the edge to the left is simply $b(T_{j-1}^i)$ and for the edge to the right the weight is $b(T_{k-j}^{i+j})$. These edges go to vertices that receive a cost depending on the next choice made on the left or right. This leads to a recursive construction in which the leaves are vertices with weight $w(i|T_1^i)$. Fig. 4.24 shows two surveillance graphs created from two subtrees. A strategy on this surveillance graph (which is itself a tree), starting at the root, now uniquely determines an obstacle sequence. An optimal strategy determines the lowest cost obstacle sequence given the choices made from the choice sets.

Unfortunately, the number of subtrees is still very large and grows at least ex-

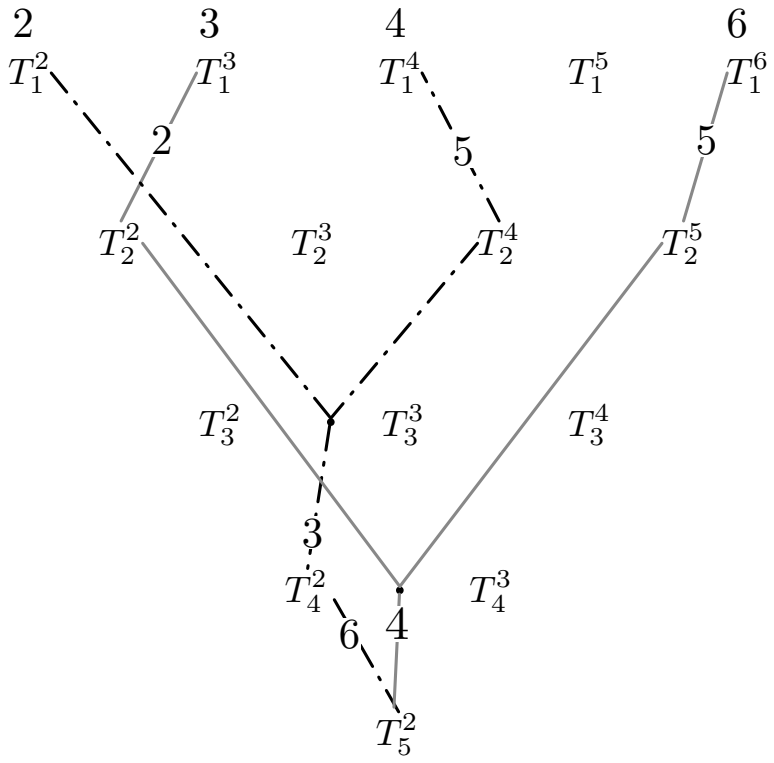


Figure 4.23: Two different paths in the tree-like structure from fig. 4.22. Note that if a choice set splits into a left and right we need to follow both sides. Fig. 4.24 shows how these two paths lead to a surveillance graph.

ponentially, as given by the recursive formula $Subtrees(T_k^i) = \sum_{j=1}^{k} Subtrees(T_{k-j}^i) \cdot Subtrees(T_{j-1}^i) \geq 2 \cdots T_{k-1}^i$ with $Subtrees(T_0^i) := 1$ and for $k > 2$. Every such subtree corresponds to a surveillance tree on which we compute an optimal strategy which determines a unique sequence of obstacle indices, the best possible sequence of the given subtree. Doing this for all such subtrees we can find the optimal sweep schedule, but this is obviously not very practical. Since the subtrees overlap we can attempt to adapt the Graph-Clear algorithms and assign labels to edges in the entire compressed choice tree.

Adapting the label-based algorithm from Section 3.5 is rather straight-forward

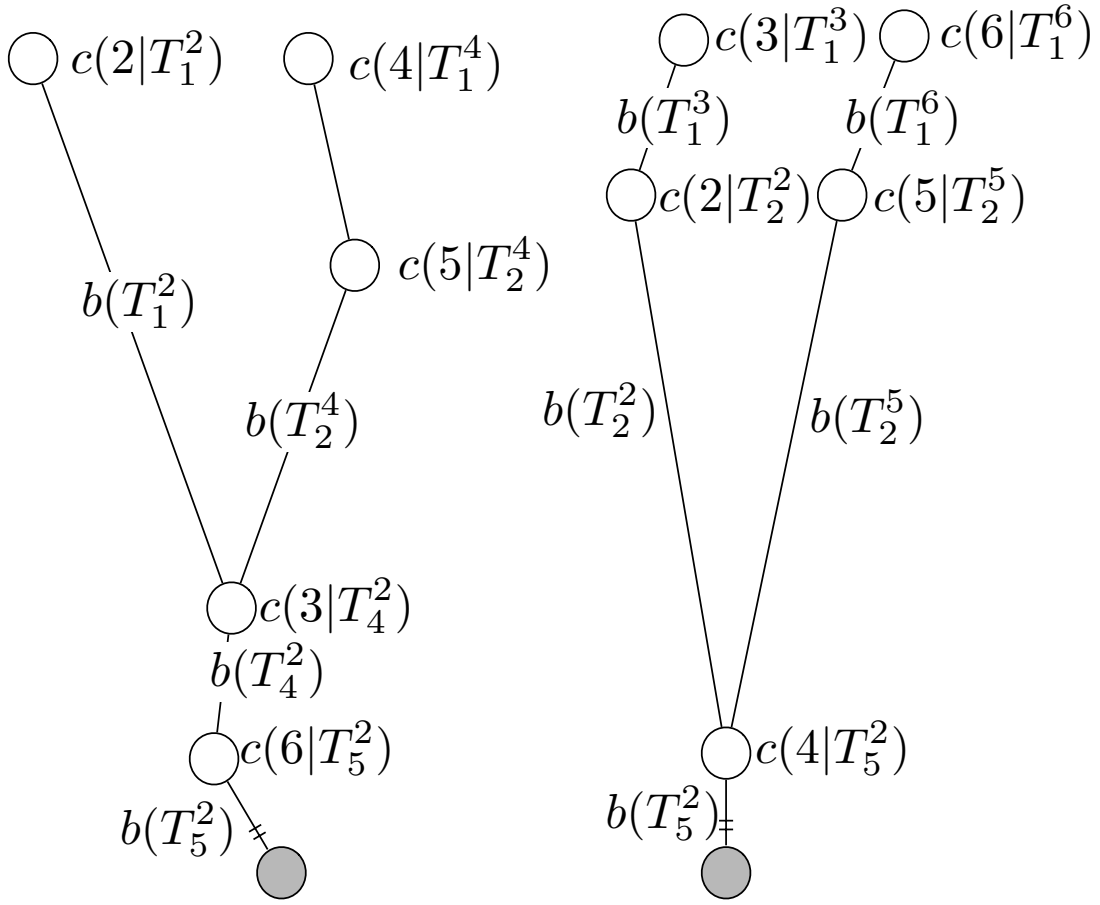


Figure 4.24: The figure shows two surveillance graphs that correspond to the two paths chosen in fig. 4.23. The cleared vertex at the bottom represents the choice of o_1 as the first obstacle.

and we can actually achieve polynomial time complexity. Recall that making a choice in a set T_k^i also chooses which vertex to be added to the surveillance graph (see fig. 4.25). The edge is always given by T_k^i with weight $b(T_k^i)$, but for the next vertex we have k choices and we would like to pick the one that will lead to a lower cost to clear it and further vertices behind it. Let us write e_k^i for the edge associated to T_k^i. We can now compute labels for edge e_k^i, one for each of the choices in T_k^i. This works recursively by starting at the leaves. A

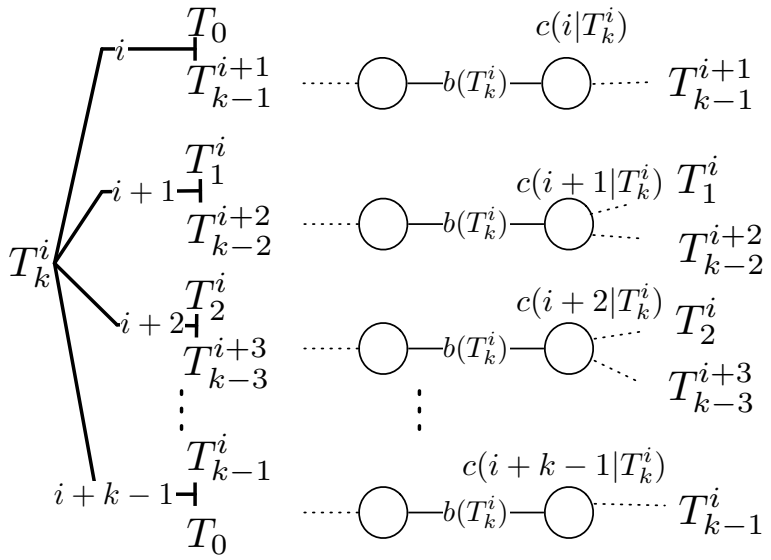


Figure 4.25: This figure shows how a choice set T_k^i leads to an edge with weight $b(T_k^i)$. The next vertex beyond that edge is determined by the choice made in T_k^i and all alternatives with their respective weights are shown. The next choices made in the choice sets for the left and right side create new vertices. The edges towards these are marked with dashed lines.

leaf vertex is always a vertex given by a weight $w(i|T_1^i)$ and edge towards that vertex with weight $b(T_1^i)$. The label that will be given to the edge e_1^i is then $\lambda(e_1^i) = c(i|T_1^i)$. For T_k^i with any $k = 2, \ldots, n_o - 1$ we first compute a label for each choice. Let j be the j-th index in T_k^i, then $\lambda_j(e_k^i)$ is the cost of clearing the entire subtree given that we chose the j-th index in T_k^i. It is computed like the labels for Graph-Clear, except that we only have two deal with two neighbors, left and right. Let $\rho_{left} = \lambda(e_{j-1}^i) - b(T_{j-1}^i)$ and $\rho_{right} = \lambda(e_{k-j}^{i+j}) - b(T_{k-j}^{i+j})$. The side with smaller ρ has to be cleared first. So if $\rho_{right} < \rho_{left}$, then $c = \max\{\lambda(e_{k-j}^{i+j}) + b(T_{j-1}^i), \lambda(e_{j-1}^i)\}$, otherwise $c = \max\{\lambda(e_{j-1}^i) + b(T_{k-j}^{i+j}), \lambda(e_{k-j}^{i+j})\}$. Now, the maximum cost occurring while clearing both sides and the center vertex

with weight $w(o|T_k^i)$, with $o = i + j - 1$, becomes:

$$\lambda_j(e_k^i) = \max\{w(o|T_k^i), c\} \tag{4.4}$$

Finally, we only consider the choice with the lowest cost label which then becomes the label for e_k^i, i.e. $\lambda(e_k^i)$ given by:

$$\lambda(e_k^i) = \min_{j=1,\ldots,k}\{\lambda_j(e_k^i)\}, \tag{4.5}$$

Fig 4.26 labels $\lambda_j(e_k^i)$ attached to edges for $k = 1, 2, 3$. Only the label $\lambda(e_k^i)$ will be used in subsequent label computations for choices higher up in the tree. This implicitly prunes all choices at a T_k^i but one. Fig. 4.27 shows the choice tree from 4.22 pruned by computing minimum labels. Notice that we will always have exactly one subtree now which is already the subtree with the lowest label cost.

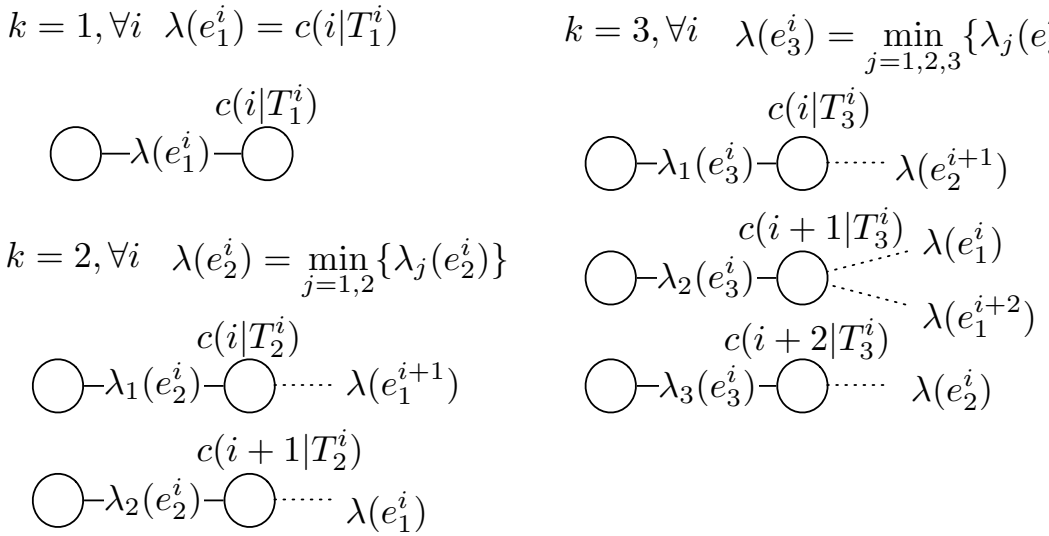


Figure 4.26: Illustration of the computation of labels $\lambda_j(e_k^i)$ and $\lambda(e_k^i)$ for $k = 1, 2, 3$.

But the label-based algorithm and hence the minimum label cost is not optimal for Graph-Clear. The optimal algorithm for Graph-Clear on trees, presented

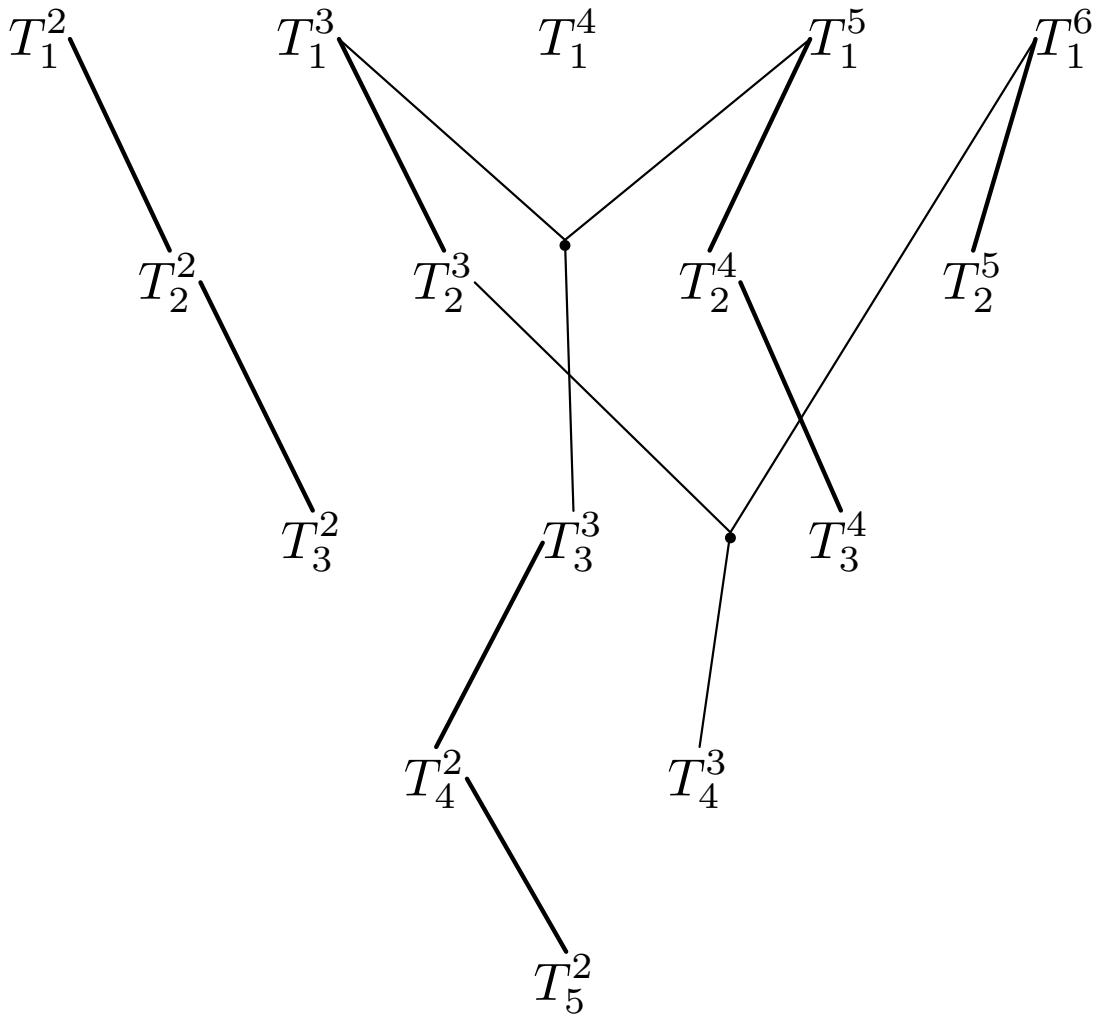


Figure 4.27: Computing labels for edges representing choice sets implicitly prunes all outgoing choices from a choice set to one. The figure shows such a pruned tree and only the edges for choices are shown that correspond to a vertices that leads to the minimum label.

in Section 3.6, is slightly more complicated. Recall the main difference between the two algorithm. The optimal algorithm with cut sets allows entering a subtree of the surveillance tree, clearing it partially and returning to another subtree. The label-based approach on the other hand resembles a depth-first strategy since

once it enters a subtree it clears it entirely before it clears a neighboring subtree. Instead of labels the optimal algorithm has cut-set that represent the different states of a subtree behind an edge and the associated clearing and blocking costs. This improvement makes the algorithm optimal. But unfortunately, it also introduces a complication that makes it difficult to adapt it to a choice tree in polynomial time. The reason for this is as follows. The label computation above allows us to prune all choices but one, namely the one with minimum label cost. This is possible since once a side is chosen it will be cleared and hence the lowest cost subtree is the best alternative. Now, for cut sets this is not necessarily true. We may have two choices of vertices (and hence subtrees) out of which one has a higher total clearing cost, but also has intermediate cuts with low cost and low blocking weight while the second choice leads to a subtree with lower overall cost, but no intermediate cuts (recall Section 3.6). Clearly, if one is presented with these two choices at the root of the choice tree one would pick the one with the lower cost. But deeper within the tree it is not clear which choice will ultimately lead to a lower overall cost at the root, since the subtrees will be combined with other subtrees through a new vertex closer to the root of the choice tree. This combination may make the choice with the cut set that has an intermediate cut a better choice by reducing the blocking cost and then allowing to clear the other subtree at lower overall cost. The conclusion is that we need to keep track of all choices and their associated cut sequences that can have a potential advantage further up in the choice tree and cannot prune these choices. In the worst case we may have to keep track of all choices, which cannot be done in polynomial time. Unless one can prove a lemma that guarantees that choices can be pruned from T_k^i on the basis of their cut sequences the cut set algorithm cannot be applied in polynomial time to yield optimal sweep schedules. Hence, the problem of determining optimal sweep schedules in polynomial time remains subject to

further investigation.

4.4.1 Split Points for Sweep Lines

There are a few technical considerations for the existence and computation of the lowest cost split point p. We discuss these here since they offer a promising direction for further work towards finding conditions or a complete algorithm for optimal sweep schedules. Recall that we seek a split point $p \in \delta C_k$ that minimizes the joint cost of a sweep line from p to C_i and a sweep line from p to C_j. Previously, in Section 4.2.2 we encountered a similar problem, but with three obstacles that shared a Voronoi vertex. In this case we can guarantee that there is a straight line connecting C_k with both C_i and C_j, which makes the problem much simpler. To generalize this to obstacles that are not necessarily close to each other we have to consider the following. Write $l_i(x)$ and $l_j(x)$, $x \in C_k$ for a lowest cost sweep line from x to C_i and C_j respectively. Note that l_i and l_j must not intersect nor can they intersect with the boundary of any obstacle other than C_k, C_j, C_i. The goal is now to minimize the function $c_{split}(x) := c(l_i(x)) + c(l_j(x))$ defined on $x \in \delta C_k$, which can be solved as an optimization problem. The problem lies more with finding $l_i(x)$ and $l_j(x)$. Consider the cost function c_{alt} instead of c. With this cost function the lowest cost split line is always the shortest path from x to C_i or C_j respectively. Finding this path in our simply-connected environment would involve some elements of path planning. But such paths, if they are not straight lines, will be very close to other obstacles. Now, if one can move a midpoint of one of these two sweep lines $l_i(x)$ and $l_j(x)$ onto one of these close obstacles, denoted by o' at no larger cost, then this obstacle can be added to B prior to o at no additional cost. This means that choosing o' before o has lower or equal cost than choosing o before o'. Intuitively, this means that we can

ignore far away obstacles as the next choice. Working out this relationship may well lead to conditions under which we can reduce the number of choices that have to be considered. This directions seems most promising for further work on this topic and towards improving the construction of sweep schedules by giving precise conditions for pruning choices for the obstacle index sequences.

4.5 Discussion and Conclusion

In this chapter we presented a novel pursuit-evasion problem for complex two dimensional environments that captures the limited range assumption by modeling the sensing capabilities as lines that have a cost. The problem is a suitable model for environments in which the distance between non-adjacent obstacles is larger than one robot's sensing range. Solutions to the problem are formulated as moving sweep lines which represent the movement of robots as they follow the lines and cover them with sensors. We developed a heuristic approach to compute sweep schedules from Voronoi Diagrams using Graph-Clear strategies and showed that there are instance when it is suboptimal. We then developed a combinatorial perspective that allows us to consider less restricted sweep schedules than those created from a Voronoi Diagram. This perspective relies on the ability to compute lowest cost obstacle index sequences. We can compute these with Graph-Clear algorithms, but it turns out that we cannot, as of now, guarantee that the optimal Graph-Clear algorithm based on cut sets can be adapted to this problem in polynomial time. Here there is probably the best starting point for further work on the topic, by either attempting to show that Line-Clear is already NP-hard in simply-connected environments or finding a constraint on the number of cut sets that have to be considered that lets the algorithm run in polynomial time. Once this is achieved a more rigorous formal treatise is then justified to support

the proof of the respective conjectures. Overall, it can be said that Line-Clear is already a hard problem even in simply-connected environments.

Despite the difficulties in finding optimal solutions the Line-Clear approach has a practical benefit. In practice, even the sub-optimal label-based algorithm performs sufficiently well and has the advantage that it reduces the travel distance of the robot team justifies its use. Every edge in the graph is traversed at most twice in the depth-first approach. The cut-set algorithm, however, can require leaving and entering the same subtree multiple times and can have much longer travel distances. Hence, even for Line-Clear the label-based polynomial time adaption is useful given that travel time is a constraint for the clearing process. In Chapter 5 we shall see the ideas for Line-Clear being used as a practical method to extract graphs via the Voronoi Diagram. Furthermore, we can attempt to apply the Line-Clear ideas even when no map is given, as we shall see in Chapter 6. Therein robots simply form sweep lines between obstacles and discover the environment by moving these around until they encounter new obstacles on which they split. Removing the assumption that a map is given is obviously a great step towards broadening the types of applications that Line-Clear can be useful for. This allows the deployment of robots to occur in a partially or completely unknown environment or when part of the environment is dynamic and unpredictable. In such cases a strategy can only be planned for the part of the environment which is already known. A robot team may for example have to check whether a door is locked or after a in a disaster whether a corridor is obstructed by debris. In a completely unknown environment the exploration starts from scratch without any initial knowledge and preliminary strategies have to be adapted to new information that is collected about the environment. In summary, there are a number of open questions regarding Line-Clear that encourage further work, but its core ideas and concepts can already be useful for

applications, as we shall see in the next chapters.

CHAPTER 5

Extracting Surveillance Graphs From Maps

In this chapter we present automated methods for the extraction of surveillance graphs from maps. The relationship of a graph to its maps is obviously affected by the type of implementation for sweep and block actions. But there are also general principles that can be exploited, such as that good positions for blocks are at narrow parts of the environment. Section 5.1 exploits this principle using the Voronoi Diagram introduced in Section 4.2.1 to partition maps by finding minima on Voronoi edges on which it sets up blocks. Following this in Section 5.2 we present an extraction method based on ideas from Line-Clear and Chapter 4 which leads to some improvements from the previous extraction method.

5.1 Voronoi-based Extractions

In this section we provide a first step to close the loop between the graph-based theoretical formulations of Graph-Clear and practical scenarios. Preliminary results from this section also appeared in [KC08a]. We show how it is possible to algorithmically extract suitable surveillance graphs from occupancy grid maps. We also identify local graph modification operators, called contractions, that alter the graph being extracted so that the Graph-Clear problem can be solved using less robots. The algorithm we present is based on Voronoi Diagrams from Section 4.2.1. Voronoi Diagrams and approximations thereof can be computed in

a variety of ways, such as watershed like algorithms or directly from sensor readings. Our algorithm is evaluated by processing maps produced by mobile robots exploring indoor environments. It turns out that the proposed algorithm is fast, robust to noise, and opportunistically modifies the graph so that less expensive strategies can be computed.

One of the main aspects of Graph-Clear is its conceptual mechanism that allows to model and study different search and clear strategies by abstracting from the underlying robotic platforms used. Graph-Clear in particular aims to model surveillance tasks where multiple robots with limited sensing capabilities cooperate to detect intruders in complex environments. Theoretical properties of Graph-Clear, as well as solving algorithms have been extensively studied in Chapter 3. Here we rather focus on the practical deployment of robot teams that clear complex environments by using the Graph-Clear formalism. In particular we address the problem of automatic extraction of surveillance graphs from occupancy grid maps. Informally speaking, this task entails allocating graph vertices on rooms and graph edges on corridors or connections between rooms. The reader will realize that this step is similar to creating topological maps from occupancy grid maps. In order to enable the Graph-Clear framework it is also necessary to determine weights associated with edges and vertices and then partition the environment to yield a graph with good strategies. These weights measure the cost, i.e. the number of robots, needed to enforce relevant properties on the graph, like for example preventing intruders from passing through a door, or from hiding in a room. The method to assign weights presented in this section is parametric with regard to the sensing capabilities of the robots used to implement the strategies, and has therefore general applicability. In addition, while extracting graphs from occupancy grid maps we have identified certain operations that modify the graph so that the algorithm generating strategies produces solutions requiring

less robots.

While most ideas herein generalize to varying implementations of the basic Graph-Clear actions, i.e. blocking edges and sweeping vertices, we will present experimental results for particular implementations of these actions on two realistic robot maps, one created from a laser range finder on a P3AT at UCMerced and one generated from the Radish online robotics data repository [HR03] called "sdr_site_b".

Our proposed approach is based on a Voronoi Diagram for the given environment. Voronoi Diagrams can readily be constructed from sensor data, occupancy grid maps or vectorized maps. For our purposes we will assume a two dimensional grid map. Let us first discuss how blocking on edges and sweeping on vertices can be implemented.

5.1.1 Blocking

The requirement from Graph-Clear is that a block detects intruders as they attempt to pass through the edge. This can be fulfilled by continuously covering the area corresponding to the edge with sensors. In practice no sensor can give a 100% guarantee that an intruder will be detected and one may wish to cover the area with multiple sensors, or if speed constraints apply to intruders then one might just need one robot patrolling fast enough along the edge while not covering it all at once. The actual choice of implementation can differ widely depending on the application. Our basic assumption is that any implementation will benefit from edges placed at narrow section of the environment.

5.1.2 Vertex sweeping

For implementations of vertex sweeping the choices are even more manifold. In open uncluttered regions one could use the sweep-pursuit-capture strategy presented in [BBH07]. In cluttered but not too large regions the approach from [MRS05] could be applied. For vertices in which the sensor range is larger than the diameter of the region one could use the approach from [SRL04]. These methods are shortly presented in Chapter 2. A good choice for an implementation of the sweeping action is heavily dependent on the type of robot used and the shape of region that the vertex represents. That being said, however, one of the advantages of Graph-Clear is not only that it can scale local clearing methods to very large environments, but that it allows us to use simple clearing methods for the vertices which could be implemented by very simple robots, even those just roaming around a vertex randomly in which case the time they are allowed to spend in a vertex determines a likelihood of detection. Obviously such vertex sweeps do not assume a worst case adversary. In Section 5.1.6 we will demonstrate the graph construction with a very simple bounding box sweeping.

Our approach for the graph construction proceeds in two stages. The initial construction places edges at every possibly beneficial position, i.e. every narrow section of the environment. But already in a simple cases such as in fig. 5.1 we can see that too many edges are created. We would like to reduced the number of edges and vertices to improve the graph taking into consideration the vertex sweeping and edge block implementation. Hence the construction proceeds with *contracting* (i.e. merging) vertices from the initial construction when an edge between two vertices is not beneficial. The next two sections present this two-staged construction.

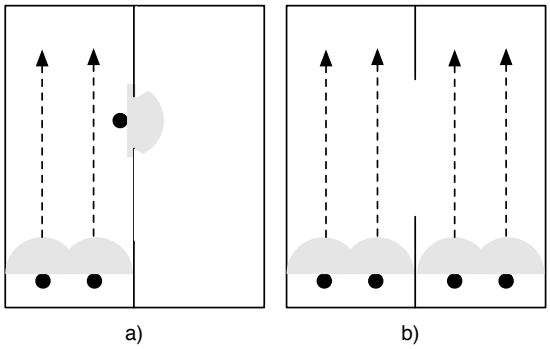


Figure 5.1: Illustrating the advantage of narrow connections between open regions. For robots with a limited sensing range the environment in part a) can be cleared with 3 robots, while the one on part b) requires 4. The reader should note that the surface is the same.

5.1.3 Initial Graph Construction

Given a map, we first compute the Voronoi Diagram. Similar to [Thr98], in which the graph was used for fast path planning, we use the minima of the local clearance function defined on the edges of the Voronoi Diagram to create the surveillance graph. The local clearance function on the edges of the Voronoi Diagram is simply the distance from the point on the edge to the nearest obstacle point. A minima is considered any point for which $\exists \varepsilon > 0$ s.t. within its ε-neighborhood there are no other points with strictly smaller clearance and $\forall \delta > 0$ there is at least one point within its δ-neighborhood with strictly larger clearance. In colloquial terms, one very close neighbor should be larger and within any small neighborhood none of the neighbors should be smaller. Here we differ from the construction in [Thr98] which considers all points for which all points within an ε-neighborhood are not smaller, which would also include entire plateaus and also those of maxima. With our definition, if a minimum value is achieved on a compact subset of the edge we select the two end points of the set. Fig. 5.2 shows a Voronoi edge and two

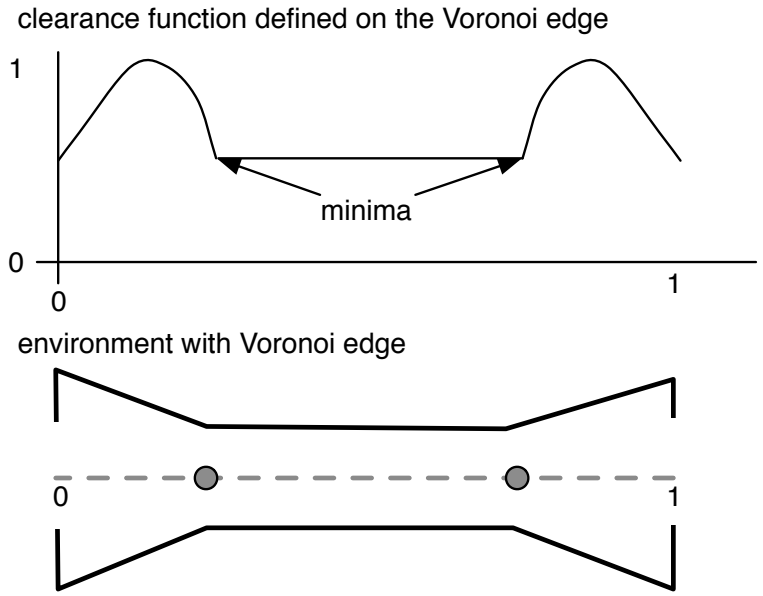


Figure 5.2: A simple environment with a Voronoi edge in the center as a dotted line and the clearance function in the graph on top. The minima on the Voronoi edge are marked by grey circles.

minima on it.

It is worth to note that minima cannot lie on vertices of the Voronoi Diagram, so by only considering edges we do not miss any minimum. For each minimum we consider the lines from the minimum to the two nearest obstacle points, which since we are on a Voronoi edge lie in two different obstacles. These lines will be represented by an edge in the surveillance graph, i.e. we are partitioning free space into regions based on these lines and each region becomes a vertex in the surveillance graph. Given an accurate Voronoi Diagram this construction yields a valid partitioning and hence a valid graph. Fig. 5.3 shows such a construction.

Once the edges and vertices of the surveillance graph are constructed we use the implementation of the edge block and the vertex sweeping actions to compute the weights for the surveillance graph. This concludes the initial construction.

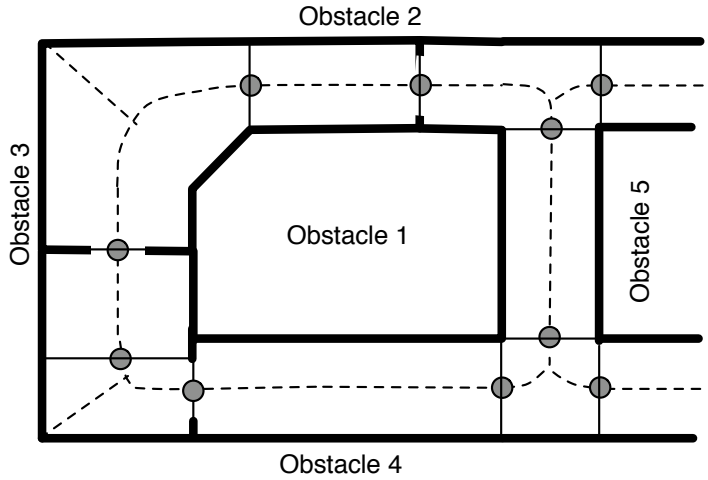


Figure 5.3: A Voronoi Diagram and its minima. The Voronoi Diagram is marked with grey dashed lines, the minima with grey circles and the lines to the closest obstacle points with thin black lines. Obstacle boundaries are thick black lines. Corners in corridors tend to produce minima, unless a narrow part proceeds it as seen in the upper left corner of the figure.

The resulting surveillance graph will be the starting point to find a graph with better strategies. In most environments the presented construction will introduce many more edges than would be beneficial, as seen in fig. 5.1 in which an edge between the two regions is only of advantage when it is sufficiently narrow. Since this depends on the implementations of the actions and the type of robot, we will have to introduce a method that considers this when improving the surveillance graph. Gladly, we can use a general approach that merely calls the weight computation of the implementations which is discussed in the next section. Thereafter we will demonstrate the method with two realistic examples.

5.1.4 Improving the graph

The initial construction does not guarantee the existence of good strategies for the Graph-Clear problem. Hence, the next step is to contract vertices wherever this may lead to a better strategy. This will also remove spurious minima that may be introduced by noise in the map or approximations of the Voronoi Diagram, as seen in fig. 5.4.

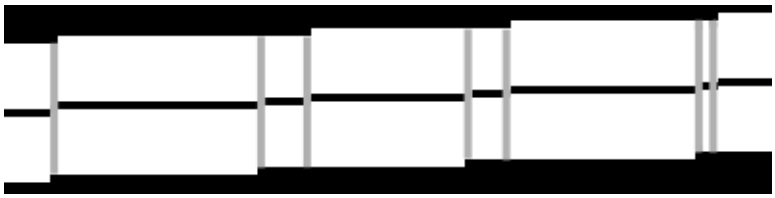


Figure 5.4: An example in which a discrete approximation of the Voronoi Diagram in a grid map leads to introduction of unwanted edges. The black line in the center is the Voronoi Diagram edge and minima are marked by grey lines

The simplest type of contraction is between a leaf vertex l and its sole neighbor v. Let e be the edge connecting them and v' be l and v merged. Contracting l and v cannot make the strategies for the surveillance graph worse if $w(v') - w(v) \leq w(e)$. It is easy to verify from Eq. 3.10 in Section 3.5 that under these conditions none of the labels in the graph can get larger. Furthermore, from Section 3.7.2 it follows that if e is not in the tail of another a label on an edge coming from the neighbors $v_2, \ldots, v_m$ of v, then this label necessarily improves due to the removal of the edge. Recall that the tail of a label, in colloquial terms, is the set of edges whose ending vertices are cleared before sweeping the one with the highest cost $c(e_i)$.

The second contraction is for vertices of degree two. Let c be such a vertex, v_y, v'_y its neighbors and e, e' the respective edges. If $w(e) \geq w(e')$ and $w(v^c_y) \leq$

$w(v_y) + w(e) - w(e')$ then a contraction of v_y and c into a single vertex v_y^c is of advantage. Fig. 5.5 shows this contraction.

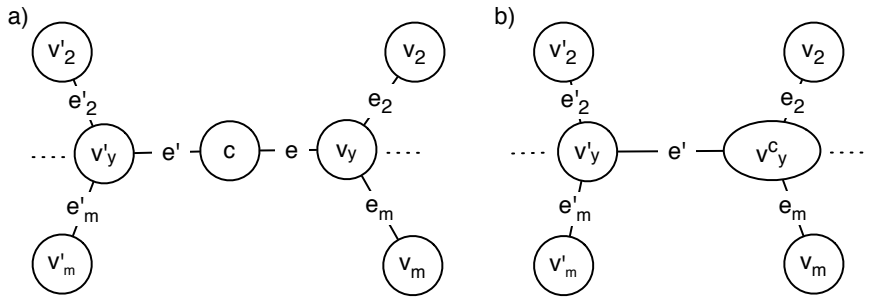


Figure 5.5: A contraction of a vertex with degree two and its neighbor. Part a) shows the initial graph and part b) the graph after the contraction.

If either of these two conditions is satisfied then a contraction is guaranteed not to lead to worse strategies on the surveillance graph, regardless of the value of the labels on the edges. Analyzing general contractions is beyond the scope of this paper and involves a careful and formal consideration of the recursive nature of the label computation, occurrences of the maxima in batches and the role of the tails as it is done in [KC08b]. We will, however, demonstrate the potency of already the simple types of contractions in our experimental section.

5.1.5 Implementing blocking and sweeping actions

As previously indicated, blocking and sweeping actions of a strategy for the surveillance graph can be implemented in manifold variations depending on the particular needs of the application. The requirements for the implementation is merely that the computation of the weights on vertices and edges is possible and that the action can be executed. When using vertex sweeping strategies that have strict assumptions, such as the region of the vertex being simply connected, these assumptions has to be considered when building the graph. In this case one

needs to detect vertices in which this is not given and subdivide them into further vertices. Issues with particular types of sensors, their range and error rates are all aspects that come into play when designing the implementation details. Once the implementation gives satisfactory guarantees for the detection of intruders, then Graph-Clear can be applied. At this point one could also consider the probabilistic variant of Graph-Clear and instead of providing weights, the implementation would provide a function describing its performance dependent on the number of robots used (see Section 3.8).

5.1.6 Experimental Results

To demonstrate some the presented ideas in an application we constructed a robot grid map of part of the UC Merced Science and Engineering building with a Pioneer P3AT mobile platform equipped with a SICK PLS200 laser range finder. The map is built using the GMapping software [GSB]. Figure 5.8 shows this map, which we will further denote as UCM map. As a second map we used the sdr_site_b data set from the Radish online robotics data repository [HR03]. This latter set will be denoted as SDR map, and the correspondingly generated map is seen in figure 5.9.

To both maps a low-pass filter was preliminary applied to remove noise and get smoother boundaries. Both maps have a complicated structure, many occlusions, loops and noisy artifacts. The resolution of the grid map for the UCM map is 690x790 pixels while the SDR map is 645x573. The free space in the UCM map and SDR is approx. 35.8% and approx. 46.4% respectively, which corresponds to 442^2 pixels for the UCM map and 414^2 pixels for the SDR map. In the SDR map there are two small pockets at the bottom of the map which were included in the open space calculation but are not accessible and hence not part

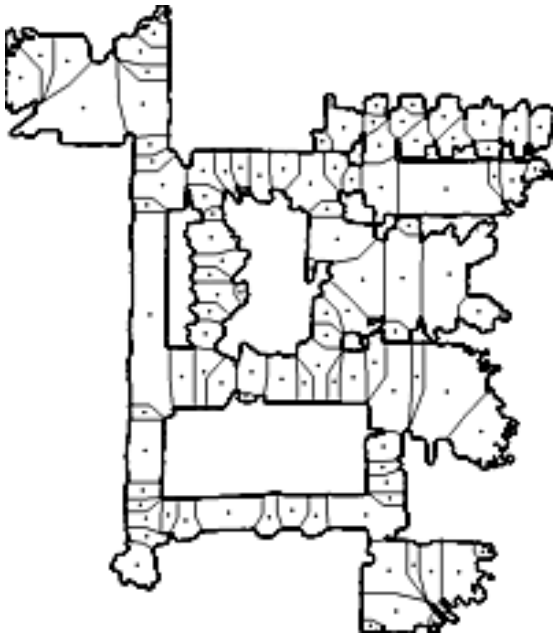


Figure 5.6: The map created by the P3AT at UC Merced with initial graph construction. The thick black lines are boundaries between free and occupied space. The small black circles are vertices placed in their corresponding region which are separated by thin lines.

of the graph construction. Since the maps are both grid maps a simple wave propagation algorithm to compute an approximation of the Voronoi Diagram has been used. The algorithm implicitly assumes that points on a straight line belong to the same obstacle and a diagonal marks a new obstacle. Diagonal collisions are permitted, complicating implementation slightly. Since our focus is on the graph construction we will spare the remaining details of the Voronoi Diagram construction noting that good algorithms have been developed in the vast body of literature on the topic.

Based on the crude approximation of the Voronoi Diagram we construct the surveillance graph by detecting the minima on the Voronoi Diagram edges. We mark every point on the Voronoi Diagram as a minima which has several neigh-

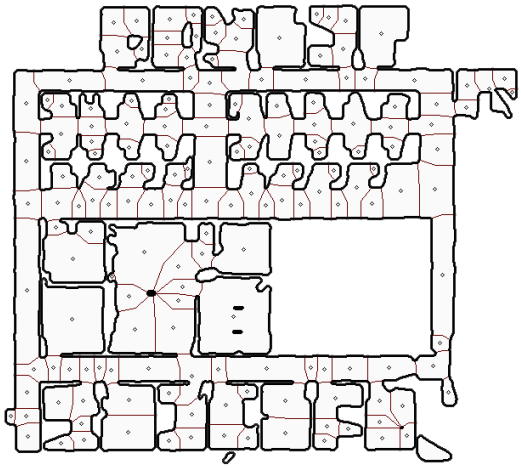


Figure 5.7: The sdr_site_b from Radish [HR03] with initial graph construction.The thick black lines are boundaries between free and occupied space. The small black circles are vertices placed in their corresponding region which are separated by thin lines.

bors in at least one direction being larger and no other point on the Voronoi edge within 3 steps in any direction being smaller. To avoid too many minima on the initial graph, we set a minimum distance between initial minima to 10 steps on the diagram. The selection of minima can be application dependent, e.g. for some applications it may be desirable to have minima that are guaranteed to be further apart. Whilst very close edges are likely to be merged in the contraction stage, it is still more convenient not to clutter the initial graph with many spurious edges when those are easy to avoid. The resulting graphs for the two maps are displayed in figures 5.8 and 5.9.

Once the surveillance graph is constructed we compute the weights on edges by computing the distance d between the two closest obstacle points of the minimum. The weight on the edge becomes $w(e) = \lceil \frac{d}{r} \rceil$ whereby r is the maximum the sensor can cover between any two points. For example, for an omnidirectional

sensor this will be the diameter of its disk, while and for a 45 degree laser range scanner it will be the maximum range of one beam. For vertices we assume a simple bounding box sweeping method, i.e. we compute a rectangular bounding box around region of the vertex. Let s be the length of the shorter side of the bounding box, then the vertex weight becomes $w(v) = \lceil \frac{s}{r} \rceil$ where r is as before. We assume only horizontal and vertical lines for the bounding box. This simple vertex sweep implementation should lead to less contractions in the given environments as it penalizes merging vertices that lead to complicated regions in which such a bounding box is a poor sweep method.

Once the initial graph is given we compute Graph-Clear strategies on it. First we convert the graph into a tree by computing the Minimum Spanning Tree (MST) with respect to to the inverse of the edge weight to yield those edges in the MST with the highest weight. Edges that are not in the MST will be blocked continuously to reduce the graph to a tree. In Section 3.10 details of this method are presented and it is shown that not all of the non-MST edges have to be blocked simultaneously, i.e. the total cost of clearing the environment can be further reduced. We then used the hybrid strategy algorithm from Section 3.7 on the tree. The partitioning problem for the hybrid algorithm mentioned in [KC08b] was solved with brute force, albeit not hindering computational performance as the solutions were computed in the order of milliseconds. Labels in all directions were computed and the vertex with the best cost was chosen as the starting vertex.

Finally, we start the contraction process which proceeds in loops, contracting all vertices satisfying the criteria from Section 5.1.4 at each iteration until no more such contractions are found. The resulting graph can be seen in fig. 5.8 and 5.9. On this graph we compute new Graph-Clear strategies.

The graph construction has been carried out for varying sensing range. A summary of the results is found in table 5.1, where r denotes the sensing range, n_0 the initial number of vertices, n the number of vertices in the final graph. The number of robots needed to execute the computed strategies are ag_0 for the initial graph and ag for the final graph. The number of non-MST edges, each corresponding to a cycle in the graph, as well as their total weight, is also given as b and b_c respectively. In the random graphs from Section 3.10 it turned out that to successfully apply the strategy from the tree to the graph we required about $0.5 \cdot b_c$ additional robots to what the tree strategy requires. Furthermore, we included the total area one can cover with the robots assuming they have an omnidirectional sensor. As c_1 we denote the area all robots needed to executed the tree strategy could cover as a percentage of the total area of the free space of the given map. For c_2 we also include all robots needed to block all non-MST edges continuously. These percentages would be reduced if we assumed a 180 degree sensor as we only need sensor coverage to maintain sweep lines, i.e. in theory even a single beam would already suffice, albeit in practice a 180 degree or omnidirectional sensor can give repeated observations of the same target which is more robust considering the erroneous nature of the sensors and the need to integrate the observations to obtain robust target detections.

5.1.7 Discussion and Conclusion

The experiments demonstrate a successful construction of a surveillance graph for two complicated environments. The number of robots needed is significantly reduced for the contracted variant of the graph. Furthermore, we can see that we can detect all intruders in the environment for just a minor fraction of the total area of the environment. More importantly, the approach scales well to

Map	r	n_0	n	ag_0	ag	b	b_c	c_1	c_2
UCM	5	108	40	68	58	3	15	0.6%	0.7%
UCM	10	108	25	37	28	3	9	1.1%	1.5%
UCM	20	108	22	18	14	3	6	2.3%	3.2%
UCM	40	108	13	10	8	3	3	5.1%	7.1%
UCM	60	108	19	9	6	3	3	8.7%	13.0%
UCM	100	108	13	6	4	3	3	16.1%	28.1%
SDR	5	172	79	42	36	8	31	0.4%	0.8%
SDR	10	172	72	22	19	7	17	0.9%	1.6%
SDR	20	172	60	12	9	7	10	1.7%	3.5%
SDR	40	172	42	7	6	8	8	4.4%	10.3%
SDR	60	172	32	6	5	6	6	8.3%	18.1%
SDR	100	172	14	6	4	6	6	18.3%	45.8%

Table 5.1: Summary of the experimental results.

large teams with each robot having only limited capabilities. The number of robots needed increases linearly with respect to their sensing range. Another minor observation is that the number of non-MST edges in the SDR map varies, which is a result of the different contractions applied to the initial graph due to the different weights. Some of the cycles are then contained within a vertex and hence do not appear in the surveillance graph. We have hence demonstrated the applicability of Graph-Clear and provided a valuable method for the construction of surveillance graphs from grid maps which already works well in practice, despite leaving open many more directions for further improvements. Particularly, we will next look at vertex sweeps and edge blocks based on Line-Clear implementations which more accurately reflect the sweeping costs of a vertex with limited range sensors. Furthermore, we showed that already simple criteria for contractions

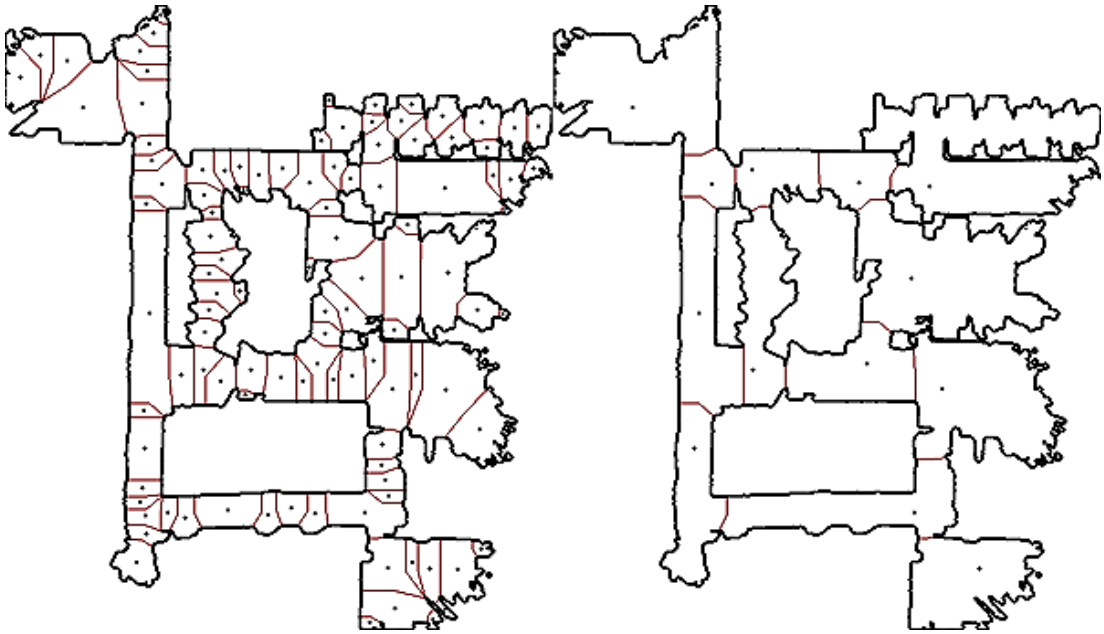


Figure 5.8: The map created by the P3AT at UC Merced with initial graph construction on the left. The thick black lines are boundaries between free and occupied space. The small black points are vertices placed in their corresponding region which are separated by thin lines. On the right is the final graph resulting from contractions.

lead to significant improvements of strategies. Also, the construction process is robust against errors in the approximation of the Voronoi Diagram as we did not use a state of the art algorithm for this purpose. Yet, contractions are by no means exhaustive and ideally a comprehensive theory of these contractions should be put into place. For all practical purposes, however, current contraction methods seem to suffice.

There are open questions with respect to to an incremental construction of surveillance graphs and strategies based on an incremental construction of a Voronoi Diagram and local optimization techniques. We shall discuss an approach for the incremental construction of a Graph-Clear as a robot team explores an

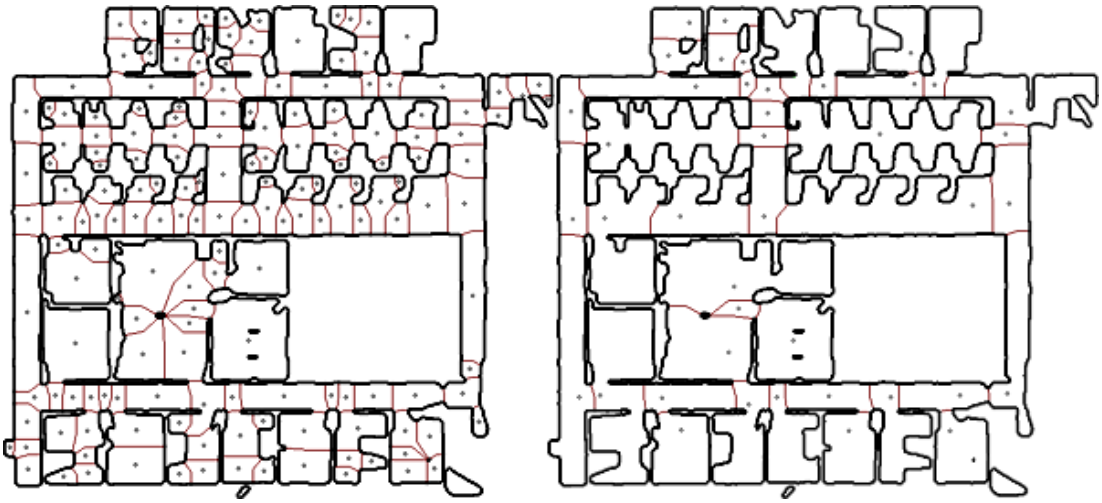


Figure 5.9: The sdr_site_b from Radish [HR03] with initial graph construction. The thick black lines are boundaries between free and occupied space. The small black points are vertices placed in their corresponding region which are separated by thin lines. On the right is final graph resulting from contractions.

environment in Chapter 6.

5.2 Line-Clear Extractions

In this section we present a method to extract surveillance graphs from occupancy grid maps by using the ideas from Line-Clear. Preliminary results from this section also appeared in [KC09c]. The extracted graphs model the complexities of any given planar environment accurately, and are constructed as duals of the Voronoi Diagram. This gives a natural geometric embedding for *blocking* and *sweeping* actions of the graph into the environment by directly associating them to *sweep lines* that robots have to cover with sensors. With this method we solve two problems at once, namely the generation of surveillance graphs and the implementation of actions on a robot team. Sweep lines can then be directly

translated into control inputs to the robot team. The new method is superior to previous heuristics for the extraction of graphs not only through its direct geometric relationship to the environment, but also due to its increased performance in direct experimental comparisons.

The theoretical aspects of the approach applied here are discussed in Chapter 4. In this section we are primarily concerned with practical ramifications to model the sensing capabilities of a robot team as lines between obstacles. For this we consider a sweep line as a line covered by the sensors of multiple robots which ensure that no intruder can pass through. As the line moves through the environment, robots continue to cover it with their sensors. In particular, we shall apply the ideas from Section 4.2 and use them to obtain surveillance graphs of the environment.

5.2.1 Implementation

The actual implementation to use the line clearing approach to construct a surveillance graph from an occupancy grid map proceeds in several stages. First, to smoothen the map we convert it to a polygon by computing its α shape using the CGAL library [Da08]. These shapes are frequently used to reconstruct the shape of a dense set of points. Once we get the polygon boundary from the occupied grip points we apply the Ramer-Douglas-Peucker line-simplification algorithm [Ram72] to get a polygon boundary with less line segments. A parameter specifying the degree of simplification is required which we shall denote as ε. We will discuss the sensitivity to these parameters in Section 5.2.2. After these two steps the polygon segments provide convex obstacles sets that can be processed to compute the Voronoi Diagram.

Once the Voronoi Diagram is computed we proceed by computing the split

points for each vertex and thereby its directional weights. We also associate the endpoints of a minimum length blocking line to edges and the length of this line becomes the block weights. Given the weights we apply the contiguous algorithm for trees described in Section 3.5 as well as the minimum spanning tree based cycle blocking. This gives us a strategy in form of a sequence of vertices. We compute the overall cost of the strategy as well as the cost of clearing the tree without considering the cycle edges.

From here on it is but a small step towards the actual motion of the robots. Given the sequence of vertices we can construct a sequence of moving lines. The first vertex is cleared by blocking one edges and clearing it coming with a new sweep line starting from that edge, leading to blocks on all its edges. From there on every next vertex is a movement of one blocking line towards the critical point for the direction the line is coming from. These continuously moving lines can be followed by a team of robots as seen in Section 4.1.1. To account for localization and navigation errors robots can be spaced with sensors overlapping as seen in figure 5.10. The δ parameter also helps to offset possible approximation errors during the conversion of the grid map into a polygon.

5.2.2 Experiments

To validate the implementation of the algorithm from Section 5.2.1 we ran it in a variety of configurations on the grid maps from Section 5.1.6. Figure 5.11 shows the map after it was processed by computing its α-shape and simplifying the polygon boundary. We shall call this map UCM map. The second map is obtained from the Radish online robotics data repository [HR03] sdr_site_b data set. We will denote it as SDR map. It is shown after processing in figure 5.12.

We compare our algorithm to the one used in Section 5.1.6 which also extracts

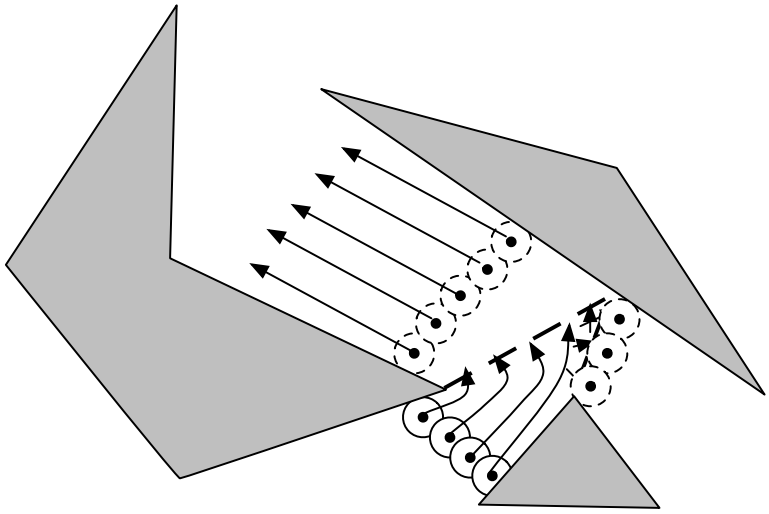


Figure 5.10: Robots following a sweep line with δ overlap and splitting into two sweep lines at a critical point. Robots with solid disks are moving towards future positions marked as robots with dashed disks. Note that the four robots require additional 4 robots to reach the dashed positions.

surveillance graphs. Therein the very crude sweeping routine does not acknowledge complexities in the environment within a vertex. Recall that therein the strategies are computed with the hybrid algorithm. These generally have lower cost than contiguous strategies, but they do not satisfy contiguity and can hence not be used for a Line-Clear approach in which a contiguous sweep schedule is being constructed. Although one could attempt to extend the approach from Section 4.2 and also use it to compute non-contiguous sweep schedules, but this remains subject to further work.

The evaluation of the algorithm from Section 5.1 included the cost for strategies on the generated tree denoted by ag. Let ag_r denote the cost of strategies extended to the entire graph. Since all non minimum spanning tree were assumed blocked in Section 5.1 we set ag_r to $ag + b_c$, where b_c is the cost for all

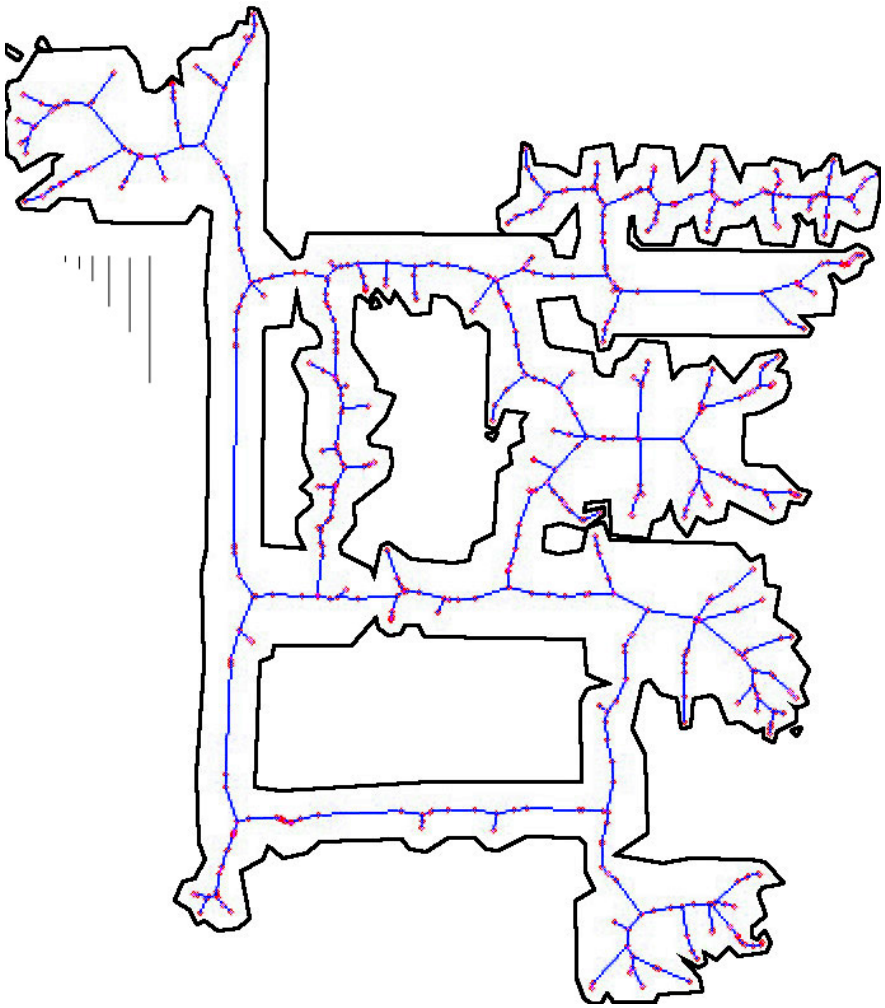


Figure 5.11: UCM map with borders thickened for illustration purposes. The graph is embedded with thin lines as edges inside free space. Vertices are small circles. The map is a polygon map obtained from the original grid map from fig. 5.8 after applying the α-shape and line simplification with $\alpha = 10$ and $\varepsilon = 3$. Distances are measured in pixel. To illustrate scale six horizontal lines of length 5,10,20,40,60 and 100 pixel are added.

cycle edges. Let b denote the number of cycles. In our case, ag_r denotes the cost for only blocking cycle edges when needed and corresponds to the actual

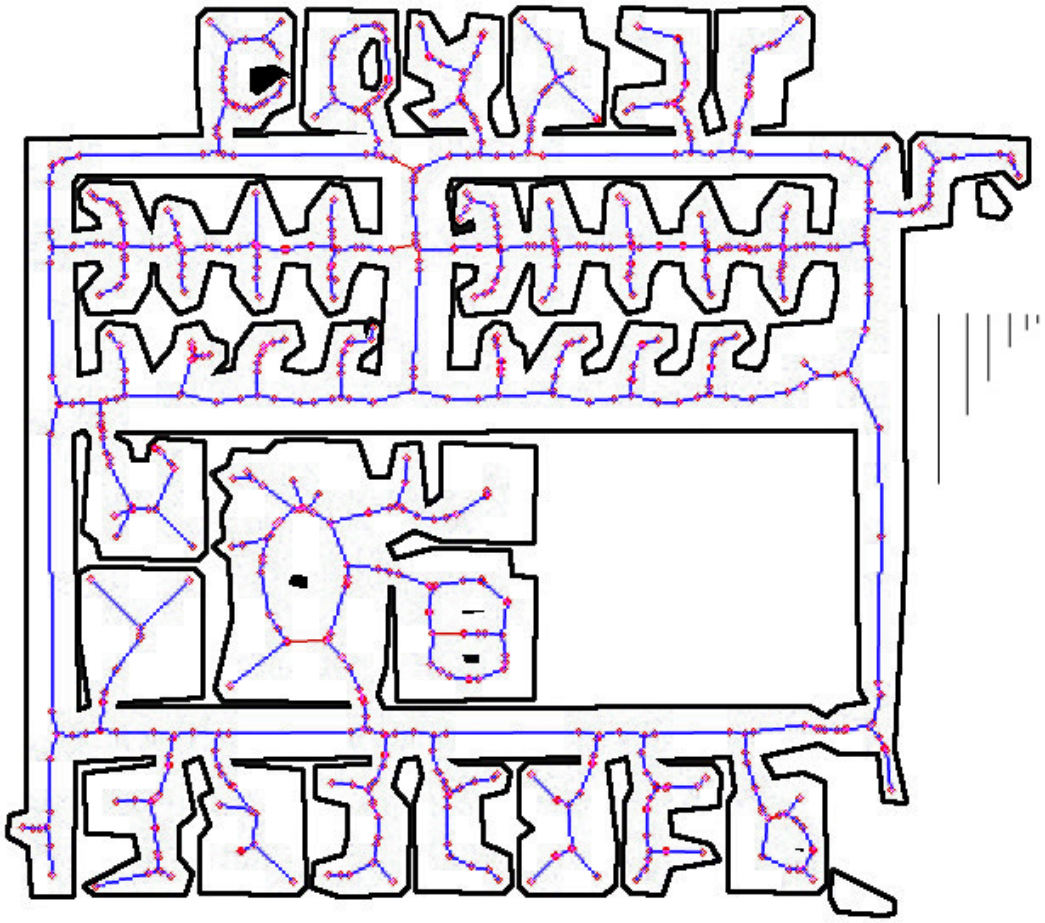


Figure 5.12: SDR Map with borders thickened for illustration purposes. The graph is embedded with thin lines as edges inside free space. Vertices are small circles. The map is create from the original grid map from fig. 5.7 exactly as fig. 5.11.

number of robots needed to clear the environment by following the sweep lines associated to vertices and edges. Write r for the sensing range in pixels and δ for the parameter adjusting for errors.

An accompanying video shows a sequence of block and split lines in the UCM map ($\epsilon = 3$, $\alpha = 10$, $\delta = 2$, $r = 40$) as red and green, respectively. Associated

cost for lines is show in red for blocks, green for split lines and blue for block lines due to cycle edges. At each step part of the robot team is moving from a red block line to a green split line. Counters on the top left show the clearing step, current cost and maximum encountered cost.

Table 5.2 summarizes the results from Section 5.1 for comparison purposes. Of importance are primarily ag_r and ag as a function of r. Tables 5.3 and 5.4 present

	UCM map				SDR map			
r	ag_r	ag	b	b_c	ag_r	ag	b	b_c
5	73	58	3	15	67	36	8	31
10	37	28	3	9	36	19	7	17
20	20	14	3	6	19	9	7	10
40	11	8	3	3	14	6	8	8
60	9	6	3	3	11	5	6	6
100	7	4	3	3	10	4	6	6

Table 5.2: Summary of the experimental results from [KC08a].

results from executing the algorithm from Section 5.2.1 on the UCM and SDR maps. Table 5.3 can be compared to table 5.2 since $\delta = 0$. Despite the additional constraint to require contiguous strategies the new approach outperforms the one from Section 5.1 on the UCM map for sensing ranges from 5 to 60. Also on the SDR map the new approach produces better strategies on the tree (ag) for sensing ranges 5 to 60. The similar performance for sensing ranges at 100 may result from the fact that the method from Section 5.1 ignores the complexities of the environments when merging it into a graph with 13 to 14 vertices each of which is swept with the rectangular sweep.

It is also interesting to note that the new approach removes the problem en-

countered in Section 5.1 in which two vertices are merged leading to the removal of a cycle edge. The cycle in the environment is then inside the vertex and effectively ignored. The new approach continues to capture the geometric complexity without such heuristics and the number of cycle edges remains relatively constant. There is, however, a slight variation resulting from the processing of the grid map. The produced alpha shapes are not deterministic and slightly different alpha shapes may lead to a different surveillance graph which can also lead to a slight variation in the number of cycles, as well as slight variations in the tree and hence the strategy cost. Usually these cycles are degenerate[1], resulting from the particular computation of the Voronoi Diagram as the dual of the Delaunay Graph. These, however, do not pose a problem neither for computation nor for the execution of the strategy.

As expected, increasing δ leads to slightly more costly strategies as seen in table 5.3. Obviously, robots with small sensing ranges are affected more than those with large sensing range since the ratio of δ to r matters. But already a relatively small δ of 2 leads to cost increases across all sensing ranges.

Table 5.4 shows the effect of varying the degree of simplification for the Ramer-Douglas-Peucker algorithm. Setting ε from previously 7 (in table 5.3) to 3 leads to no significant differences in ag. The differences in ag_r can be explained by an unfortunate selection of cycle edges due to a different selection via the minimum spanning tree. Also note that degenerate cycle edges disappeared for the SDR map.

All in all the experiments show that the algorithm is simple to implement and robust while maintaining consideration for the geometric complexities. The only approximations are made during the polygon creation and simplification which

[1]Degenerative in this context means that two Voronoi vertices are at exactly the same point in the environment.

		$\delta = 0$				$\delta = 2$			
Map	r	ag_r	ag	b	b_c	ag_r	ag	b	b_c
UCM	5	56	46	3	15	90	75	3	24
UCM	10	32	24	3	9	36	30	3	10
UCM	20	18	13	3	6	20	14	3	6
UCM	40	9	7	3	3	10	8	3	3
UCM	60	9	6	3	3	8	6	3	3
UCM	100	7	5	3	3	6	4	3	3
SDR	5	45	31	14	48	72	47	14	73
SDR	10	26	16	14	27	30	18	15	31
SDR	20	14	8	14	17	15	9	15	18
SDR	40	11	6	14	11	10	6	14	11
SDR	60	10	5	14	10	8	5	14	10
SDR	100	9	5	14	10	7	4	15	10

Table 5.3: Summary of the experimental results with $\alpha = 10$, $\varepsilon = 7$. Note that some MST-edges are degenerate and have 0 weight.

are marginal in comparison to the heuristics employed in Section 5.1.

5.2.3 Discussion and Conclusion

This section provided an improved method to relate surveillance graphs to the planar environment they represent by using the methods discussed in Section 4.2. This leads to an efficient extraction of surveillance graphs from any robotic grid map or polygonal environments. Regarding the computation of Graph-Clear strategies it is apparent that the reduction to a tree is not entirely satisfactory and motivates the study of approximation algorithms to the problem on the graph.

	UCM map				SDR map			
r	ag_r	ag	b	b_c	ag_r	ag	b	b_c
5	97	76	3	24	73	50	10	69
10	38	30	3	9	30	19	10	28
20	18	14	3	6	15	9	10	15
40	10	8	3	3	10	6	10	10
60	8	6	3	3	7	5	10	10
100	7	5	3	3	11	5	10	10

Table 5.4: Summary of the experimental results with $\alpha = 10$,$\delta = 2$, $\varepsilon = 3$. Note that degenerate MST-edges from table 5.3 do not appear here.

From a practical point of view the presented vertex sweeps and edge blocks can be executed by a robot team which follows the lines that are associated with the sweeps and stops at the blocks. Here a wide body of literature is available and a control theoretic approach in the spirit of [BCM09] is a viable direction to pursue. Robots have to follow a moving boundary which can be achieved with an event-driven asynchronous robotic network as described in Chapter 6 of [BCM09]. Therein issues such as communication are also addressed. Furthermore, the robot team requires some coordination to assign paths that result from following the lines to individual robots. Here performance parameters such as time and travelled distance can start to play a role. A robot which is not needed for a few vertex sweeps could already travel to the vertex where it is needed next, speeding up the overall execution. Also an interesting question is whether a minimalist approach such as in [SRL04] can be employed for following sweep lines and we shall investigate this direction in Chapter 6. Another important direction is the consideration of probabilistic sensing and errors in control. In Section 3.8 Graph-Clear is extended to a probabilistic sensing model. The algorithm therein can be

combined with the δ parameter and together give a probabilistic guarantee that no intruder passes through a sweep line. Failure to detect may result from the sensor or an errors in following a sweep line which can open up a gap. Increasing δ and using more robots reduces this probability.

CHAPTER 6

Applications and Experiments

In previous chapters we presented the pursuit-evasion problems Graph-Clear, (Chapter 3) and Line-Clear (Chapter 4). The purpose of this chapter is to discuss how to apply these ideas and methods for the control and coordination of a robot team. The notion of clearing a vertex, blocking an edge or covering a sweep line are rather abstract and we are going to show how to interpret and implement them with examples. In general, the specifics of the given hardware will determine what problems will have to be solved in order to apply Graph-Clear or Line-Clear. Yet, there are a few elements that can be useful for a variety of scenarios such as the methods presented in Chapter 5 that automatically generate surveillance graphs given a sweep and block routine.

In Section 6.1 we discuss experiments with two PioneerP3AT that are intended as a proof of concept. They demonstrate an approach to convert Graph-Clear strategies into robot paths which can then be followed by robots in proper order as determined by a Graph-Clear strategy. These exploratory experiments were carried out prior to the development of Line-Clear and inspired the basic idea of allowing movement of a frontier forward which also lead to the modified variant of Graph-Clear described in Section 3.9. A more significant contribution and starting point for further work is presented in Section 6.2. Therein we extend the principles of Line-Clear to a scenario in which no map is available. We design a hybrid algorithm that clears an environment with robots that can only

communicate and sense target locally, follow obstacle boundaries, and sense their neighbors positions. The robots do not have to build a metric map of the environment, but they do build a topological map represented by the surveillance graph. Such a map, however, is only useful to the team and not to an individual robot since it describes the movement of the lines. We conclude with Section 6.3 and outline further work.

6.1 First Experiments with Simple Sweeps and Two Robots

In this section we describe the first application of Graph-Clear to experiments carried out with two Pioneer P3AT robots. These experiments are primarily intended as an illustration and proof of concept to demonstrate particular implementations for sweeping and provide a first application of Graph-Clear. They were carried out prior to the development of Line-Clear and it is interesting to note that the resource limitation of the two robots lead to the consideration of modifications for Graph-Clear, such as the relaxation of having to block all edges of a vertex while it is being swept (see Section 3.9). Furthermore, the concept of pushing forward the frontier between contaminated and cleared regions sparked the development of Line-Clear.

6.1.1 Extracting Surveillance Graphs

In Section 5.1 we introduced a method to automatically extract surveillance graphs from an occupancy grid map by using its Voronoi Diagram. The extraction is based on the Generalized Voronoi Diagram [CB95], which we shall denote as Voronoi Diagram. Once the Voronoi Diagram is computed the method selects minima of the clearance function defined on the edges of the Voronoi Di-

agram. In our case, the selection of minima proceeds by choosing a point on the Voronoi Diagram with the smallest clearance and then choosing each next point with smaller clearance that is at least a certain distance away when moving on the Voronoi Diagram. At a resolution of $0.032m$ per grid point this distance was chosen to be $3.2m$, i.e. in the grid map no minima can be closer than 100 grid points. This value determines the size of the vertices that will be extracted in the first step. For each minimum an edge for the surveillance graph is created. More precisely, we use the line connecting the two closest obstacles to the minimum on the Voronoi Diagram edge to represent the border between two vertices of the surveillance graph. To improve the surveillance graph adjacent vertices are merged under the criteria given in Section 5.1. Fig. 6.1 shows the Voronoi Diagram and the selected minima after merging for the environment used in the robot experiment in Section 6.1.3. To obtain a strategy, we use the contiguous algorithm from Section 3.5. Being a sequence of block and sweep actions to be executed on a graph that is embedded in a plane, the strategy implicitly determines which paths the robots follow as presented in Section 6.1.3.

6.1.2 Implementing Surveillance Graph actions

The abstractions of the sweeping and blocking actions in a surveillance graph obviously require an implementation to enable a real application. The choice of implementation is dependent on the mobility of the robotic platform, the type of sensor, and properties of the environment. Here, we will present particular sweeping and blocking implementations and discuss their limitations. In general, the sweeping and blocking implementations should also influence the extraction of graphs from the map, i.e. if the implementations requires certain assumptions to be satisfied, then the extraction has to ensure that no vertex or edge is created

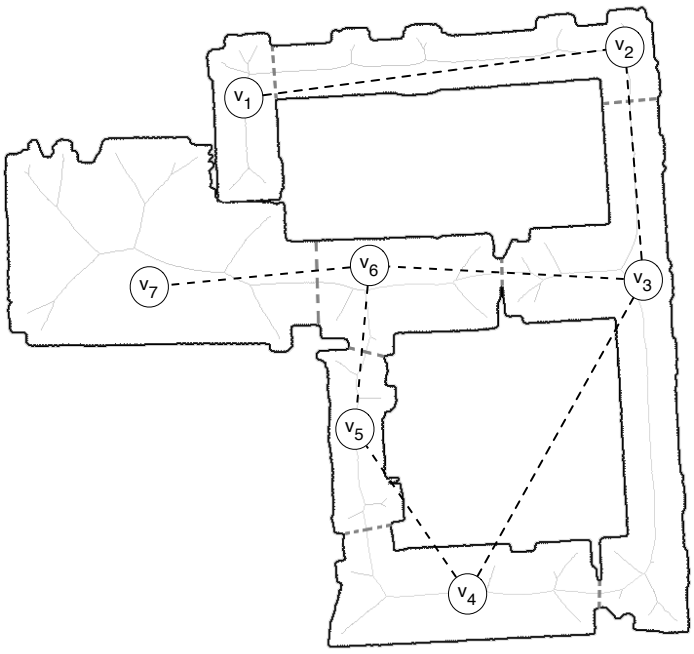


Figure 6.1: Map of part of the Science and Engineering building at UC Merced built with a SICK laser on a Pioneer P3AT and the gmapping software [GSB]. The grey thin lines show the discrete approximation to the Voronoi Diagram, the thicker black lines the boundary of the environment, and the thick dashed grey lines show the boundary of regions associated to different vertices. Vertices are circles with edges as black dashed lines.

which cannot be cleared or blocked by the respective implementation.

For the blocking actions, the extraction method from Section 5.1 associates straight lines between two obstacle points with an edge of the surveillance graph. To block the edge the robot needs to be positioned such that the associated line is covered by the sensor. With omnidirectional sensors this is rather simple as one can position the robot in the center of the line or when multiple robots are needed one can position them all along the line. With a restricted field of view

such as with off-the-shelf webcams, one has to compute positions such that the edge line is within the area covered by the camera and make sure such positions are feasible, i.e. in free space with line of sight of the appropriate region. Due to the construction of the surveillance graph based on a Voronoi Diagram, one can ensure that such feasible positions exists. This is the case when the field of view of the camera is larger than $\pi/2$. Recall that the two obstacle points that define the line are the closest two such obstacles. Hence in a circle with a radius equal to the clearance at the minimum point on the Voronoi edge we have no other obstacle points (see figure 6.2) that could obstruct the sensor. For our experiments we will position robots on the line.

Sweeping is significantly more complicated than blocking. In general a sweeping action is executed on a vertex while all its edges are blocked. This requirement arises naturally to prevent recontamination before, during, and right after the clearing. This allows the use of any implementation for a sweeping action that simply ensures that any intruder present in the region for the vertex will be detected supposing none leave or enter (which would be detected by the blocking). There is, however, an alternative approach to this procedure which would impose additional requirements on the sweeping implementation, but potentially lead to a reduction in the total number of robots needed for the clearing. We will first illustrate the basic idea that gave rise to the modified Graph-Clear variant presented in Section 3.9 and then describe our particular implementation.

Consider the clearing of any vertex in a contiguous strategy. If the vertex is not the first, then at least one edge will be blocked before the clearing, and if the vertex is not the last, then at least one edge will have to be blocked right after the clearing. In fig. 6.3 we show this idea for the simple case in which exactly one edge is blocked before, and one after the clearing. It is easy to see

that for such a sweep one does not have to require that all edges are blocked during the sweep. To compute paths for such sweeps automatically we consider a surveillance graph created by the Voronoi Diagram-based method from Section 5.1. We will use the Voronoi Diagram to compute paths from the blocked edges, denoted as starting edges, to edges that will have to be blocked after the sweep, denoted as stopping edges. The strategy computed by the contiguous algorithm determines which edges are the starting and stopping edges. For vertices with degree two this approach is trivial, since we simply move the robots along the Voronoi Diagram from the starting edge towards the stopping edge. For vertices with degree three we can have either one starting edge and two ending edges or vice versa. In either case, at least two robots are required for the sweep and they will meet at a Voronoi edge. To avoid collisions we compute paths along the Voronoi Diagram with an offset. Fig. 6.6 shows these paths for the vertices of the environment from fig. 6.1 for two robots with a simple offset either left or right with respect to the direction of the movement.

For the purpose of our experiments we only need to consider vertices up to degree three. It is, however, possible to generalize this procedure and consider a vertex v with degree larger than 3. Let s be the number of starting edges and $e = degree(v) - s$ the number of stopping edges. We can now construct a tree using the starting and stopping edges as leaves and Voronoi edges as internal nodes representing the possible paths between the starting and stopping edges as seen in fig. 6.4. The problem then reduces to finding an appropriate path in the tree. The primary assumption made is that there are no cycles in the Voronoi Diagram within one vertex which can be ensured during the partitioning. Another assumption is that there are no branches of the Voronoi Diagram within a vertex that do not lead to another vertex, i.e. small pockets are ignored as seen in fig. 6.6.

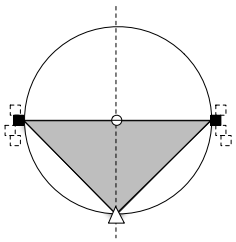


Figure 6.2: Guaranteed blocking positions for blocking using a camera with $\pi/2$ opening angle. The small triangle is the sensor, its coverage is grey and the obstacles associated to the minimum clearance value on the Voronoi edge (dashed line) are black squares. The other obstacles are squares with dashed boundaries. The circle shows the guaranteed obstacle free area.

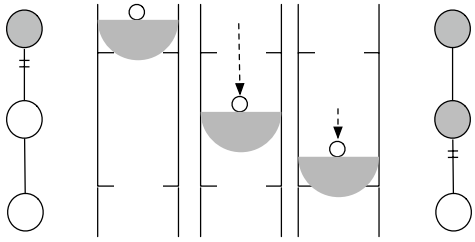


Figure 6.3: Example of an improved sweeping implementation. On the left is the graph representation with cleared parts in grey and blocked edge with a stroke. The center shows how a robot sweeps, ensuring that no intruder can enter. Sensor coverage is shown in grey. On the right we have the status of the graph after the sweep.

For the coordination we have to make sure that the robot team executes the blocking and sweeping actions with proper timing. The coordination that is required is relatively simple. If a robot is at a starting edge of a sweep task, then it is by default assigned to the sweep for the vertex. For sweeps with both robots they both are placed at the first pose of the sweep and once both robots arrive they follow the path until they reach the Voronoi edge where their paths either split or merge. They wait in the respective position until both robots signal that they are ready and then proceed with the second part of the sweep until they reach the stopping edges. A robot not participating in a sweep is blocking an edge and waits until a next sweep with two robots is executed. This simple

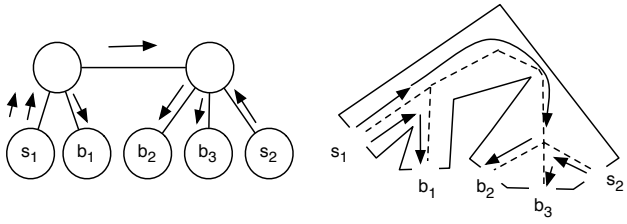


Figure 6.4: A generalization of the vertex sweep implementation to vertices of degree larger than 3. The tree on the left corresponds to the environment on the right with the Voronoi Diagram as dashed lines. Each arrow indicates the movement of one robot. The leaves s_i and b_i are starting and ending points respectively.

coordination scheme already suffices for our particular application since most of the coordination is implicitly contained in the strategy.

6.1.3 Experiment Design

Real robot experiments with two Pioneer P3AT with a SICK PLS200 laser as seen in fig. 6.5 were carried out to demonstrate applicability of the surveillance graph framework using the implementations discussed in the previous section. Prior to the experiments the grid map was created using one of the robots and the Gmapping software [GSB]. The resulting map was threshold to give a binary occupancy map as done in [KC08a] and then the surveillance graph was extracted as described in Section 6.1.1. On this graph each robot computed the contiguous strategy using the algorithm described in Section 3.5. Then the robots computed the sweeping paths for each vertex. To control the robots we used Player 2.1.0 [BCG05] using a wavefront algorithm for path planning and a vector field histogram for local navigation. For localization we used an adaptation of the Monte Carlo localization algorithm from [FBD99]. As a map we used the

Figure 6.5: The two Pioneer P3AT equipped with a SICK PLS200 laser.

thresholded 700 by 700 map which was also used to construct the graph and is hence an imprecise approximation. Navigation algorithms were used off-the-shelf with very little configuration, i.e. only the thresholds for the safety distance were changed to enable the robots to navigate through narrow door openings. The communication uses an ad-hoc wireless network with the robots broadcasting their status. The status message are straightforward and contain the robot id, the current sweeping step and whether they completed their assigned path. Both robots are starting at vertex v_1 (see fig. 6.1) at a known location and robot one is assigned the sweep of the starting vertex. Given that we have only two robots we did not incorporate advanced coordination to avoid collisions and we can do with a simple assignment of goal poses by discretizing the paths from the sweeps. These then serve as input into the path planner.

The main goal of the experiments is to demonstrate that the presented methods suffice to control a robot team which will then clear the environment according to the computed surveillance graph strategy with readily available and accessible methods for localization, navigation, and communication. Sensors for the actual detection of targets were not used, but a robot following the strategy could detect a target with an omnidirectional sensor with a sensing range of around 8m, i.e. on the computed paths they would detect a target, if present. When trying to detect people in the building this could be achieved by mounting multiple cheap web-

cams on the robot and attempting to detect people in the images. Alternatively, a remote operator may wish to view the video feed and mark targets.

6.1.4 Results and Discussion

We conducted three batches of experiments. One initial set of 4 experiments was performed to observe the robots and record unanticipated problems. A second batch, consisting of a single experiment, was carried out with a different surveillance graph extracted from the map. The third batch was the final batch to test robustness of the approach to repeated execution and contained three successful experiments in which the robots followed their paths as required. The first batch revealed that the localization performed sufficiently well throughout the experiments and enabled the robots to follow their trajectories without colliding with the walls. Problems with the planner were encountered when navigating through narrow openings due to the default security distance. Decreasing it lead to both robots getting very close when paths were close, but still enabled satisfactory overall performance. For the second batch a slightly different map was used with the door between vertex v_3 and v_6 opened further. This led to a different graph as different minima were selected the vertices were merged differently. Fig. 6.7 shows the resulting graph which could in principle be cleared with two robots, but in this case the contiguous algorithm computed a strategy which required three robots. This is due to the fact that it does not consider the improved sweeping strategy. The two robots in the experiments attempted to execute the strategy and followed the paths in fig. 6.7 leading to recontamination when the robot blocking the bottom edge abandons its position. The third batch consisted of three consecutive experiments in which the robots followed the computed path satisfactory without recontamination. A video of the robots moving through the environment is avail-

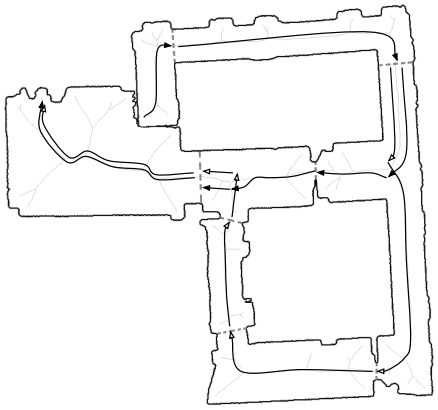


Figure 6.6: The paths computed for all vertices in the environment from fig. 6.1. Paths from one robots are shown with a black and the other with a white arrow.

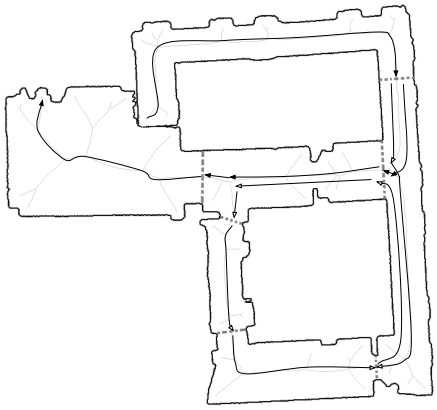


Figure 6.7: The paths computed for all vertices in the environment from fig. 6.1. Paths from one robots are shown with a black and the other with a white arrow.

able at https://robosrv.ucmerced.edu/public/videos/experiment1.mov. Please note that the video is mirrored due to the recording software.

6.1.5 Conclusion

The implementations of sweeping and blocking actions and the real robot experiments demonstrate that Graph-Clear can provide a practical benefit for the distributed coordination of a team of robots for target detection in realistic environments. The implementations are rather simple and worked with currently available mapping, localization and path planning algorithms. Furthermore, additional improvements for collision avoidance and local path coordination can readily be incorporated. The current limitations of the experiments are that the actual detection of targets with a low cost sensor was not considered and that scalability with respect to the environment and the team could not be demonstrated

due to limited resources. The main purpose of these experiments, however, was an early proof of concept implementation which due to the resource constraints at the time led to the development of Line-Clear.

6.2 Line-Clear without Maps

In this section we revisit the ideas for Line-Clear from Chapter 4 and design a distributed algorithm that moves robots on lines to clear an environment without knowing its map. An application of Line-Clear with a known map is now straightforward given the results of the previous sections. Without a map, however, we need to resort to a truly distributed algorithm that explores the environment as it is cleared. For this we shall assume minimal requirements for the robots and not impose that they are able to build a map. We will first present the algorithm and then present results from an implementation that clears simply-connected environments.

For this section we assume that robots are holonomic, can detect targets within a range r_{detect}, and communicate with other robots within a range $r_{comm} > 2 \cdot r_{detect}$. Furthermore, they can sense obstacles and follow the boundary of an obstacle at distance $r_{obstacle}$ by sensing the tangent of the boundary. Finally, they can detect robot neighbors within a range r_{detect}, and determine their relative position. The desired distance between a pair of robots on a line is denoted by $r_{follow} \leq 2 \cdot r_{detect}$. The free space of the environment is denoted by $\mathcal{E} \subset \mathbb{R}^2$ and is bounded. Each of the n robots is assigned a unique identifier, i.e. an integer in the range $1, \ldots, n$. In the following, robot with id 1 is referred to as the first robot, and robot with id n is the last. All n robots are initially deployed together, i.e. their communication graph is connected [1]. Given that robots closer

[1] Bullo et al. in [BCM09] define this concept rigorously for robotic networks. For our

than $r_{follow} < r_{comm}$ can communicate, we assume that robots forming a line can maintain a set of shared variables while operating. In the following we will describe the main steps involved in the algorithm. These center around the ability of the robot team to move forward on a sweep line, to split a sweep line into two new sweep lines, to search for new obstacles, and to backtrack to previous states and choose to move a different sweep line forward. An additional step to create the first sweep line may be necessary and this is shortly discussed in Section 6.2.1.

Once a sweep line and some cleared space exist the robot team can choose a line to move it forward into contaminated space by following the obstacle boundaries. A sweep line is followed by a reserve of robots that are not needed to cover the sweep line. These jump in when the length of the sweep line increases and drop out when it decreases. These two events are recorded with a surveillance graph to account for the cost of the movement. If the robots hit an obstacle they will split the sweep line into two new sweep lines and choose one of them to continue with. This event is also recorded in the surveillance graph. If the line grows too long and no more robots are available it moves backwards to have some robots drop out into the reserve and then initiate a search procedure that attempts to find new obstacles close by. If this procedure fails the line moves further backwards and backtracks in the surveillance graph to find a vertex that is associated to a sweep line that has not yet been moved forward. We shall denote such vertices as unexplored since they can receive new edges as their associated sweep line moves forward. Hence, unexplored vertices are all vertices for which the robot team does not know the cost of extending a sweep line further. Sweep lines dissolve when the two robots on each end of the line meet. In our implementation two robots meet if their distance is less than the desired following

purposes each robot is simply a node in a graph and if two robots are within communication range their nodes receive an edge.

distance r_{follow}, but in general any small threshold suffices. One example of this happening is when the line is moving through a corridor orthogonal to the wall on the left and right. Once it hits a wall at the end of the corridor the robots at the left and right end of the line will follow this wall towards the center of the line and meet. Fig. 6.8 shows these steps and their transitions in a diagram. The basic principles of this procedure are similar to Line-Clear since it clears an environment by moving lines forward. But there are three key differences which are as follows. First, instead of computing low cost split positions on a third obstacle the approach has to rely on the third obstacle that is encountered while moving the line forward. This can lead to higher costs. Second, moving a line forward cannot rely on the well defined movement between blocking and splitting positions as done for Line-Clear since new block and splitting positions are not known in advance. Hence, they have to move heuristically attempting to go as far as possible. Third, since the graph is not known in advance we cannot compute Graph-Clear strategies for the entire environment. This is unavoidable when exploring and clearing simultaneously. As a consequence the robots may have to recontaminate parts of the graph to get it into a state from which better strategies can be executed. Note that in the worst case the discovery of the last vertex will require a recontamination of the entire environment. This happens exactly when all Graph-Clear strategies that require the given number of robots or less have to start at that last vertex. Such recontamination can be avoided by adding more robots than strictly necessary and this represents a tradeoff between finding the lowest number of robots needed and avoiding recontamination during the exploration. We shall now discuss the details of the algorithm and present the routines it is based upon.

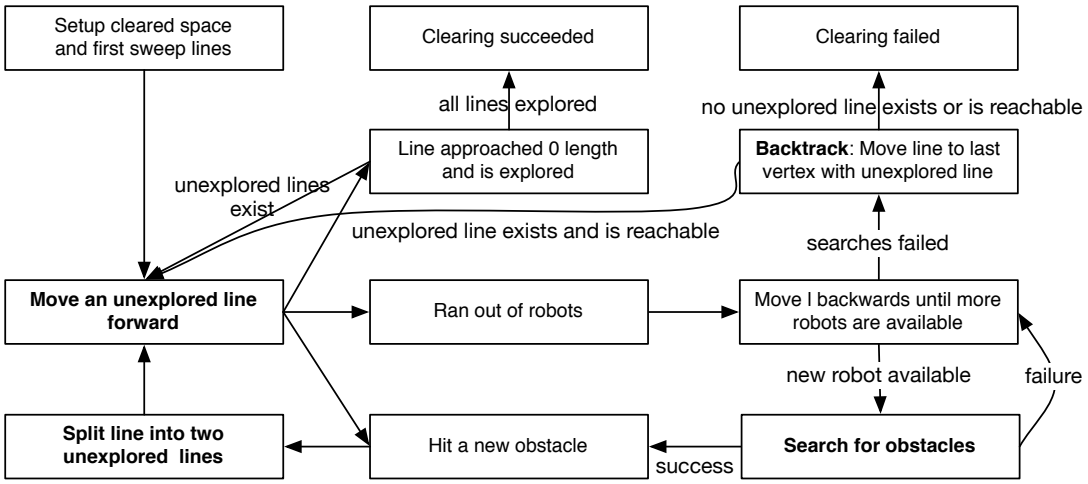


Figure 6.8: A diagram showing the high level states of the algorithm.

6.2.1 Bootstrapping

Before the Line-Clear components of the algorithm can proceed the robots need to arrange themselves into a line formation. Depending on the scenario the robots may already be in a line formation at a home base that is considered clear in which case the bootstrapping is already done. Otherwise the robots will have to find two obstacle boundaries between which to span a line. There are a multitude of ways of achieving this and we will shortly present one.

Suppose robots are not deployed in the form of a sweep line. Since the communication graph is connected the team has the capability to form a chain with robot i following its neighbor robot $i-1$ and robot 1 leading the chain. Robot 1 is then moving into an arbitrary direction until hitting an obstacle boundary. Next, the last robot of the chain moves to find a second obstacle boundary. All other robots follow while maintaining a chain, so the movement of the last robot is restricted. Fig. 6.9 shows the reach such a chain in part a). Part b) of Fig. 6.9 shows that if there is no obstacle boundary reachable within the half circle with

a radius that 12 robots can cover then the environment cannot be cleared.

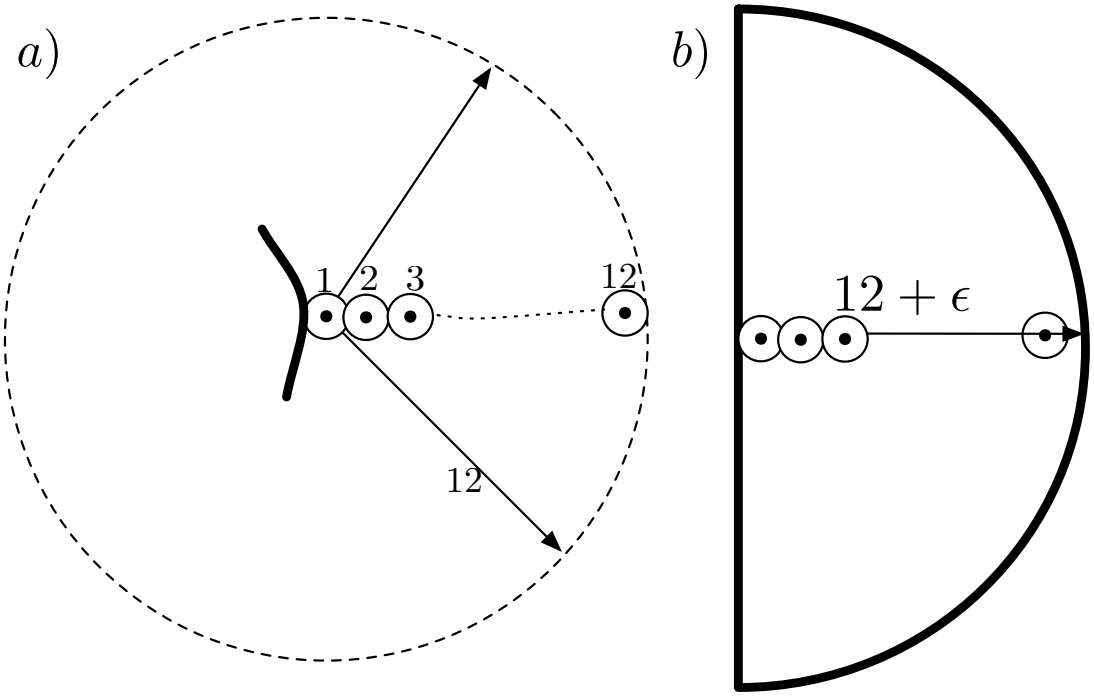


Figure 6.9: The maximum reach given 12 robots.

Finally, the one sweep line that is now set up needs to either move into a state in which two sweep lines can be set up or one of its sides can be declared cleared. The former is only possible if the length of the sweep line in terms of robots is smaller or equal to $\frac{n}{2}$ and then two parallel sweep lines can be created, effectively assigning the space in between them as clear. If the first sweep that is set up is longer than $\frac{n}{2}$ robots it can move forward until it shrinks to $\frac{n}{2}$. If this is not possible to either side, then the environment cannot be cleared with the algorithm starting with a sweep line between the two obstacle boundaries encountered first. In this case the first robot has to find a different starting boundary. Moving the line can involve obstacle searches that extend the reach of the robots to the maximum while maintaining a sweep line between the two obstacles. We will discuss the details of the obstacle searches later.

In our implementation we use a simple bootstrapping procedure that moves robot 1 towards an obstacle boundary and aligns all robots on a line orthogonal to the tangent of the boundary until another obstacle is encountered. Then one side of the line is declared cleared, as a homebase, and the algorithm starts by moving the first and single sweep line forward.

6.2.2 Moving a Sweep Line

Once a line is chosen to move forward the two robots on its endpoints control the forward movement by following the obstacle boundary. An example is seen in Fig. 6.11. The two end robots, denotes as the left line-leader and right line-leader, sense the tangents at the two obstacle boundaries between which the line of robots is spanned. From these tangents one can determine whether a forward movement will shrink or grow the line. Let us write $b_1(t)$ and $b_2(t)$ for the curves defining the obstacle boundaries with $b_1(t_0)$ and $b_2(t_0)$ being the points between which the line is spanned. At these points let the tangent along the obstacle boundary as sensed by the robots be given by $\hat{T}b_1(t_0)$ and $\hat{T}b_2(t_0)$ respectively. The relative angle between tangents and the line determines whether a forward movement on b_1 or b_2 shrinks or grows the line. Given this information, the rules for moving the line are straightforward. If both tangents determine a decrease in length, then both line-leaders follow their boundary. If one is decreasing, then only that robot moves. If both are increasing, then only the one with less increase moves. Since robots can communicate with their neighbors on the line and sense their relative position we can assume that they all have access to the relative positions of the line-leaders and can use this information to position themselves on the line between them. As seen in Fig. 6.11, robots that are currently not needed to cover the line, denoted as reserve, just follow the robots of the line and

join in when the length of the line increases beyond a length that the current robots that are part of the line can cover. This method is essentially a local gradient search for short lines moving forward and as such is susceptible to local minima. Fig. 6.10 shows how a line gets stuck in the analogue of a local minimum and grows larger than it needs to if the robots had global knowledge about the structure of the environment.

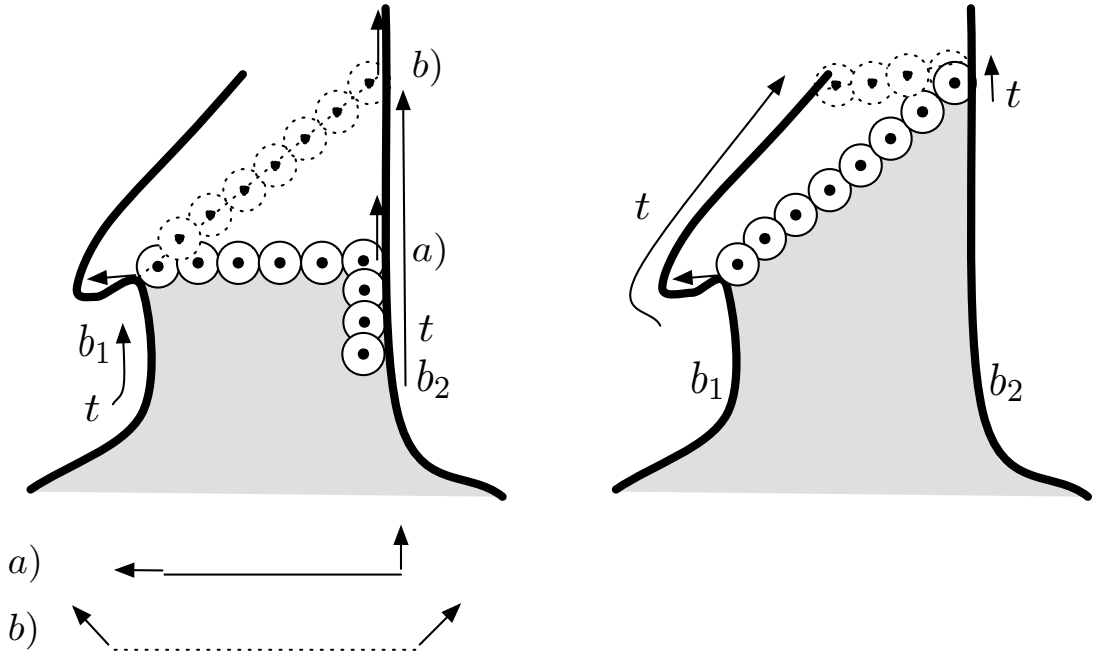


Figure 6.10: An example of how robots moving on a line cannot find the optimal way of moving it forward and the left side stops at position a). Yet, as the right side proceeds the left and right tangents approach the same values at position b). At this point the left side will move again and the line will shrink again. The bottom of the figure shows the local view from the robots of the tangents.

As the line moves forward it also keeps track of events that are occurring. These events trigger state transitions for the whole system or simply record the costs of moving by constructing a surveillance graph. The line moving routine

can be expressed in the form of three algorithms. Two robots execute execute Algorithm 8. These are the left and right line-leaders following obstacle boundaries b_1, b_2 by calling $Line\_Leader(b_1, b_2)$ and $Line\_Leader(b_2, b_1)$. Both have access to the shared variables $\hat{T}b_1(t)$ and $\hat{T}b_1(t)$, i.e. each others sensed obstacle tangent, in form of a vector. Algorithm 9 is executed by all robots that are following the line between the two leaders while Algorithm 10 is executed by robots that are following as part of the reserve. All robots executing these algorithms are part of one connected communication graph and have access to the following shared variables and functions:

1. r_{all}: the number of robots associated to the line, i.e. the number of robots currently running the Algorithms 8,9, and 10.
2. r_{line}: the number of robots running only Algorithms 8 and 9.
3. r_{left} and r_{right}: positions of the left and right line-leaders relative to a shared coordinate system.
4. *position*: the position of the robot in the line counted from the left line-leader. The left line-leader is at position 1 and the right line-leader at position r_{line}.
5. a: a function that returns the smaller angle between two vectors.
6. $wait()$ simply leaves the robot stationary.
7. $move\_robot(to)$ moves the robot to point *to* given in local coordinates of the robots.
8. $move\_along(d)$ moves it in the direction d.
9. $follow\_line()$ lets the robot follow the line.

10. *sense_obstacles*() returns true if a new obstacle is encountered and sets shared variable *new_obstacle* to true.

11. *first_in_reserve*() returns true if the robot's id is the smallest amongst all those running Algorithm 10.

12. Four functions trigger events that are used later on:

    (a) *trigger_dissolve_line*(): line is no longer needed.

    (b) *trigger_search*(): start obstacle search procedure.

    (c) *trigger_line_shrinks*(): line is shrinking by one robot.

    (d) *trigger_line_grows*(): line is growing by one robot.

The events triggered are caught by Algorithm 13 introduced later.

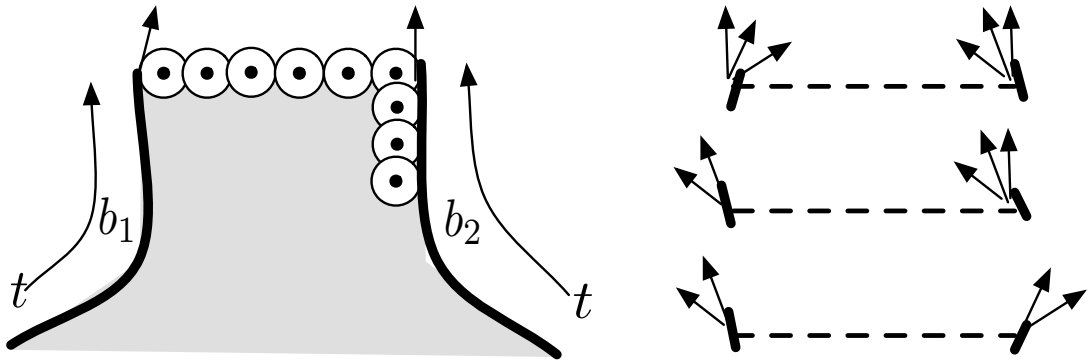


Figure 6.11: An illustration of the main line moving forwards to extend the cleared area marked in grey. On the right hand side are possible configurations for the tangent of the line. A tangent at an endpoint away from line leads to its length increasing while a tangent inwards leads to a decrease. The three cases depicted are one where 1) both sides lead to a decrease, 2) one side leads to a decrease and one to an increase 3) both sides lead to an increase in length. On the left figure some robots in the gray area serving as reserve are shown.

while $!new\_obstacle$ **do**

  $l \leftarrow r_{b_2}(t) - r_{b_1}(t)$

  if $a(\hat{T}l_{b_1}(t), l) < \pi/2$ OR $a(\hat{T}l_{b_1}(t), l) \leq a(\hat{T}l_{b_2}(t), -l)$ **then**

    $move\_along(\hat{T}l_{b_1}(t))$

  else

    $wait()$

  if $length(l) < r_{follow}$ **then**

    $trigger\_dissolve\_line()$

    return

  else if $length(l) > (r_{all} - 1) \cdot r_{follow}$ **then**

    $trigger\_search()$

    return

  else if $length(l) > (r_{line} - 1) \cdot r_{follow}$ **then**

    $trigger\_line\_grows()$

return

Algorithm 8: $Line\_Leader(b_1, b_2)$

while !$new\_obstacle$ **do**

  if $sense\_obstacles()$ **then**

    return

  $l \leftarrow r_{right} - r_{left}$

  $m \leftarrow (position - 1) \cdot \frac{r_{follow}}{length(l)}$

  if $m > length(l)$ **then**

    $trigger\_line\_shrinks()$

    return

  else

    $g \leftarrow r_1(t) + \cdot l$

    $move\_robot\_to(id, g)$

return

Algorithm 9: $Line\_Center()$

while !$new\_obstacle$ **do**

  if $first\_in\_reserve()$ **then**

    $l \leftarrow r_{right} - r_{left}$

    $m \leftarrow \frac{length(l)}{r_{line}-1}$

    if $m \leq r_{follow}$ **then**

      $trigger\_line\_grows()$

      $Line\_Center()$

    else

      $follow\_line()$

  else

    $follow\_line()$

Algorithm 10: $Line\_Reserve()$

6.2.3 Splitting a Line

Once an obstacle is encountered the robot team splits the line into two new sweep lines. This is similar to the split for Line-Clear in Chapter 4 with the exception that the split point is not necessarily the one at lowest cost. The simplest way to split the line is to move one robot from the reserve onto the same spot as the robot that encountered the obstacle and then declare the two robots the left line-leader and right line-leader of each of the two new lines. This, however, is also the most costly. The robots can attempt to reduce the cost of the split by moving one of the line-leaders or the robot that hit the new obstacle along their respective obstacle boundaries. Again, this can involve a reduction in line length via a local gradient descent. Fig. 6.12 illustrates this.

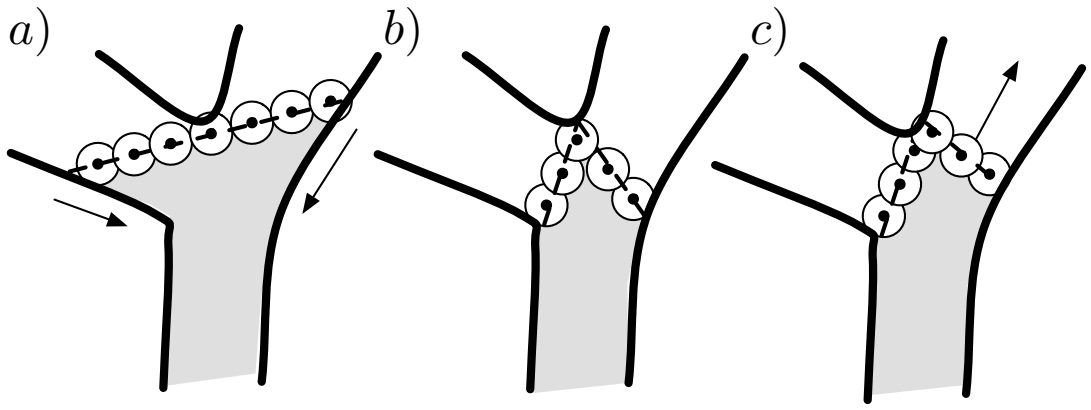


Figure 6.12: In this example a sweep line hits an obstacle and subsequently moves the two line-leaders into the direction for which the line lengths decrease. This reduces the number of robots needed by two. Then the sweep line splits, needing one additional robot for a total of seven robots. Robots that are in the reserve are not shown.

There are a few open questions with regard to splitting a line that call for more theoretical investigations. Primarily the question whether we can find an optimal

split point with a given number of robots would be an important one to answer. In our implementation we chose to only move the left and right line-leaders to improve the cost, just as seen in fig. 6.12.

6.2.4 Obstacle Search

The purpose of an obstacle search is to discover obstacles that cannot be encountered simply by moving the sweep line forward. This can happen even when the number of robots would theoretically suffice to reach a new obstacle. An obstacle search is executed when the sweep line grows longer and runs out of robots in the reserve. To execute an obstacle search the sweep line moves backwards into a state in which at least one robot is in the reserve. This robot can then be used to extend the reach of the line as seen in fig. 6.13. Therein the sweep line is separated into two line segments, s_l on the left and s_r on the right with one robot covering the ends of both of these segments. This extends the reach of the chain of robots. Recall that we have r_{all} robots and need r_{line} robots to cover the line at its narrowed position. At this point we can allocate $i + 1$ robots to the left line segment s_l with $i \in \{1, \ldots, r_{all} - 2\}$. The remaining robots $r_{all} - i$ cover segment s_r. Fig. 6.14 shows the simple trigonometry involved in finding the outmost position for every $i \in \{1, \ldots, r_{all} - 2\}$. Since the lengths of the triangle formed by each point p_i are known we can find p_i and move the robots to cover the s_l and s_r for every choice of i. If no obstacle is encountered for any i the line moves further backwards until yet another robot is freed and executes another search. For this we need the method $move\_line\_backwards(r)$ which moves the sweep line with r robots backwards and returns the new number of robots on the sweep line once it decreases. If the backward movement cannot free up an additional robot it returns the old number of robots and the search has failed

and it triggers a search failure.

The method described in this section is implemented for our experiments, but in principle one can use other implementations for an obstacle search as long as they improve the capabilities of the robot team to discover obstacles. Notice that the presented method does not guarantee that an obstacle will be discovered even when with a known map an obstacle could be reached. This is due to the fact that the points that can be reached is dependent on the position of the line when the obstacle search is executed. Fig. 6.15 illustrates this with an example. It is unclear how to amend the present method so that one could guarantee that an obstacle will be discovered, but it will surely involve starting a search with multiple line positions. But without metric information or reliable odometry this is bound to be challenging. In Algorithm 11 we present some details on this procedure.

If the search procedure is successful and a new obstacle is encountered the line splits into a left and right side, similar to the discover of an obstacle while moving a line forward. If the search procedure is not successful, then the robots go back to a previous split which they choose with the help of a surveillance graph. This is the backtracking that involves recontamination which we discuss in the next section.

6.2.5 Surveillance Graphs and Line Coordination

The methods introduced in the previous section can been seen as local behaviors of the robot team that enable the movement of robot lines through the environment. In principle, they can be implemented in a variety of ways depending on the robots and we presented methods that require rather limited capabilities. We shall now describe how to coordinate the execution of these methods and

```
r_old ← r_line
r_new ← move_line_backwards(r_old)
while !new_obstacle do
  if r_new < r_old then
    for i ← 1 to r_all − 2 do
      Compute p_i
      for robot ← 2 to r_all − 1 do
        if robot < i then
          s2 ← r_2(t) − p_i
          g ← (r_follow / length(s2)) · (robot − i)
        else if robot >= i then
          s1 ← r_1(t) − p_i
          g ← (r_follow / length(s1)) · (i − robot)
          move_robot(robot, g)
        else
          g ← p_i
        move_robot(robot, p_i)
    r_old ← r_new
    r_new ← move_line_backwards(r_old)
  else
    trigger_search_failed()
    return
return
```

Algorithm 11: *Obstacle_Search*()

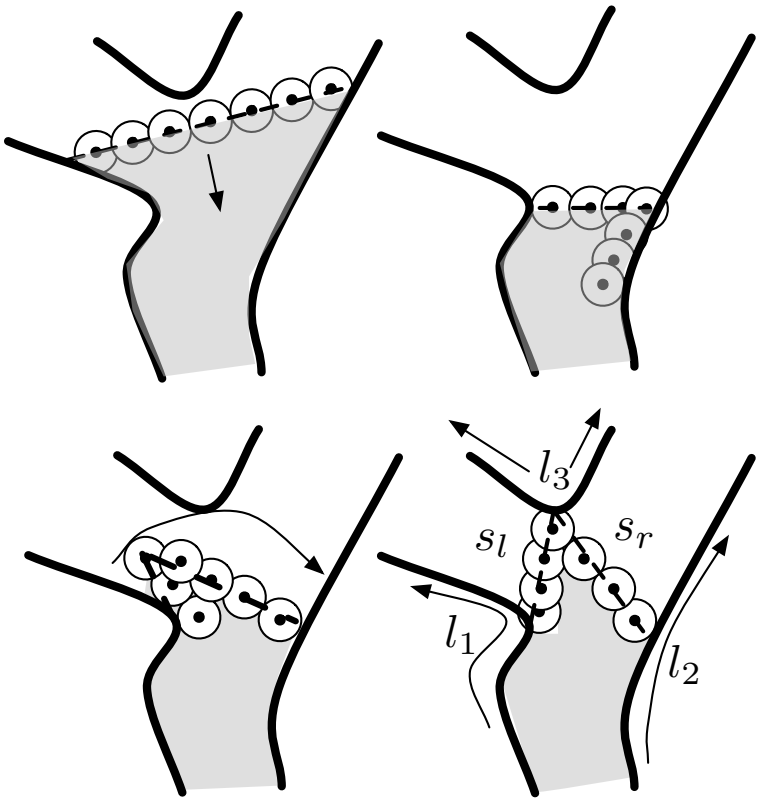


Figure 6.13: Top left: a line runs out of robots and cannot move further (the arrow indicates how it will move back). Top right: it then moves back. Bottom left: it then searches for a new obstacle. Bottom right: the line is split in two and each of them can individually move.

construct a surveillance graph of the environment that represents the discovered topology of the environment as well as the cost of clearing it. This surveillance graph is then used to coordinate the execution of the local behaviors and scale them to clear a large environment.

A surveillance graph that represents all previous line movements and their costs is easily constructed as follows. First, a vertex is created and given the cost of the starting line. Every successful encounter of a third obstacle leading to a split creates three new vertices. The first receives as cost the number of robots

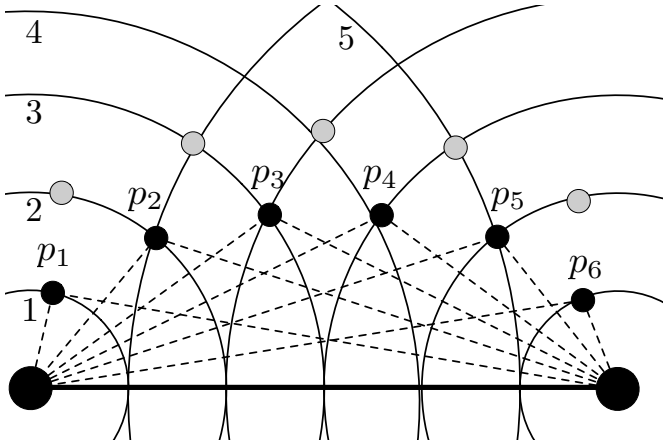


Figure 6.14: The radii of the circles are multiples of r_{follow}. The line from which the search originated is marked as a thick black line with thick circles representing the left and right line leader. The original line requires 7 robots and the small black dots show which points can be reached with a total of $r_{all} = 8$ robots by separating them onto the two new line segments into, i.e. for $i = 1$ there are $i + 1$ robots on the left segment and $8 - i$ robots on the right segment. The grey circles indicate the points that could be reached if we had $r_{all} = 9$.

that are needed to execute the split on the third obstacle into two sweep lines, and is connected to the graph trough an edge added to the previous vertex. The weight of that edge is the number of robots needed to cover the shortest line that was encountered since the last vertex was added. Finally, two vertices for the obstacle encounter are created for each new line and these receive as weight the cost of the respective new line. For each of these new vertices, an edge to the first vertex of the obstacle encounter is added, and it receives the same weight as the vertex. This is illustrated in fig. 6.16. All vertices that are just added are marked unexplored and have hence a sweep line associated that may lead to the addition of new edges to that vertex. Once a new vertex is added with an edge to that unexplored vertex it is considered explored. Alternatively, if a line is

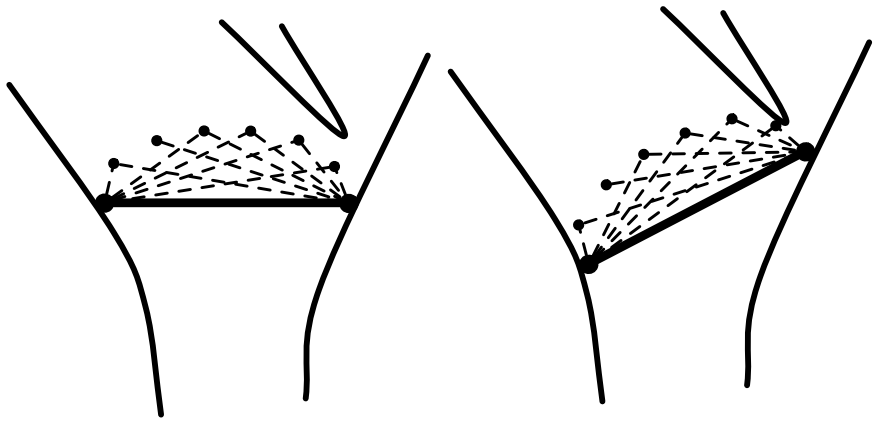


Figure 6.15: In this example a line moving forward runs out of robots and moves backward. It then initiates a search but cannot reach the third obstacle. Executing a search from a different line position can, however, reach it.

dissolving due to *trigger_dissolve_line*() the vertex does not receive a new edge and becomes a leaf that is also set to explored.

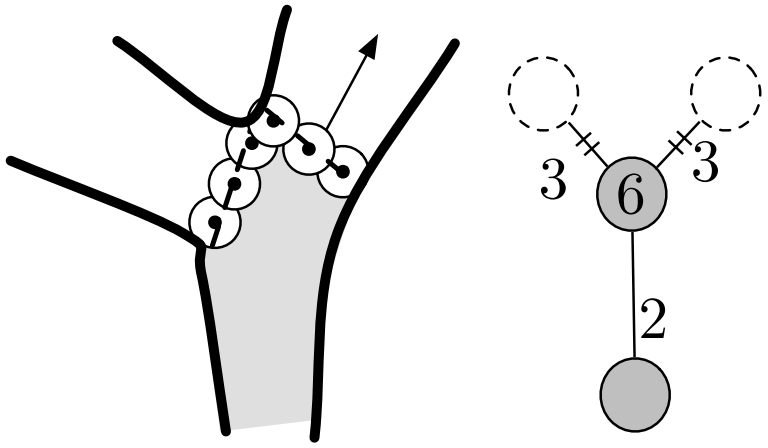


Figure 6.16: This figure shows the three vertices added to a graph when a split occurs. Three vertices and three edges with their weights are added and connect to the existing graph.

Furthermore, a number of consecutive increases in the number of robots while

Initialize surveillance graph $G = (V, E, w)$ with one vertex v_{last}

while *true* **do**

  $event \leftarrow catch\_event()$

  if $event = new\_obstacle$ **then**

    add vertices v_1, v_2, v_3 to V and to U

    $w(v_1) \leftarrow cost(s_1) + cost(s_2)$, $w(v_2) \leftarrow cost(s_1)$, $w(v_3) \leftarrow cost(s_2)$

    add edges $(v_{last}, v_1), (v_1, v_2), (v_1, v_3)$ to E

    $w(v_{last}, v_1) \leftarrow last\_min$, $w((v_1, v_2)) \leftarrow cost(s_1)$, $w((v_1, v_3)) \leftarrow cost(s_2)$

    $U.remove(v_1)$, $U.remove(v_{last})$, $line\_grew \leftarrow false$

    if $length(s_1) > length(s_2)$ **then**

      $v_{last} \leftarrow v_2$, $last\_min \leftarrow cost(s_1))$

    else

      $v_{last} \leftarrow v_3$, $last\_min \leftarrow cost(s_2))$

  else if $event = line\_grows$ **then**

    $line\_grew \leftarrow true$

  else if $event = line\_shrinks$ **then**

    if $line\_grew$ **then**

      $line\_grew \leftarrow false$, $U.remove(v_{last})$

      add vertex called v to G, add edge (v_{last}, v) to E

      $w((v_{last}, v)) \leftarrow last\_min$, $w(v) \leftarrow cost(l_{current})$, $v_{last} \leftarrow v$

      $last\_min \leftarrow cost(l_{current})$

    else

      $last\_min \leftarrow last\_min - 1$

  else if $event = dissolve\_line$ **then**

    $U.remove(v_{last})$

return

Algorithm 12: *Graph_Construction*()

moving the line forward followed by one decrease creates a vertex. This represents the cost of passing through a region. Naturally, edges are created for every two vertices that are encountered consecutively when moving a sweep line forward. Their weight is always given by the number of robots needed to cover the shortest sweep line encountered during the movement between vertices. We call this sweep line *blocking line*, since it represents the cost of a blocking action of the surveillance graph as it has the lowest cost of preventing contamination between the vertices with a robot line. Built this way, the surveillance graph is a record of the line movements. A traversal of a vertex in the graph can be associated to moving a line from a blocking position to either another blocking line or a split. It hence captures the motion primitives that the robot team can jointly execute. Fig. 6.17 illustrates this and shows how vertices and edges are created as the lines move and split. Algorithm 12 presents some details on the graph construction in pseudocode. This algorithm has access to the variables s_1, s_2 which are the new sweep lines after a split, the variable U which is a set of unexplored vertices and generates a graph G that is merged when robots of different lines meet.

Given a sufficiently large number of robots for the environment being cleared, one can incrementally discover the whole graph for an environment by just choosing to continue on an arbitrary side, left or right, after a split, and coming back once the side is entire cleared. If a successful search is triggered the robot team can also continue with a split. But the sequence in which vertices are encountered can be expected to lead to a higher cost in terms of robots than the best possible sequence based on a Graph-Clear strategy. The discovery of more vertices may require all previously cleared vertices to be recontaminated to reach a state of the graph from which an improved strategy can be executed. In the worst case the discovery of the last vertex can require the recontamination of all previously cleared vertices. This is an unavoidable problem, also occurring

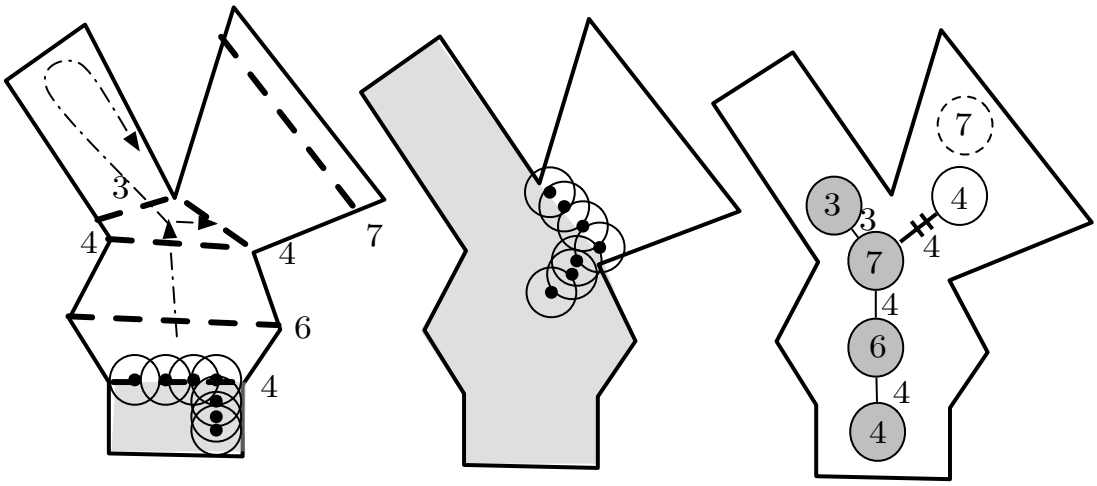


Figure 6.17: Example of the construction of a graph as a result of the exploration with lines. The cleared and known vertices are marked in grey. The lines leading to their creation are marked as thick dashed lines with the associated number of robots needed. The robots explore the environment, following the arrows and stop before discovering the vertex with weight 7. Hence the neighboring vertex 4 is not considered explored.

in [SRL04] which also computes and executes partial solutions that can require to start from scratch once the entire environment is explored.

Let us now describe the final part of the algorithm. At any point only one sweep line is moving. It has all available robots in its reserve. Once a split is encountered the robots choose with which direction, s_l or s_r (left or right), to continue while the other side will remain stationary until the robots from the moving sweep line return and activate it. Once the length of the moving line reduces to zero, i.e. the two endpoints meet, it is dissolved and all its robots follow either the left or right wall to go back to the next stationary line. The same happens when the moving line encounters a stationary line which joins the robots of both lines and sends them to the next stationary line. This happens

only for multiply connected environments, i.e. those with cycles. For now we shall ignore this event and merely note that it can be dealt with by adding a cycle edge into the graph and then simply dissolve the two lines consecutively. This extension is not part of our current implementation. Every stationary line is always associated to an unexplored vertex and hence one can find the next stationary line by searching through the graph. As vertices are added during the exploration they are considered unexplored until the line passing through them has led to the addition of another vertex or triggered a dissolution of the line. The choice of the next stationary line to become a moving line after a split is in principle arbitrary since the cost of clearing the environment behind this line is not known. As a heuristic one can choose to extend the line with more robots, leaving less robots as stationary. Similarly, after the dissolution of the line the choice is also arbitrary. As a heuristic we minimize the graph distance for our implementation, but other criteria can be applied, such as picking the longest stationary line.

If the entire algorithm continues without failed obstacle searches occurring it is in principle a depth first traversal and discovery of a tree, if the environment is simply connected, or a graph, if it is multiply-connected. If an obstacle search fails, the robots move the sweep line backwards until a stationary line is encountered. Then the moving line then becomes stationary and the remaining robots choose the next closest stationary line on the graph and attempt to extend it. It is, however, possible that all stationary lines that are available will lead to failed searches. In this case the given number of robots does not suffice to continue to explore the environment from the current state of the graph and it has to be recontaminated in order to free up robots that are currently at stationary lines blocking contamination. There is a variety of possibilities how to achieve this. The brute force approach would be to recontaminate all known vertices and

move all robots to the chosen unexplored vertex. Obviously, a totally contaminated state of the graph does not require any robots to block contamination and hence frees all robots. To avoid as much recontamination as possible one could resort to modifying the algorithms from Chapter 3 to identify states of the graph that allow more robots to be freed to search for obstacles. Unfortunately, in the worst case it may well be that the unexplored vertex has a cost that requires all robots. The environment could still be cleared with the given number of robots, but only if starting at the high cost vertex. With plenty of excess robots and if the total time needed for clearing is an issue, then cut sets should be used to identify which vertices to recontaminate and free robots one by one. Algorithm 13 shows how to combine all the previous methods. The following functions are required for this:

1. $move\_line\_forward(l)$: calls the proper functions on the robots that move a sweep line l forward.

2. $stationary\_line\_at(v)$ returns the sweep line that is associated to a vertex v.

6.2.6 Implementation and Testing

Most of the procedures presented above can be implemented in a variety of ways, depending on the robot platform. In particular, if robots are non-holonomic or part of a heterogenous team, the wall and line following behaviors have to be designed with care. To demonstrate the feasibility of the proposed approach we implemented a specific wall following routine that utilizes a laser sensor to estimate the obstacle tangent. Robots are simulated with the Player/Stage software [BCG05] and have an omnidrive and a laser range finder with a 360° field

Set up the first sweep line $l_{current}$ (bootstrapping)

Create an empty set U of unexplored vertices for graph construction

Run $Graph\_Construction()$ in a separate thread

$move\_line\_forward(l_{current})$

while $!U.empty()$ **do**

  $event = catch\_event()$

  if $event = new\_obstacle$ **then**

    set up split lines s_1 and s_2

    if $length(s_1) > length(s_2)$ **then**

      $move\_line\_forward(s_1)$

    else

      $move\_line\_forward(s_2)$

  else if $event = dissolve\_line$ **then**

    move free robots to any unexplored vertex v

    $l_{current} \leftarrow stationary\_line\_at(v)$

  else if $event = search$ **then**

    $Obstacle\_Search()$

  else if $event = search\_failed$ **then**

    mark search failed for unexplored vertex v_{fail} and find new vertex $v \in U$.

    if $v = null$ **then**

      free robots by recontaminating

      unmark failed search marks for all $v \in U$

      move free robots to any unexplored vertex v_{new}

      $l_{current} \leftarrow stationary\_line\_at(v_{new})$

    else

      move robots to v, $l_{current} \leftarrow stationary\_line\_at(v)$

Algorithm 13: $clear\_environment(l_{current})$

of view and limited range of $0.7m$. Every robot is controlled by a separate client program and communicates with other robots through the network interface. As a test environment we created the simply connected environment seen in fig. 6.18. The environment is based on a 457x458 pixel bitmap with a resolution of 0.032 meter per pixel leading to width and height of approximately 14.5m.

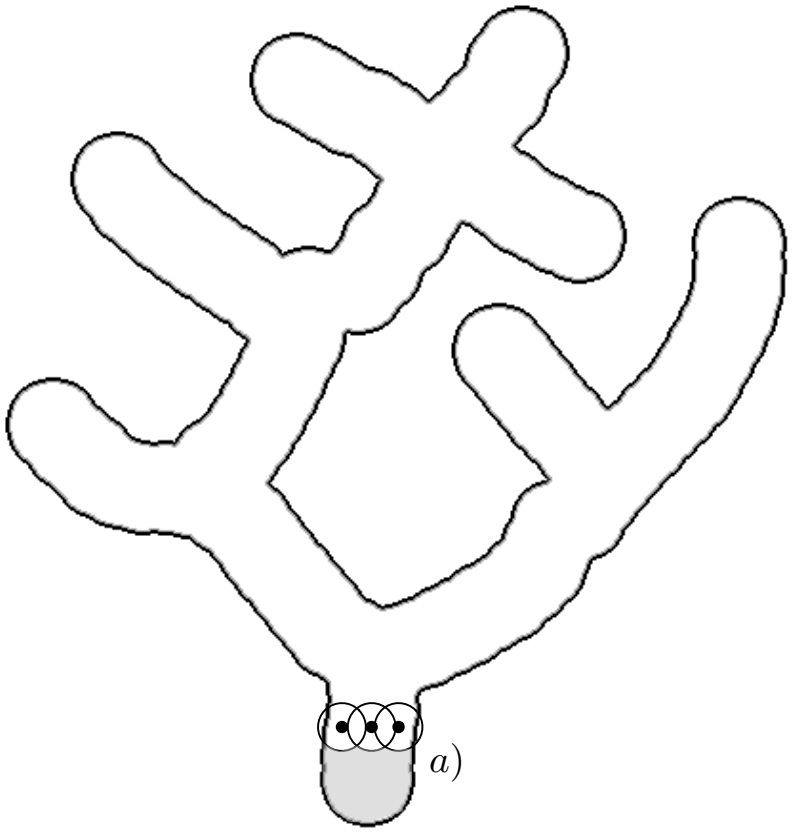


Figure 6.18: A simply-connected environment with the starting point for the clearing and homebase at a).

Applying the Line-Clear algorithm with graph extraction to this environment yields a cost of 7 robots given a desired distance between robots on a line of approximately 1 meter from each other. With a laser sensing range of 0.7 meter

this distance leads to an overlap of around 0.4 meter of the sensors between adjacent robots. This accommodates for errors in synchronization and control of the robots. The resulting surveillance graph is shown in fig. 6.19.

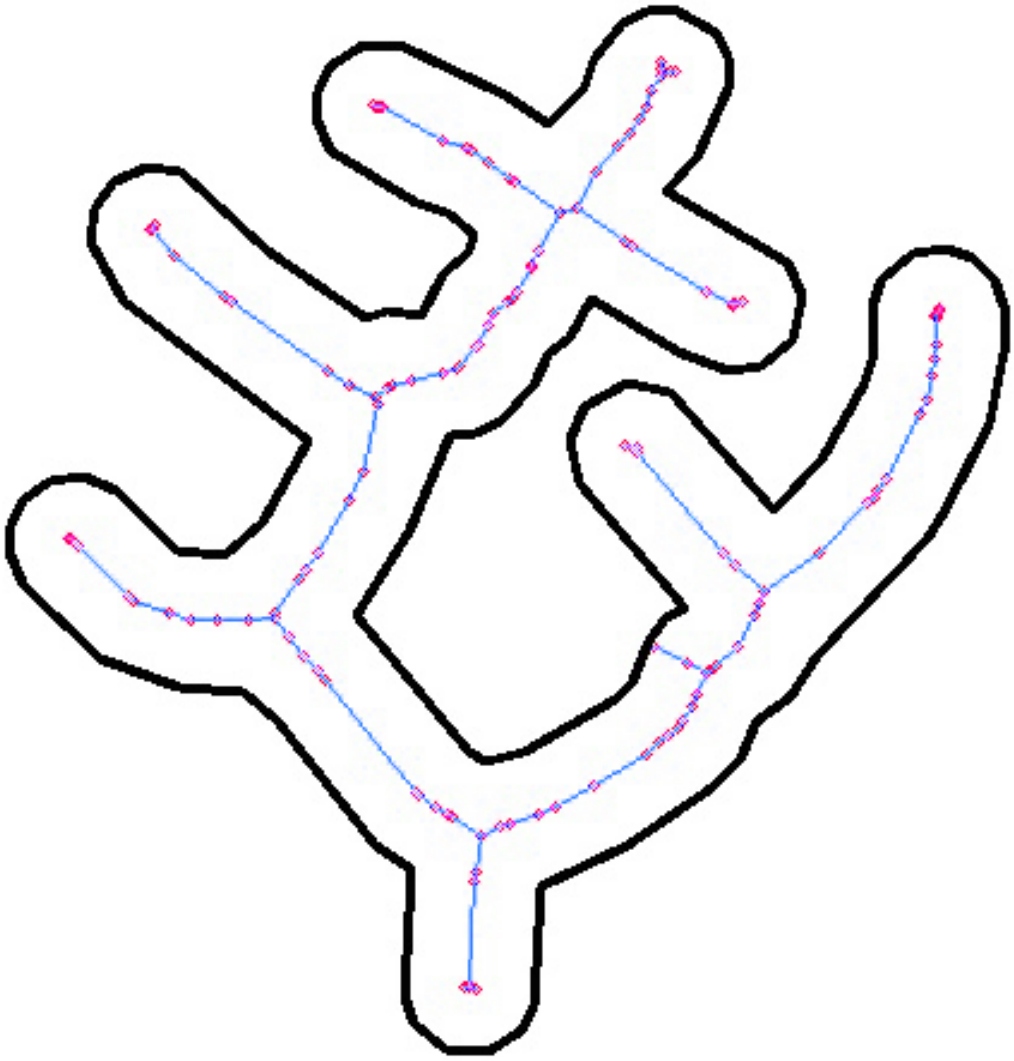


Figure 6.19: The surveillance graph corresponding to a sweep schedule created with the methods from Section 5.2. The surveillance graph has 141 vertices resulting in 7 robots for clearing it.

When implementing the distributed algorithm there are few technical details

that have to be considered. These primarily pertain to synchronization and control issues. One has to ensure that the control of the robot is stable enough so that it follows the boundary reliably. Since the sensed obstacle tangent is only an estimate based on two laser reading it can lead to some jitter in the motion of the robot. Also, in order to sense to obstacle gradient reliably the robot needs to be closer to the obstacle to make sure that two laser readings are available that are not immediately adjacent. Hence, we required the robots to be within 0.4 meters of the obstacle boundary and used two readings that were a few degrees apart to estimate the obstacle tangent. To ensure that robots can pass messages to each other, remain under stable control and make high level decisions at appropriate times we separated these functions into multiple threads running concurrently. Synchronization issues also have to be handled with care and we introduced thresholds for certain events such as a robot joining or dropping out of a line. These events will only be triggered if the line is robustly expanding or contracting to avoid very quick successive events which can lead to problems if robots do not receive them in proper order. The obstacle search and split were implemented as described before with the exception that for an obstacle search to occur at least two robots need to be in the reserve, one for extending the reach of the search and one for splitting the line if a search is successful. This is done since a search usually has only one robot hitting an obstacle and hence a split needs an additional robot to move onto the position of the robot that hit the obstacle. For a split in which two robots see the obstacle we can split the line between these two without requiring a reserve robot.

The starting position for the robot team was chosen to be the bottom of the environment as seen in fig. 6.18 as the homebase. The overlap between sensor of adjacent robots was set to 0.4 meter, just as for the computation of the sweep schedule, leading to a desired distance between robots of 1 meter. In practice

this worked well to ensure that delays in the robot control do not lead to the line falling apart. We ran the algorithm multiple times with 8 and 9 robots on the environment from 6.18 and will describe the typical results in the following. Notice that the investigations here are qualitative and not quantitative. With 8 robots the algorithms fails at the step shown in fig. 6.20. If the robots were to start at the top right of the environment, however, this problem would not occur and they are then able to clear the environment. Hence, an implementation allowing complete recontamination and resetting the starting location can improve performance. Yet it is unclear, how to effectively find new starting positions and this may well involve many failed attempts. This illustrates the sensitive to the starting location which is analogue to the choice of the best starting vertex in a surveillance graph.

With 9 robots the environment is successfully cleared from the homebase. The first split that occurs is seen in fig. 6.21. The second split that was problematic for 8 robots succeeds after an obstacle search as seen in fig. 6.22. Subsequently they clear the entire right side of the environment and join the bottom 3 robots for the left side. This side is considerably harder to clear and there we will see an example of backtracking that eventually leads to a successful clearing. Fig. 6.23 shows how the robots have to backtrack twice to collect all 9 robots when they attempt to cross the four way intersection. Each backtracking step involves recontamination of the graph that is built. Once 9 robots are available the obstacle search succeed for the top part as seen in fig. 6.24. It should be noted that due to the large number of robots the graph recorded for the last steps does not reflect the best way of clearing it. The robots simply move around the left side even though a split would be better. Since they do not have information about this we can force a split by removing a robot. With 8 robots this part can be cleared and the robots have to execute an obstacle search as seen in fig. 6.25.

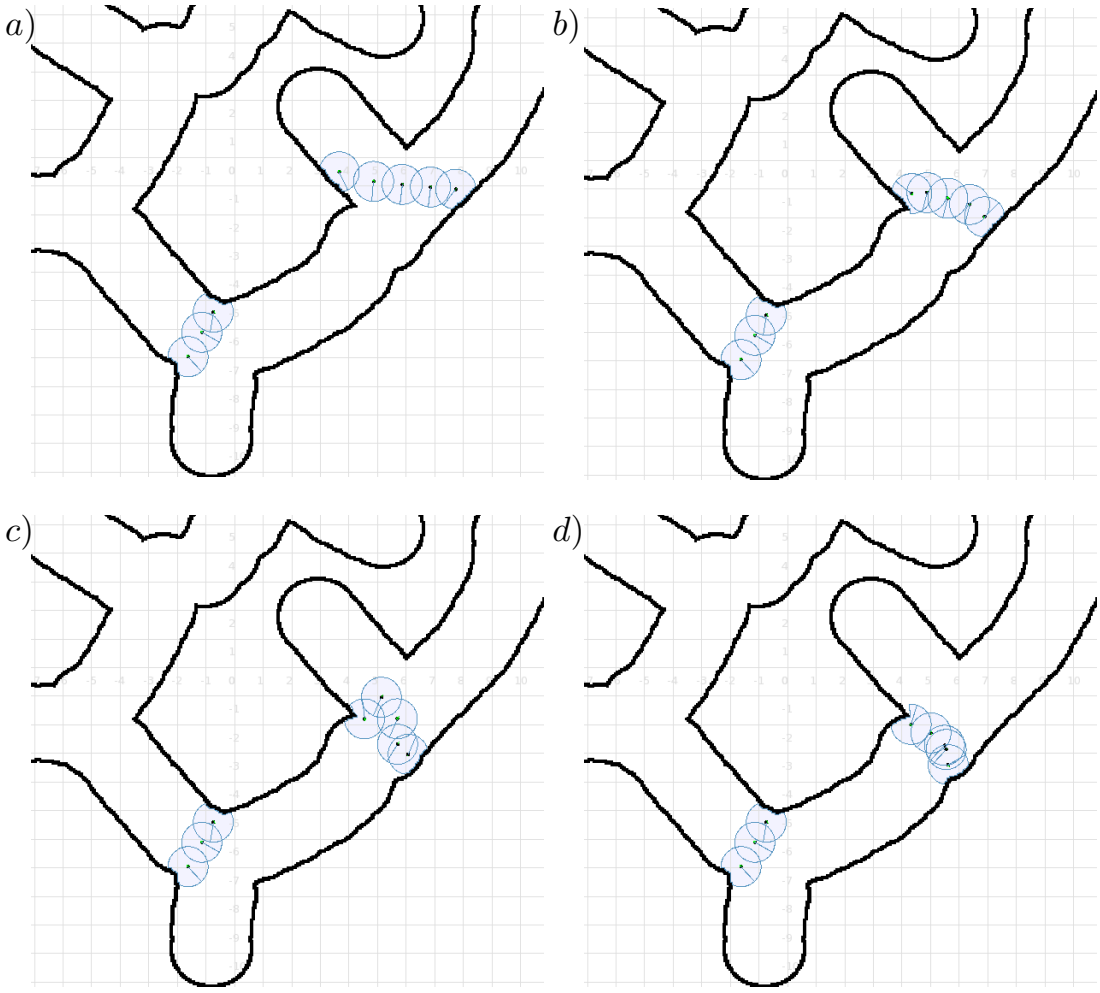


Figure 6.20: With 8 robots the distributed algorithm fails to clear this environment starting at the bottom. This is due to the failed obstacle discovery with 5 robots at the top right while 3 robots block the edge to the left. In part a) the team runs out of robots and move back to part b). Once the line shrinks it attempts an obstacle search in part c) but the reach is too limited and it fails. The team then backtracks and tries the left side while blocking the right with 3 robots and fails in an identical manner.

Fig. 6.26 shows the final graph that the 9 robots created.

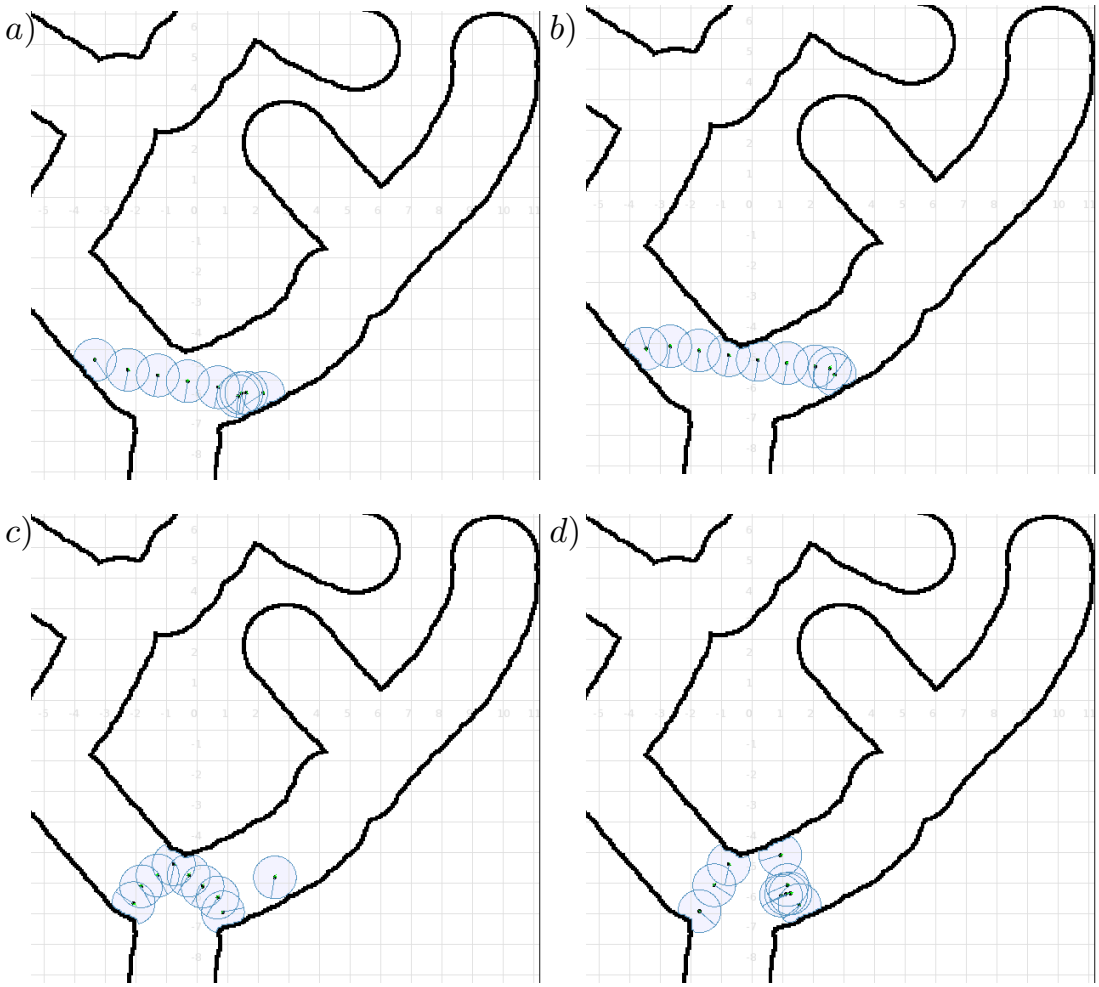


Figure 6.21: The first split of the line moving with 9 robots that subsequently clears the environment.

6.2.7 Discussion and Conclusion

The results in this section are by no means a quantitative analysis of the algorithm. For this we would have to consider more manifold scenarios and configurations and compare across these. Yet, they offer a qualitative insight that encourages further investigation. First we see clearly that the number of robots traveling with a line influences the way a graph is built. On the one hand this

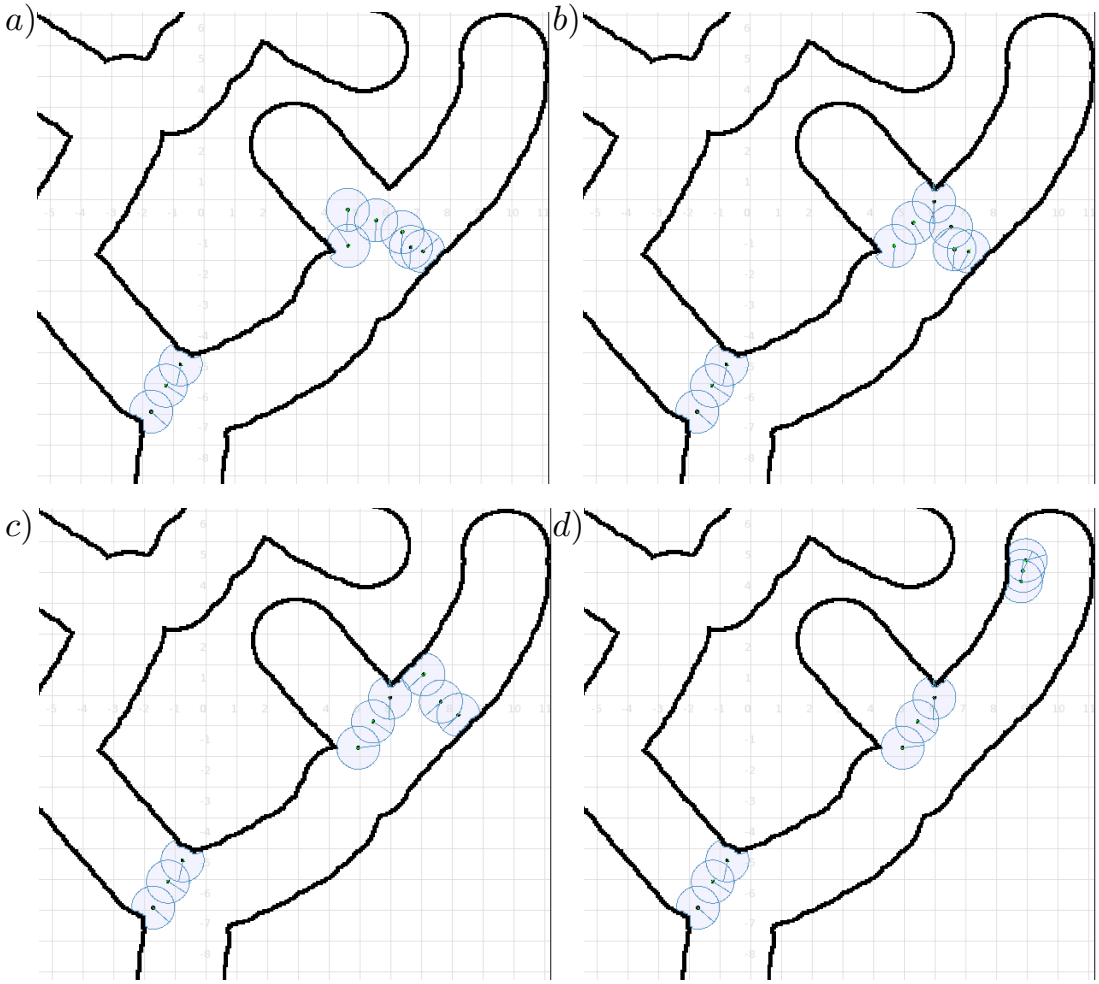


Figure 6.22: A successful search after the second search step shown in part b). After the search the line splits and the right side clears the leaf in part c) and returns in part d) to join the left side.

seems obvious, but it is significant. A robot team with less robots cannot know whether they can clear an environment unless they have tried it as seen from the contrast between fig. 6.24 and fig. 6.25. Most of these problems would be remedied if we were able to build a metric map as the robots explore the environment. But in a way this case is uninteresting and if one can build a map Line-Clear can be applied in a straight-forward manner and coupled with an exploration

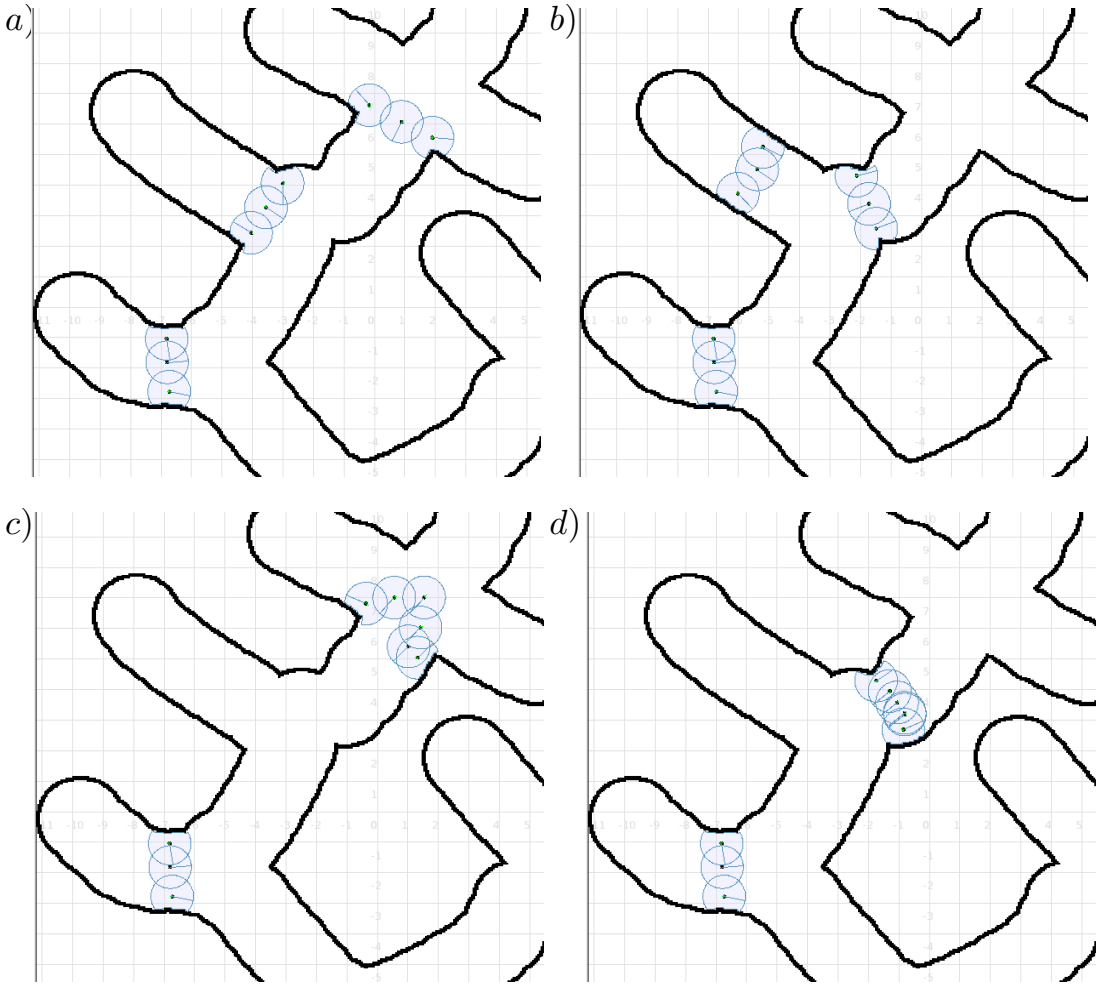


Figure 6.23: For the left side the robots have to backtrack extensively. In part a) they fail with three robots and backtrack to collect 3 more. They fail again with 6 robots and have to backtrack again to collect the bottom three.

algorithm. The approach from this section is fundamentally different and points toward the question of what minimal capabilities a robot team needs to be able to solve a pursuit-evasion task. So far we have reduced it to wall following, sensing a neighbor and targets and communicating locally. But ideally one would like to be able to prove that such a system is equally capable than one with a known metric map and localization capabilities which would enable to execute a

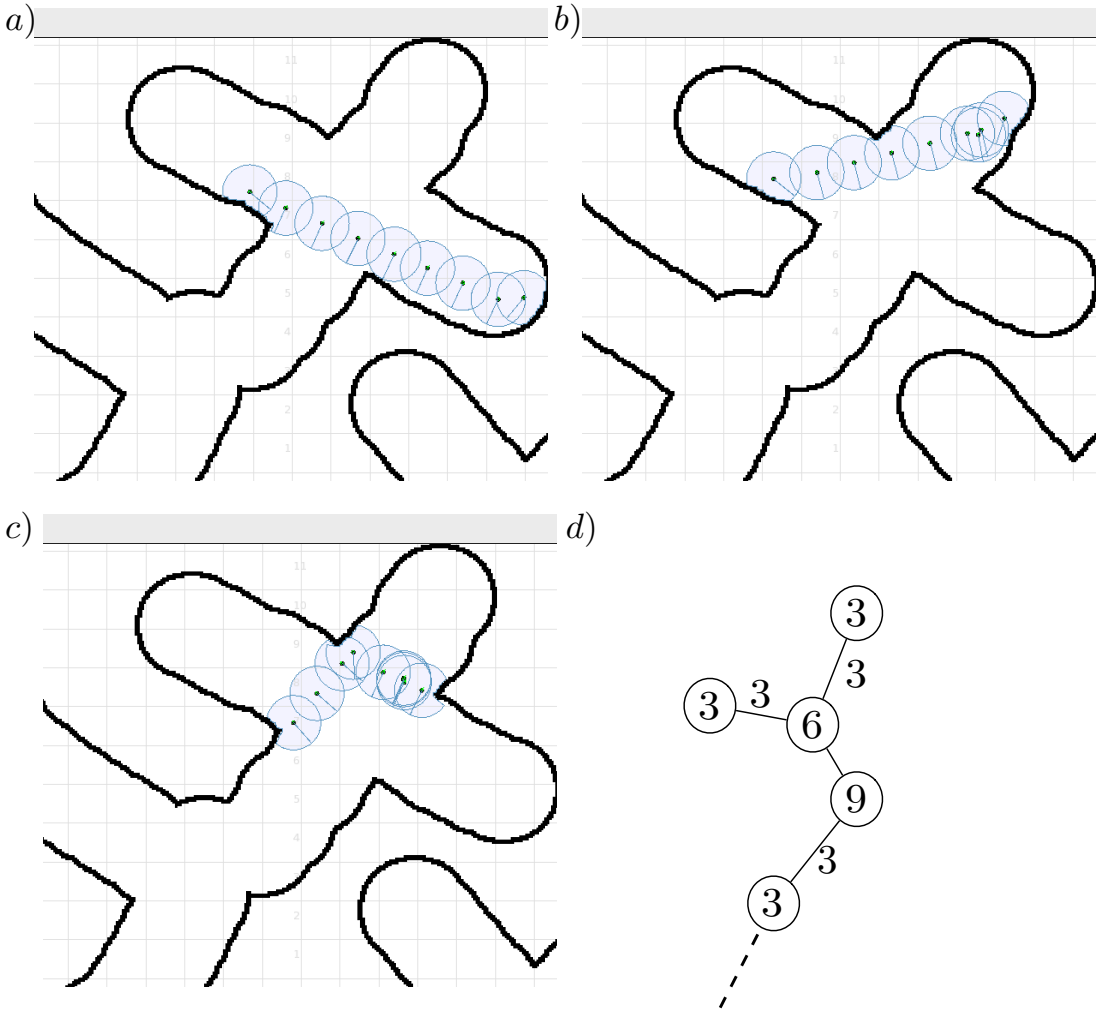


Figure 6.24: The last steps of the clearing. In part a) the robots extend far to follow the boundary and finish with steps b) and c). The graph resulting from this is shown in d).

Line-Clear sweep schedule from a map. For this further investigations are needed. One would need to show that the obstacle search can guarantee to find obstacles if it is possible that one can be in the given map. One would also have to find a routine that resets the exploration and starts anew once the team notices that further clearing is not possible.

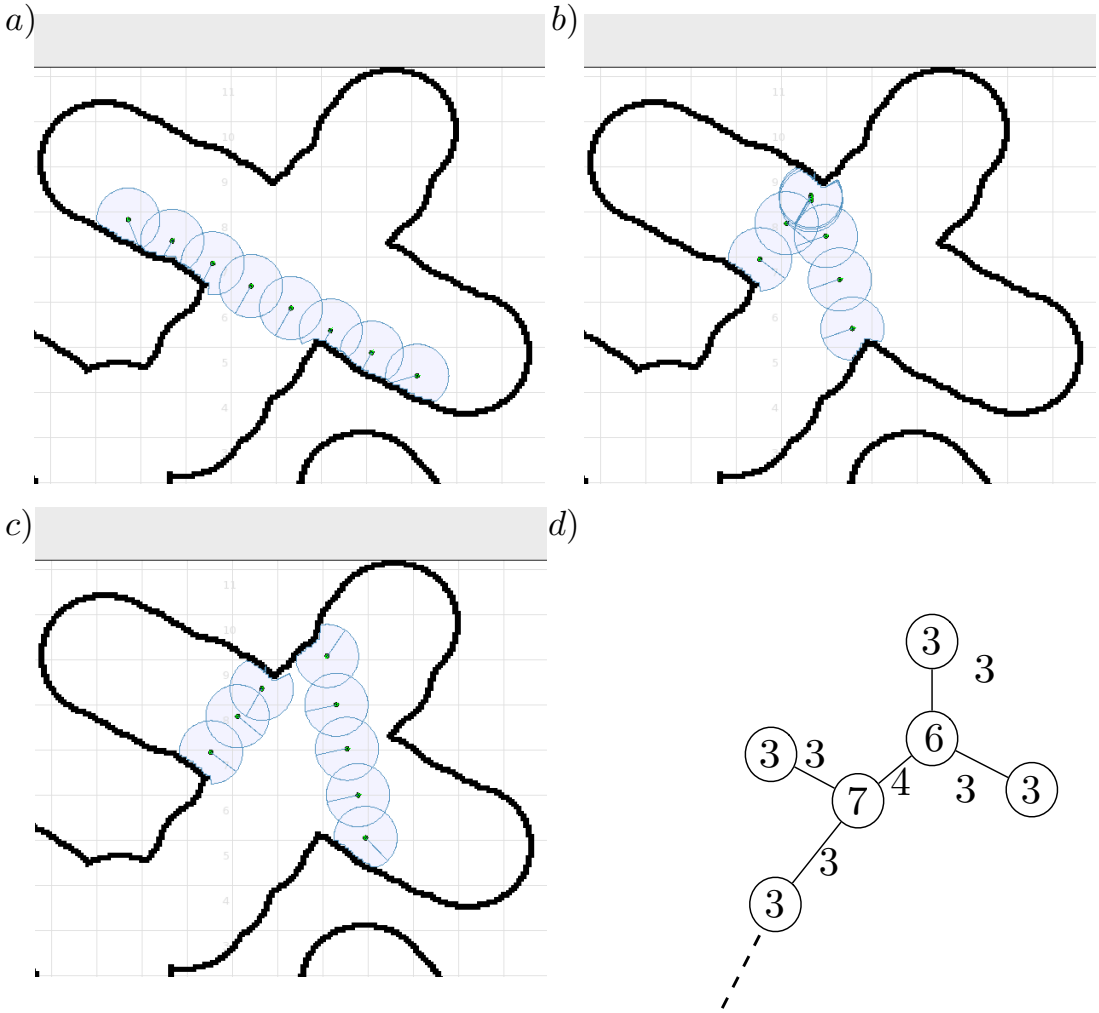


Figure 6.25: Reducing the number of robots to 8 forces an obstacle search which leads to a different graph than fig. 6.24 as shown in d).

For all practical purposes and in the spirit of large robot teams, in which individual robots are merely a commodity, we expect that the addition of a few robots is preferred to extensive obstacle searches and recontamination through backtracking. For such applications the presented work can already be useful.

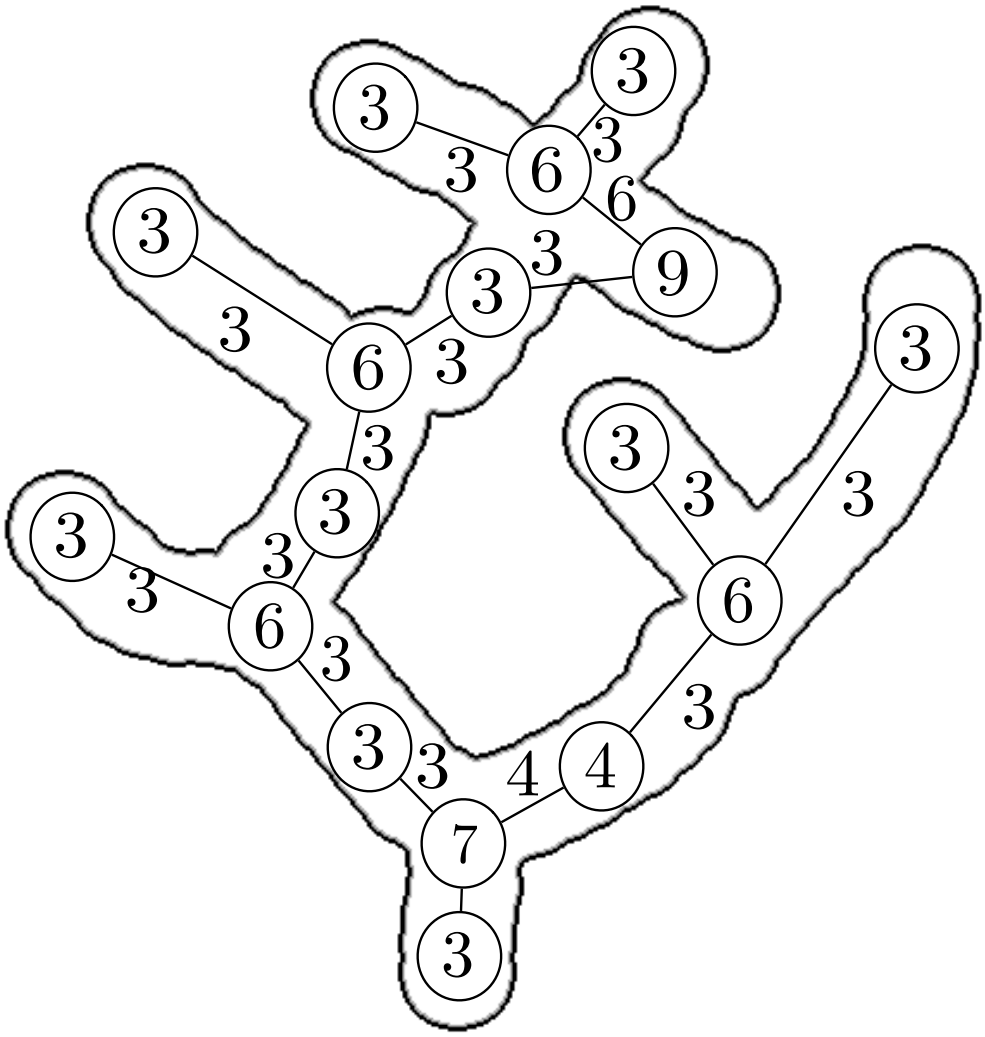


Figure 6.26: The final graph created by 9 robots embedded in the environment.

6.3 Discussion and Conclusion

This chapter briefly presented two implementations that translate the Graph-Clear and Line-Clear theory and algorithms to actual robot motion. The theory developed for Graph-Clear and Line-Clear attempts to work with high level abstractions that can then be implemented for a variety of settings. The work in

this chapter serves to exemplify its application for particular settings. Additionally, both experiments opened many further questions. Questions arising from the experiments from Section 6.1 have been discussed extensively in Chapter 4. The new questions from the distributed algorithm from Section 6.2 have not yet been studied thoroughly and are open to investigation.

CHAPTER 7

Discussion and Conclusion

In this dissertation we presented two models for pursuit-evasion, one in graphs called Graph-Clear and one in two dimensional environments called Line-Clear. These models abstract the sensing capabilities of the robot team to different degrees and require an implementation of either sweeping and blocking or line covering behaviors on the robot team. The major advantage is that they do not commit to a particular sensor or robot platform but are applicable as long as a given set of hardware is able to implement the behaviors. We have demonstrated that it is possible to automatically generate surveillance graphs from robot maps and relate these to particular sweep and block or line covering behaviors. Finally, we demonstrated how to put everything together to design a multi-robot system that solves the pursuit-evasion task in real and simulated environments.

The results of this dissertation are by no means a conclusive answer to the challenges of multi-robot pursuit-evasion. We rather tackled a few of the most glaring problems and there are very many that remain. Foremost, we dealt with the combinatorial problem of allocating robots to actions on a graph and the geometric problem of moving lines at low cost through a two dimensional environment, reusing our algorithm from the combinatorial problems. We also started to accommodate a few fundamental constraints of multi-robot systems, such as limited range and possibly faulty sensors, errors in control and limited communication range. We have, however, ignored motion constraints and hence non-

holonomic robot platforms. For these, implementations of line-covering behaviors may well lead to larger costs when considering such platforms and situations in which the robot team cannot follow the movement of a line due to its motion constraints should then be addressed. Yet, the core of the theory will still apply, even when particular blocking, sweeping, or line covering implementations have to be designed more elaborately. Another area that we did not cover is the introduction of knowledge about targets. Once knowledge about target speed or behavior is available one may attempt to maximize the likelihood of an early detection or one can relax the sensor coverage. For example if a target moves slowly, one robot may be able to patrol an entrance by moving back and forth, knowing that slow target cannot cross in the time it leaves some areas uncovered. Finally, one important aspect to address in future work is that of heterogenous robot teams, particularly those in which there are stationary and mobile platforms.

Altogether, despite the still large number of unanswered questions looming at the horizon, we are confident that the presented theory and algorithms will aid the development of multi-robot systems for pursuit-evasion. Once the trends towards reduced cost for robotic hardware continue sufficiently far and the costs of a human operator for every robot becomes prohibitive we can expect to see a dramatic rise in the use of multi-robot systems. Some of these might then be running Graph-Clear.

References

[ARS02] M. Adler, H. Racke, N. Sivadasan, C. Sohler, and B. Vocking. "Randomized pursuit-evasion in graphs." In *Proceedings of the International Colloquium on Automata, Languages and Programming*, volume 2380, pp. 901–912, 2002.

[BBH07] S. D. Bopardikar, F. Bullo, and J. P. Hespanha. "Cooperative pursuit with sensing limitations." In *American Control Conference*, pp. 5394–5399, 2007.

[BCG05] G. Biggs, T. Collett, B. Gerkey, A. Howard, N. Koenig, J. Polo, R. Rusu, and R. Vaughan. "Player/Stage project." http://playerstage.sourceforge.net, 2005.

[BCM09] F. Bullo, J. Cortés, and S. Martínez. *Distributed control of robotic networks*. Applied Mathematics Series. Princeton University Press, 2009. To appear. Electronically available at http://coordinationbook.info.

[BF87] Y. Bar-Shalom and T. E. Fortmann. *Tracking and data association*. Academic Press Professional, Inc., San Diego, CA, USA, 1987.

[BFF02] L. Barrière, P. Flocchini, P. Fraigniaud, and N. Santoro. "Capture of an intruder by mobile agents." In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 200–209, New York, NY, USA, 2002. ACM Press.

[BG08] E. Bitton and K. Goldberg. "Hydra: A framework and algorithms for mixed-initiative UAV-assisted search and rescue." In *IEEE International Conference on Automation Science and Engineering*, pp. 61–66, 2008.

[BKO00] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational geometry: algorithms and applications*. Springer, 2000.

[BLK01] Y. Bar-Shalom, X.-R. Li, and T. Kirubarajan. *Estimation with applications to tracking and navigation*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[BMF00] W. Burgard, M. Moors, D. Fox, R. Simmons, and S. Thrun. "Collaborative multi-robot exploration." In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 1, pp. 476–481, 2000.

[BS91] D. Bienstock and P. Seymour. "Monotonicity in graph searching." *Journal of Algorithms*, **12**(2):239–245, 1991.

[BS03a] M.A. Batalin and G. S. Sukhatme. "Efficient exploration without localization." In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 2, pp. 2714–2719, 2003.

[BS03b] M.A. Batalin and G. S. Sukhatme. "Sensor network-based multi-robot task allocation." In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1939–1944, 2003.

[BS05] M.A. Batalin and G. S. Sukhatme. "The analysis of an efficient algorithm for robot coverage and exploration based on sensor network deployment." In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 3478–3485, 2005.

[BSH03] M.A. Batalin, G. Shukhatme, and M. Hattig. "Mobile robot navigation using a sensor network." In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 1, pp. 636–641, 2003.

[BTK09] R. Borie, C. Tovey, and S. Koenig. "Algorithms and complexity results for pursuit-evasion problems." In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 59–66, 2009.

[CB94] H. Choset and J. Burdick. "Sensor based planning and nonsmooth analysis." In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 3034–3041, May 1994.

[CB95] H. Choset and J. Burdick. "Sensor based planning, Part I: the generalized voronoi graph." In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 2, pp. 1649–1655, 1995.

[CB07] T. H. Chung and J. W. Burdick. "A decision-making framework for control strategies in probabilistic search." In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 4386–4393, 2007.

[CFK97] Y. U. Cao, A. S. Fukunaga, and A. B. Kahng. "Cooperative mobile robotics: antecedents and directions." *Autonomous Robots*, **4**(1):7–23, March 1997.

[Cho01] H. Choset. "Coverage for robotics – A survey of recent results." *Annals of Mathematics and Artificial Intelligence*, **31**(1-4):113–126, 2001.

[CLR01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. McGraw-Hill Book Company, Boston, MA, 2nd edition, 2001.

[CMB04] J. Cortés, S. Martínez, and F. Bullo. "Robust rendezvous for mobile autonomous agents via proximity graphs in arbitrary dimensions." *IEEE Transactions on Automatic Control*, **51**(8):1289–1298, 2004.

[CMK04] J. Cortés, S. Martínez, T. Karatasand, and F. Bullo. "Coverage control for mobile sensing networks." *IEEE Transactions on Robotics and Automation*, **20**:243–255, 2004.

[CSY95] D. Crass, I. Suzuki, and M. Yamashita. "Searching for a mobile intruder in a corridor - The open edge variant of the polygon search problem." *International Journal of Computational Geometry and Applications*, **5**(4):397–412, December 1995.

[Da08] T. K. F. Da. "2D alpha shapes." In CGAL Editorial Board, editor, *CGAL User and Reference Manual*. 3.4 edition, 2008.

[DBC02] A. Drenner, I. Burt, B. Chapeau, T. Dahlin, B. Kratochvil, C. McMillen, B. Nelson, N. Papanikolopoulos, P.E. Rybski, K. Stubbs, et al. "Design of the UMN multi-robot system." In *Multi-robot systems: from swarms to intelligent automata: Proceedings from the 2002 NRL workshop on multi-robot systems*, p. 141. Kluwer Academic Publishers, 2002.

[DPS02] J. Diaz, J. Petit, and M. Serna. "A survey of graph layout problems." *ACM Computing Surveys (CSUR)*, **34**:313–356, September 2002.

[EG98] A. Elnagar and K. Gupta. "Motion prediction of moving objects based on autoregressive model." *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, **28**(6):803–810, November 1998.

[EST94] J.A. Ellis, I.H. Sudborough, and J.S. Turner. "The vertex separation and search number of a graph." *Information and Computation*, **113**(1):50–79, 1994.

[FBD99] D. Fox, W. Burgard, F. Dellaert, and S. Thrun. "Monte carlo localization: efficient position estimation for mobile robots." In *Proceedings of the National Conference on Artificial Intelligence*, pp. 343–349, July 1999.

[FGY00] M. Franklin, Z. Galil, and M. Yung. "Eavesdropping games: a graph-theoretic approach to privacy in distributed systems." *Journal of the ACM*, **47**(2):225–243, 2000.

[FKM03] F. V. Fomin, D. Kratsch, and H. Müller. "On the domination search number." *Discrete Applied Mathematics*, **127**(3):565–580, 2003.

[Fre07] E. W. Frew. "Cooperative standoff tracking of uncertain moving targets using active robot networks." In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 3277–3282, April 2007.

[FT08] F. V. Fomin and D. M. Thilikos. "An annotated bibliography on guaranteed graph searching." *Theoretical Computer Science*, **399**(3):236–245, 2008.

[GCB06] A. Ganguli, J. Cortes, and F. Bullo. "Distributed deployment of asynchronous guards in art galleries." In *American Control Conference*, pp. 1416–1421, Minneapolis, MN, June 2006.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and intractability : A guide to the theory of NP-completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[GLL99] L. J. Guibas, J.-C. Latombe, S. M. LaValle, D. Lin, and R. Motwani. "A visibility-based pursuit-evasion problem." *International Journal of Computational Geometry and Applications*, **9**:471–494, 1999.

[GSB] G. Grisetti, C. Stachniss, and W. Burgard. "GMapping – OpenSLAM.org." http://www.openslam.org/gmapping.html.

[GTG05] B. P. Gerkey, S. Thrun, and G. Gordon. "Parallel stochastic hill-climbing with small teams." *Multi-Robot Systems: From Swarms to Intelligent Automata*, **3**:65–77, 2005.

[GTG06] B. P. Gerkey, S. Thrun, and G. Gordon. "Visibility-based pursuit-evasion with limited field of view." *The International Journal of Robotics Research*, **25**(4):299–315, 2006.

[GTL04] L. Guilamo, B. Tovar, and S. M. LaValle. "Pursuit-evasion in an unknown environment using gap navigation trees." In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 4, pp. 3456–3462, April 2004.

[Guy91] R. K. Guy. "Unsolved problems in combinatorial games." In R. K. Guy, editor, *Combinatorial Games – Proceedings of Symposia in Applied Mathematics*, volume 43, pp. 183–189. Cambridge University Press, 1991.

[HDS07] G. Hollinger, J. Djugash, and S. Singh. "Coordinated search in cluttered environments using range from multiple robots." In *International Conference on Field and Service Robotics*, pp. 433–442, July 2007.

[HER00] D. F. Hougen, M. D. Erickson, P. E. Rybski, S. A. Stoeter, M. Gini, and N. Papanikolopoulos. "Autonomous mobile robots and distributed exploratory missions." In L. E. Parker, G. Bekey, and J. Barhen, editors, *Proceedings of the International Symposium on Distributed Autonomous Robotic Systems*, volume 4, pp. 221–230. Springer, 2000.

[HKS07] G. Hollinger, A. Kehagias, and S. Singh. "Probabilistic strategies for pursuit in cluttered environments with multiple robots." In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 3870–3876, April 2007.

[HKS08] G. Hollinger, A. Kehagias, S. Singh, D. Ferguson, and S. Srinivasa. "Anytime guaranteed search using spanning trees." Technical Report CMU-RI-TR-08-36, The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, August 2008.

[HMB05] H. Hexmoor, B. McLaughlan, and M. Baker. "Swarm control in unmanned aerial vehicles." In *Proceedings of the International Conference on Artificial Intelligence (IC-AI)*, pp. 911–917, 2005.

[HR03] A. Howard and N. Roy. "The robotics data set repository (Radish)." http://radish.sourceforge.net/, 2003.

[HS08] G. Hollinger and S. Singh. "Proofs and experiments in scalable, near-optimal search by multiple robots." In *Proceedings of Robotics: Science and Systems*, June 2008.

[HSD09] G. Hollinger, S. Singh, J. Djugash, and A. Kehagias. "Efficient multi-robot search for a moving target." *The International Journal of Robotics Research*, **28**(1):201–219, February 2009.

[HSK09] G. Hollinger, S. Singh, and A. Kehagias. "Efficient, guaranteed search with multi-agent teams." In *Proceedings of Robotics: Science and Systems*, Seattle, USA, June 2009.

[HV99] M. Henning and S. Vinoski. *Advanced CORBA programming with C++*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[IBD04] V. Isler, C. Belta, K. Daniiildis, and G. J. Pappas. "Hybrid control for visibility-based pursuit-evasion games." In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 2, pp. 1432–1437, 2004.

[IKK04a] V. Isler, S. Kannan, and S. Khanna. "Locating and capturing an evader in a polygonal environment." In *Proceedings of the Workshop on Algorithmic Foundations of Robotics*, pp. 351–367, 2004.

[IKK04b] V. Isler, S. Kannan, and S. Khanna. "Randomized pursuit-evasion with limited visibility." In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 1060–1069, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[IKK05] V. Isler, S. Kannan, and S. Khanna. "Randomized pursuit-evasion in a polygonal environment." *IEEE Transactions on Robotics*, **21**(5):875–884, 2005.

[ISS05] V. Isler, D. Sun, and S. Sastry. "Roadmap based pursuit-evasion and collision avoidance." *Proceedings of Robotics: Science and Systems*, 2005.

[JLM03] A. Jadbabaie, J. Lin, and A. Morse. "Coordination of groups of mobile autonomous agents using nearest neighbor rules." *IEEE Transactions on Automatic Control*, **48**(6):988–1001, 2003.

[JS01] B. Jung and G. S. Sukhatme. "Cooperative tracking using mobile robots and environment-embedded, networked sensors." In *IEEE International Symposium on Computational Intelligence in Robotics and Automation*, pp. 206–211, 2001.

[JS02] B. Jung and G. S. Sukhatme. "Tracking targets using multiple mobile robots: the effect of environment occlusion." *Autonomous Robots*, **13**(2):191–205, 2002.

[JS06] B. Jung and G. S. Sukhatme. "Cooperative multi-robot target tracking." In *Proceedings of the International Symposium on Distributed Autonomous Robotic Systems*, pp. 81–90, 2006.

[Kal60] R. Kalman. "A new approach to linear filtering and prediction problems." *Transactions of the ASME. Series D, Journal of Basic Engineering*, **82**:35–45, 1960.

[KBD03] Z. Khan, T. Balch, and F. Dellaert. "An MCMC-based particle filter for tracking multiple interacting targets." *Technical Report number GIT-GVU-03-35, Georgia Institute of Technology, Atlanta, GA*, 2003.

[KC07a] A. Kolling and S. Carpin. "Cooperative observation of multiple moving targets: an algorithm and its formalization." *The International Journal of Robotics Research*, **29**(9):935–953, 2007.

[KC07b] A. Kolling and S. Carpin. "Detecting intruders in complex environments with limited range mobile sensors." In K. Kozlowski, editor, *Robot Motion and Control, LNCIS 360*, pp. 417–426. Springer-Verlag London Limited, 2007.

[KC07c] A. Kolling and S. Carpin. "The GRAPH-CLEAR problem: definition, theoretical properties and its connections to multirobot aided surveillance." In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1003–1008, 2007.

[KC08a] A. Kolling and S. Carpin. "Extracting surveillance graphs from robot maps." In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2323–2328, 2008.

[KC08b] A. Kolling and S. Carpin. "Multi-robot surveillance: an improved algorithm for the Graph-Clear problem." In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 2360–2365, 2008.

[KC09a] A. Kolling and S. Carpin. "Probabilistic Graph-Clear." In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 3508–3514, 2009.

[KC09b] A. Kolling and S. Carpin. "Pursuit-evasion on trees by robot teams." *IEEE Transactions on Robotics*, 2009. accepted for publication.

[KC09c] A. Kolling and S. Carpin. "Surveillance strategies for target detection with sweep lines." In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5821–5827, 2009.

[KH05] M. Krishna and H. Hexmoor. "Resource allocation strategies for a multi sensor surveillance." In *Proceedings of the IEEE International*

Symposium on Collaborative Technologies and Systems, pp. 339–346, 2005.

[KHG09] A. Kehagias, G. Hollinger, and A. Gelastopoulos. "Searching the nodes of a graph: theory and algorithms." Technical Report ArXiv Repository 0905.3359 [cs.DM], Carnegie Mellon University, 2009.

[KHP04] K. M. Krishna, H. Hexmoor, S. Pasupuleti, and S. Chellapa. "A surveillance system based on multiple mobile sensors." In *In Proceedings of FLAIRS 2004 AAAI Press*, pp. 128–134, 2004.

[KHS05] K. M. Krishna, H. Hexmoor, and S. Sogani. "A T-step ahead constrained optimal target detection algorithm for a multi sensor surveillance system." In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1840–1845, 2005.

[KHS09] A. Kehagias, G. Hollinger, and S. Singh. "A graph search algorithm for indoor pursuit/evasion." *Mathematical and Computer Modelling*, **50**(9-10):1305–1317, 2009.

[KP86] M. Kirousis and C. H. Papadimitriou. "Searching and pebbling." *Theoretical Computer Science*, **47**(2):205–218, 1986.

[LaP93] A. S. LaPaugh. "Recontamination does not help to search a graph." *Journal of the ACM*, **40**(2):224–245, 1993.

[LBC98] T. Liu, P. Bahl, and I. Chlamtac. "Mobility modeling, location tracking, and trajectory prediction in wireless ATM networks." *IEEE Journal on Selected Areas in Communications*, **16**:922–936, 1998.

[LGB97] S. M. LaValle, H. H. González-Banos, C. Becker, and J.-C. Latombe. "Motion strategies for maintaining visibility of a moving target." In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 731–736, 1997.

[LH99] B. Liang and Z.J. Haas. "Predictive distance-based mobility management for PCS networks." volume 3, pp. 1377–1384, 1999.

[LH01] S. M. LaValle and J. Hinrichsen. "Visibility-based pursuit-evasion: the case of curved environments." *IEEE Transactions on Robotics and Automation*, **17**(2):196–201, April 2001.

[LL07] Y. Li and Y.-H. Liu. "Energy saving target tracking using mobile sensor network." In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 3653–3658, 2007.

[LLG97] S. M. LaValle, D. Lin, L. Guibas, J.-C. Latombe, and R. Motwani. "Finding an unpredictable target in a workspace with obstacles." In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 737–742, 1997.

[LSS02] S. M. LaValle, B. Simov, and G. Slutzki. "An algorithm for searching a polygonal region with a flashlight." *International Journal of Computational Geometry*, **12**(1-2):87–113, 2002.

[MB06] S. Martínez and F. Bullo. "Optimal sensor placement and motion coordination for target tracking." *Automatica*, **42**(4):661–668, 2006.

[MC94] G. F. Miller and D. Cliff. "Protean behavior in dynamic games: arguments for the co-evolution of pursuit-evasion tactics." In *Proceedings of the Third International Conference on Simulation of Adaptive Behavior: From Animals to Animats 3*, pp. 411–420, Cambridge, MA, USA, 1994. MIT Press.

[MCB07] S. Martínez, J. Cortés, and F. Bullo. "Motion coordination with distributed information." *Control Systems Magazine, IEEE*, **27**(4):75–88, August 2007.

[MHG88] N. Megiddo, S. L. Hakimi, M. R. Garey, D. S. Johnson, and C. H. Papadimitriou. "The complexity of searching a graph." *Journal of the ACM*, **35**(1):18–44, 1988.

[Mic04] O. Michel. "Cyberbotics Ltd - WebotsTM: Professional mobile robot simulator." *International Journal of Advanced Robotic Systems*, **1**(1):40–43, 2004.

[MJ05] M. Marzouqi and R. Jarvis. "Covert Robotics: Covert Path Planning in Unknown Environments." In *Proceedings of the Australasian Conference on Robotics and Automation (ACRA)*, 2005.

[MMA05] R. Murrieta-Cid, L. Munoz-Gomez, M. Alencastre-Miranda, A. Sarmiento, S. Kloder, S. Hutchinson, F. Lamiraux, and J. P. Laumond. "Maintaining visibility of a moving holonomic target at a fixed distance with a non-holonomic robot." In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2687–2693, 2005.

[MMH05] T. Muppirala, R. Murrieta-Cid, and S. Hutchinson. "Optimal motion strategies based on critical events to maintain visibility of a moving target." In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 3837–3842, 2005.

[MMH08] R. Murrieta-Cid, R. Monroy, S. Hutchinson, and J. P. Laumond. "A complexity result for the pursuit-evasion game of maintaining visibility of a moving evader." In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 2657–2664, 2008.

[MMS07] R. Murrieta-Cid, T. Muppirala, A. Sarmiento, S. Bhattacharya, and S. Hutchinson. "Surveillance strategies for a pursuer with finite sensor range." *The International Journal of Robotics Research*, **26**(3):233–253, 2007.

[MPS85] F. Makedon, C. H. Papadimitriou, and I. H. Sudborough. "Topological bandwidth." *SIAM Journal on Algebraic and Discrete Methods*, **6**(3):418–444, 1985.

[MRS05] M. Moors, T. Röhling, and D. Schulz. "A probabilistic approach to coordinated multi-robot indoor surveillance." In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3447–3452, 2005.

[MRV05] C. McMillen, P. E. Rybski, and M. Veloso. "Levels of multi-robot coordination for dynamic environments." *Multi-Robot Systems. From Swarms to Intelligent Automata Volume III*, **3**:53–64, 2005.

[MS83] F. Makedon and I. H. Sudborough. "Minimizing width in linear layouts." In *Proceedings of the 10th Colloquium on Automata, Languages and Programming*, pp. 478–490, London, UK, 1983. Springer-Verlag.

[MS89] F. Makedon and I. H. Sudborough. "On minimizing width in linear layouts." *Discrete Applied Mathematics*, **23**(3):243–265, 1989.

[MS03] R. Madhavan and C. Schlenoff. "Moving object prediction for off-road autonomous navigation." In *Proceedings of the SPIE Aerosense Conference*, volume 5083, pp. 134–145, 2003.

[MS06] M. Moors and D. Schulz. "Improved markov models for indoor surveillance." In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 4072–4077, 2006.

[MTH05] R. Murrieta-Cid, B. Tovar, and S. Hutchinson. "A sampling-based motion planning approach to maintain visibility of unpredictable targets." *Autonomous Robots*, **19**(3):285–300, 2005.

[Mur04] R. R. Murphy. "Rescue robotics for homeland security." *Communications of the ACM*, **47**(3):66–68, 2004.

[OGB07] K. J. Obermeyer, A. Ganguli, and F. Bullo. "Asynchronous distributed searchlight scheduling." In *Proceedings of the IEEE Conference on Decision and Control*, pp. 4863–4868, New Orleans, LA, 2007.

[OGB08] K. J. Obermeyer, A. Ganguli, and F. Bullo. "A complete algorithm for searchlight scheduling." *International Journal of Computational Geometry and Applications*, October 2008. Submitted.

[OR87] J. O'Rourke. *Art gallery theorems and algorithms*. Oxford Univeristy Press, 1987.

[ORS04] S. Oh, S. Russell, and S. Sastry. "Markov chain Monte Carlo data association for general multiple-target tracking problems." *Proceedings of the IEEE Conference on Decision and Control*, **1**:735–742, 2004.

[OS05] S. Oh and S. Sastry. "Tracking on a graph." In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, p. 26, Piscataway, NJ, USA, 2005. IEEE Press.

[OSS05] S. Oh, L. Schenato, and S. Sastry. "A hierarchical multiple-target tracking algorithm for sensor networks." In *Proceedings of the International Conference on Robotics and Automation*, pp. 2197–2202, April 2005.

[Par76] T.D. Parsons. "Pursuit-evasion in a graph." In Y. Alavi and D. R. Lick, editors, *Theory and Applications of Graphs*, volume 642, pp. 426–441. Springer Berlin / Heidelberg, 1976.

[Par78] T.D. Parsons. "The search number of a connected graph." In *Proceedings of the 10th Southeastern Conference Combinatorics, Graph Theory, and Computing*, pp. 549–554, 1978.

[Par02] L.E. Parker. "Distributed algorithms for multi-robot observation of multiple moving targets." *Autonomous Robots*, **12**:231–255, 2002.

[PLC01] S.-M. Park, J.-H. Lee, and K.-Y. Chwa. "Visibility-based pursuit-evasion in a polygonal region by a searcher." In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming (ICALP)*, pp. 456–468, 2001.

[PM07] J. Pugh and A. Martinoli. "The cost of reality: effects of real-world factors on multi-robot search." In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 397–404, 2007.

[Ram72] U. Ramer. "An iterative procedure for the polygonal approximation of plane curves." *Computer Graphics and Image Processing*, **1**(2):244–256, 1972.

[RSE00] P. E. Rybski, S. A. Stoeter, M. D. Erickson, M. Gini, D. F. Hougen, and N. P. Papanikolopoulos. "A team of robotic agents for surveillance." In C. Sierra, M. Gini, and J. S. Rosenschein, editors, *Proceedings of the Fourth International Conference on Autonomous Agents*, pp. 9–16, Barcelona, Catalonia, Spain, 2000. ACM Press.

[She92] T. Shermer. "Recent results in art galleries." *Proceedings of the IEEE*, **80**(9):1384–1399, 1992.

[Sie99] D. Siersma. "Voronoi diagrams and morse theory of the distance function." In *Geometry in Present Day Science, World Scientific*, pp. 187–208. World Scientific, 1999.

[Sko00] K. Skodinis. "Computing optimal linear layouts of trees in linear time." In *ESA '00: Proceedings of the 8th Annual European Symposium on Algorithms*, pp. 403–414, London, UK, 2000. Springer-Verlag.

[SLS02] B. Simov, S. M. LaValle, and G. Slutzki. "A complete pursuit-evasion algorithm for two pursuers using beam detection." In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 618–623, 2002.

[SOD02] M. Saptharishi, C. S. Oliver, C. Dihel, K. Bhat, J. Dolan, A. Trebi-Ollennu, and P. Kohsla. "Distributed surveillance and reconnaissance using multiple autonomous ATVs: Cyberscout." *IEEE Transactions on Robotics and Automation: Special Issue on Multi-Robot Systems*, **18**(5):826–836, 2002.

[SOS05] L. Schenato, S. Oh, S. Sastry, and P. Bose. "Swarm coordination for pursuit evasion games using sensor networks." In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 2493–2498, 2005.

[SRE00] S. A. Stoeter, P. E. Rybski, M. D. Erickson, M. Gini, D. F. Hougen, D. G. Krantz, N. Papanikolopoulos, and M. Wyman. "A robot team for exploration and surveillance: design and architecture." In *The Sixth International Conference on Intelligent Autonomous Systems*, pp. 767–774, Venice, Italy, July 2000.

[SRL04] S. Sachs, S. Rajko, and S. M. LaValle. "Visibility-based pursuit-evasion in an unknown planar environment." *The International Journal of Robotics Research*, **23**(1):3–26, January 2004.

[SRS02] S. A. Stoeter, P. E. Rybski, K. Stubbs, C. P. McMillen, M. Gini, D. F. Hougen, and N. Papanikolopoulos. "A robot team for surveillance tasks: design and architecture." *Robotics and Autonomous Systems*, **40**(2-3):173–183, September 2002.

[SSL00] B. Simov, G. Slutzki, and S. M. LaValle. "Pursuit-evasion using beam detection." In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 2, pp. 1657–1662, 2000.

[SSL09] B. Simov, G. Slutzki, and S. M. LaValle. "Clearing a polygon with two 1-searchers." *International Journal of Computational Geometry and Applications*, **19**(1):59–92, 2009.

[SSY90] K. Sugihara, I. Suzuki, and M. Yamashita. "The searchlight scheduling problem." *SIAM Journal on Computing*, **19**(6):1024–1040, 1990.

[SY92] I. Suzuki and M. Yamashita. "Searching for a mobile intruder in a polygonal region." *SIAM Journal on Computing*, **21**(5):863–888, 1992.

[TGL04] B. Tovar, L. Guilamo, and S. M. LaValle. "Gap navigation trees: minimal representation for visibility-based tasks." In M. A. Erdmann, D. Hsu, M. Overmars, and A. F. van der Stappen, editors, *Proceedings of the Workshop on Algorithmic Foundations of Robotics*, volume 17, pp. 425–440, 2004.

[Thr98] S. Thrun. "Learning Metric-Topological Maps for Indoor Mobile Robot Navigation." *Artificial Intelligence*, **99**(1):21–71, 1998.

[TL06] B. Tovar and S. M. LaValle. "Visibility-based pursuit-evasion with bounded speed." In *Proceedings of the Workshop on Algorithmic Foundations of Robotics*, pp. 475–489, 2006.

[TLM03] B. Tovar, S. M. LaValle, and R. Murrieta. "Locally-optimal navigation in multiply-connected environments without geometric maps." In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pp. 3491–3497, 2003.

[Vaz01] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2001.

[VFL09] D. Vasquez, T. Fraichard, and C. Laugier. "Growing Hidden Markov Models: an incremental tool for learning and predicting human and vehicle motion." *The International Journal of Robotics Research*, **28**(11–12):1486–1506, 2009.

[VRS01] R. Vidal, S. Rashid, C. Sharp, O. Shakernia, J. Kim, and S. Sastry. "Pursuit-evasion games with unmanned ground and aerial vehicles." In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 3, pp. 2948–2955, 2001.

[VSK02] R. Vidal, O. Shakernia, H. Kin, D. Shim, and S. Sastry. "Probabilistic pursuit-evasion games: theory, implementation and experimental evaluation." *IEEE Transactions on Robotics and Automation*, **18**(5):662–669, 2002.

[YTW05] W.-L. Yeow, C.-K. Tham, and W.-C. Wong. "A novel target movement model and energy efficient target tracking in sensor networks." *IEEE 61st Vehicular Technology Conference (VTC)*, **5**:2825–2829, 2005.

[YUS97] M. Yamashita, H. Umemoto, I. Suzuki, and T. Kameda. "Searching for mobile intruders in a polygonal region by a group of mobile searchers." In *Proceedings of the Thirteenth Annual Symposium on Computational Geometry*, pp. 448–450, 1997.