Solving Stochastic Orienteering Problems with Chance Constraints Using Monte Carlo Tree Search

Stefano Carpin Thomas C. Thayer

***Abstract*— We present a new Monte Carlo Tree Search (MCTS) algorithm to solve the stochastic orienteering problem with chance constraints, i.e., a version of the problem where travel costs are random and one is given a bound on the tolerable probability of exceeding the budget. The algorithm we present is online and anytime, i.e., it alternates planning and execution and the quality of the solution it produces increases as the allowed computational time increases. Differently from most former MCTS algorithms, for each action available in a state the algorithm maintains estimates of both its value, and the probability that its execution will eventually result in a violation of the chance constraint. Then, at action selection time, our proposed solution prunes away trajectories that are estimated to violate the failure probability. Extensive simulation results show that this novel approach is capable of producing solutions better than former work, while offering an anytime performance.**

I. Introduction

Orienteering is a combinatorial optimization problem that can be used to model numerous problems relevant to robotics and automation, such as logistics [14], environmental monitoring [23], surveillance [10], and precision agriculture [20], just to name a few. In its basic formulation, one is given a graph G with rewards associated with vertices and costs assigned to edges, as well as a budget B. The goal is to find a path collecting the highest sum of rewards of the visited vertices, while ensuring that the path cost does not exceed the budget B. In this paper, for sake of simplicity, we assume that B is a temporal budget, but it could as well be energy or any other resource consumed by the robot as it moves from location to location. Numerous variants have been proposed and, as discussed in section II, most are computationally intractable. In robotics applications this model is usually adopted in scenarios where vertices are associated with places that a robot must visit (e.g., to pick a package, or to deploy a sensor, collect a sample, etc.), and edges are associated with paths connecting the different locations, with the edge cost modeling the time or energy spent by the robot to traverse it. Most former research in this area has considered scenarios where costs and rewards are deterministic, and only few works have explicitly considered cases where these are affected by uncertainty. In most practical applications, however, the time or energy spent to traverse an edge (i.e., to move from one location to another) is not known upfront, but is rather a continuous random variable whose realization will only be known at run time. When this is the case, it follows that for a given path the cost is also a random variable. For example, the robot may have to wait to traverse an aisle in a warehouse to give way to another robot coming in the opposite direction, or it may have to take a detour because a passage is blocked, and so on. The variant of the problem where edge costs are stochastic is known as the *stochastic orienteering problem*, and while studied in the past [4], it has received far less attention than its deterministic version. Obviously, the stochastic version is not simpler than the deterministic one. Assuming that the probability density function (pdf) characterizing the traversal time of edges is known, some former solutions reduced the stochastic problem to the deterministic one by using the expected traversal time. Such solutions are in many situations unsatisfactory, because optimizing for expectation may lead to realizations where the robot exceeds the budget B while executing the task. This may be a major inconvenience, because if the robot runs out of energy and stops, it has to be recovered and recharged.

In our recent works [16]–[18] we presented a new approach to solve the stochastic orienteering problem whereby we introduce *chance constraints*, i.e., while solving the problem we consider a constraint on the probability that the stochastic cost of the path exceeds the assigned budget. It shall be mentioned that besides our works, there is very little former literature attacking this specific problem, as discussed in Section II. Differently from all previous works, our recently developed algorithms produce a *path policy*, i.e., a time-parametrized schedule that, depending on how much budget is left, determines where to move next while ensuring that the probability of exceeding the budget remains bounded below an assigned constant. This approach is therefore adaptive, i.e., rather than producing one path, it gives a policy that will result in different paths depending on the actual realizations of the edge traversal times. This is achieved by reducing the planning problem to a constrained Markov Decision Process (CMDP) with a suitable structure. This approach, while effective, has some limitations. First, the path policy is determined using an initial solution to the deterministic version of the orienteering problem. This initial solution is computed using a heuristic, and if the heuristic selects a poor path, the algorithm has no way to move away from it. Second, in constructing the finite CMDP, the continuous temporal dimension capturing the time left is discretized into time intervals. This generates a

S. Carpin is with the Department of Computer Science and Engineering, University of California, Merced, CA, USA. T. Thayer is with SLAC National Accelerator Laboratory, Menlo Park, CA, USA. This work is partially supported by the USDA-NIFA under award # 2021-67022-33452 and by the National Science Foundation (NSF) under Cooperative Agreement EEC-1941529. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the U.S. Department of Agriculture or the NSF.

tradeoff, where a finer grain discretization generates a larger state space for the associated CMDP, and this translates to increased computation time. Moreover, the discretizazion naturally leads to an approximation of the underlying transition probabilities in the state space.

In an effort to overcome these limitations of former solutions, in this paper we present a completely different approach to solve the stochastic orienteering problem with chance constraints. The main idea is to repeatedly search the space of possible paths using a Monte Carlo Tree Search (MCTS) approach in an online fashion. Following a typical rolling horizon approach, after an initial path has been identified, only the first segment is executed. Then, based on the *actual* time spent to execute the first motion, the budget is updated and the method re-run. Additionally, to account for the chance constraint, we introduce a novel *backup* procedure based on Monte-Carlo sampling that allows eliminating from the search tree the paths that would violate the budget constraint. Moreover, the search in the tree is informed by a novel criterion we dub UCTF (Upper-bound Confidence for Trees with Failures) that extends the widely used UCT formulation. The original contributions of this paper are the following:

- we formulate the stochastic orienteering problem with chance constraints as an MCTS planning problem;
- we introduce novel tree policies and backup policies to incorporate and manage the probability of violating the given constraint;
- we demonstrate that this approach is superior to our former solutions.

The rest of the paper is organized as follows. Section II discusses selected related work, while detailed background on stochastic orienteering and MCTS algorithms is provided in section III. Our new algorithm is introduced in section IV, and then experimentally evaluated in section V. Conclusions and future work are then presented in section VI.

II. Related Work

The deterministic orienteering was first formalized in [8] where it was also shown to be NP-hard. Consequently, most solutions to the problem relied on heuristic approaches [9], [19]. Exact solutions for limited size instances can be found using integer programming formulations [7], while approximate solutions have been proposed but have seen limited use [5]. Stochastic variants of the problem can encompass stochastic costs for the edges, or stochastic rewards for the vertices. As the deterministic orienteering problem is a special case of stochastic orienteering, it follows that stochastic orienteering is NP-hard, too. In [4] the authors propose an exact solution for a special class of graphs, and various heuristics for general graphs, but do not consider chance constraints, i.e., bounds on the probability of exceeding the buget. The works presented in [22] and [21] tackle a problem similar to ours, inasmuch as they consider a risk-sensitive formulation for the stochastic orienteering problem. In [22] the authors propose an algorithm to solve it based on local search, while in [21] the authors propose a mixed integer program based on sample average approximation. These solutions are fundamentally different from the one we propose because they are formulated offline a priori, and not updated as the mission unfolds based on the travel costs experienced during the mission. This may lead to excessively conservative solutions that collect less reward on average (think for example to the case when one follows a predetermined path where traversal times are much lower than expected.) Our previous works [16]–[18] are the first ones to propose the concept of a *path policy* while solving the stochastic orienteering problem with chance constraints. While the computation is still offline, rather than computing a single path, these methods produce a set of rules that can be queried at runtime to determine which vertex to visit next based on the remaining budget. Therefore these solutions will return different paths depending on the specific realization of the stochastic processes governing travel times along the edges.

Monte Carlo Tree Search (MCTS) encompasses a family of any-time methods to solve planning problems using generative models. Albeit a mature technique [3], [6], [12], MCTS gained significant popularity while being recently used in combination with reinforcement learning, most notably in [13]. The use of MCTS algorithms for problems with chance constraints has been so far limited. In [1] the authors propose an algorithm for chance constrained Markov Decision Processes that is guaranteed to return a policy satisfying the chance constraint.

III. Background

In this section we shortly provide the formal definition of the stochastic orienteering problem (SOP) with chance constraints, as well as relevant information regarding MCTS algorithms. The reader is referred to the cited papers for more details.

A. Stochastic Orienteering with Chance Constraints

The deterministic orienteering problem is defined as follows. Let $G = (V, E)$ be a directed graph with n vertices, $r : V \to \mathbb{R}^+$ be a reward function defined over the set of vertices, and $c : E \to \mathbb{R}^+$ be a cost function defined over the edges. Let $v_s \in V$ and $v_g \in V$ be the start and goal vertices, respectively. In the following, without loss of generality we assume G is a complete graph (when this is not the case one can simply add all missing edges and set their costs equal to the sum of the costs along the shortest path.) For a path $\mathcal{P}$ over G connecting v_s to v_g, the reward of the path $R(\mathcal{P})$ is the sum of rewards for the visited vertices, with the stipulation that if a vertex is visited more than once, then its reward is collected just once. The cost of the path $C(\mathcal{P})$ is instead the sum of the costs of the edges along $\mathcal{P}$, but in this case if an edge appears multiple times, its corresponding cost is charged every time. For a given budget $B > 0$, the orienteering problem asks to solve the following constrained optimization problem

$$\mathcal{P}^* = \arg\max_{\mathcal{P}\in\Pi} R(\mathcal{P}) \qquad \text{s.t. } C(\mathcal{P}) \leq B$$

where Π is the set of all possible paths connecting v_s with v_g. In the stochastic version of the problem, the cost of each edge (v_i, v_j) is not a constant, but rather a continuous random variable with a known probability density function (pdf) with positive support and finite expectation. In general, each edge may have a different pdf. The availability of these density functions is essential in our method, because samples drawn from the distributions will be used to determine the probability of violating the budget constraint. This assumption is also consistent with MCTS literature where a generative model is assumed to be fully known to implement the rollout step described later on. In the following, for the stochastic version of the problem $c(v_i, v_j)$ is the expectation of the random variable associated with the edge (v_i, v_j). In this case, for a given path $\mathcal{P}$ the corresponding cost $C(\mathcal{P})$ is therefore also a random variable given by the sum of the random variables associated with the edges appearing along the path. Given a fixed failure probability P_f, the stochastic orienteering problem with chance constraints (SOPCC in the following, for brevity) asks to solve the following constrained optimization problem:

$$\mathcal{P}^* = \arg\max_{\mathcal{P}\in\Pi} R(\mathcal{P}) \qquad \text{s.t. } \Pr[C(\mathcal{P}) > B] \leq P_f$$

i.e.., we now constrain the probability that the cost of the path exceeds the budget B. When solving this problem, one can compute the solution off-line, i.e., before its execution starts or online, i.e., it may adapt the path based on the available residual budget. This is the approach we followed in [16]–[18], where we computed a *path-policy* that would guide the robot through the vertices while being aware of the remaining budget. In these works the policy is computed off-line, but it features a family of paths and the selection of the path to follow is done on-line. Alternatively, one can opt for an online approach, where the path is continuously refined based on the time spent by the robot while traversing the edges. In this case, MCTS offers an interesting approach.

B. Monte Carlo Tree Search

MCTS is an approach to solve decision making problems in an online fashion, where planning and execution alternate. It belongs to the family of receding-horizon (also called roll-out) methods [2], whereby one solves the planning problem using a finite time-horizon, but then executes just the first action in the plan, and then re-plans from scratch based on the outcome of the first action. In MCTS planning, the algorithm builds a rooted tree whose root node represents the current state, and whose edges connect states that can be reached through the execution of a single action. In the case of orienteering, each state represents a vertex in the graph. Node $b \in V$ in the tree can be a descendant of node $a \in V$ if (a, b) is an edge in E. Indeed, in the orienteering problem an action corresponds to moving from one vertex to another one. Key elements in MCTS algorithms are the following:

1) a selection process to move from the root of the tree down to a leaf following a so-called *tree policy*;
2) an *expansion* step executed to add leaf nodes to the tree;
3) a *rollout policy* to be executed from a leaf to establish how "good" a leaf is;
4) a *backup policy* to be executed from the leaves back to the root to guide the eventual selection of the best action from the root.

After the tree is built, an action is selected among those available in the root node. The action is executed, and the tree is discarded and rebuilt having as root the vertex reached after having executed the action. The reader is referred to [15] (chapter 8 and references therein) for more details. As pointed out in [11], different tree policies (step 1) and backup policies (step 4) may have a dramatic impact on the performance of MCTS based planning. In particular, while UCT (Universal Confidence bound for Trees) [12] is often considered the standard tree policy, it is not directly applicable to our problem because actions yielding high value (adding a high reward for the path in the orienteering problem) may also increase the probability of violating the chance constraint $\Pr[C(\mathcal{P}) > B] \leq P_f$ if they have high edge cost. For this reason, we propose an alternative tree policy based on UCT, but factoring in also failure probabilities (we call this tree policy UCTF – UCT with Failures). Similarly, backup strategies based on plain Monte Carlo averaging are not applicable because they do not consider whether constraints are violated or not. Inspired by the complex backup strategies studied in [11], in this work we instead propose a backup policy that explicitly considers failure constraints.

IV. An Online MCTS Algorithm for Stochastic Orienteering with Chance Constraints

The online algorithm we propose alternates planning and execution (see algorithm 1). Throughout its execution, the search tree expansion is conditioned on the residual budget B which is updated after each action is selected and executed. Algorithm 1 sketches the overall approach. At the first iteration the algorithm solves the SOPCC with the assigned budget B and root node v set to v_s. The solution of SOPCC defines the first action to take, i.e., the robot moves from v_s to the vertex $next_v$ returned by SOPCC and incurs a random travel cost c_r. The budget is then updated by setting $B = B - c_r$ and the SOPCC is solved again with the updated budget B and with starting vertex v set to $next_v$. The process repeats until either the final vertex v_g is reached, or the budget is completely spent. In this last case, the run is considered a failure.

A first advantage of this solution is that, differently from our cited works [16]–[18] it is neither necessary to discretize the temporal dimension to build a CMDP with a finite state space, nor it is necessary to numerically approximate the transition probabilities between states.

The MCTS algorithm to solve the SOPCC (line 3 in algorithm 1) customizes the general MCTS approach described in section III as follows. To each vertex $v \in V$ we associate a set of actions, i.e., the set of vertices that can be directly reached from v. The tree $\mathcal{T}$ we build is rooted at the vertex where the robot is currently positioned at, and

Algorithm 1: Alternating Planning and Execution

Data: $G = (V, E)$, $v_s, v_g \in V$, B

$v \leftarrow v_s$
while $B > 0$ **and** $v \neq v_g$ **do**
  $next_v \leftarrow$ SOPCC(v, B)
  move to vertex $next_v$ and let c_r be the incurred cost
  $B \leftarrow B - c_r$
  $v \leftarrow next_v$
end
if $B > 0$ **then**
  return Success
else
  return Failure
end

is parametrized by the available budget B. Therefore, all quantities stored in $\mathcal{T}$ are relative to the available budget B. There is a one-to-one correspondence between nodes in the tree $\mathcal{T}$ and vertices in the graph. Each node in the tree may have from 0 to $n-1$ children. Vertex v_j can be a child of v_i in the tree only if there is an edge connecting v_i with v_j, i.e., if there is an action from v_i that leads to v_j. Figure 1 shows a tree associated with a simple graph with five vertices.

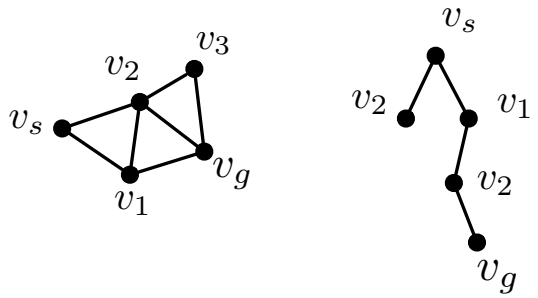

Fig. 1: The right side of the figure shows a possible MCTS tree $\mathcal{T}$ associated with the simple graph on the left and rooted in v_s. Vertices v_1 and v_2 are children of v_s because they are directly connected to it. Executing action v_1 from v_s means moving from v_s to v_1. Vertex v_3, not appearing in the tree, cannot be a child of v_s because it is not directly connected to it. Observe that v_2 appears as a child of both v_s and v_1 because it is connected to both, but it occurs along two different paths from the root. All paths in $\mathcal{T}$ from v_s to a leaf encode a possible path in G. In this simple example there are two paths, namely v_s, v_2, and v_s, v_1, v_2, v_g. Note that while the MCTS is being built not all paths must end at the goal vertex v_g.

For a given path from the root to a leaf, one can simulate the time it takes to traverse it by adding random samples drawn from the known pdfs associated with the edges along the path. For every internal node v_i, we store three attributes for each of its children v_j:

- $N[v_j]$ is the number of times that action v_j was attempted from v_i;
- $Q[v_j]$ is the expected reward associated with the *feasible* path of maximum reward with that selects v_j from v_i, if it exists. Feasible, in this context, means that the estimated failure probability does not exceed the assigned bound P_f. If all paths connecting v_i to v_j violate the failure probability P_f, then $Q[v_j]$ is set to the expected reward associated with the paths starting from v_i and going through v_j.
- $F[v_j]$ is the estimated failure probability of the path defining the value $Q[v_j]$ just defined. Therefore, when the path is feasible $F[v_j] \leq P_f$.

These three quantities are incrementally updated as the tree is being built and expanded, and the specifics will be given when discussing the backup policy. As stated formerly, the attributes $Q[v_j]$ and $F[v_j]$ associated with node v_i are not absolute, but are rather a function of the budget B specified when building the tree.

As pointed out in the previous section, there are various elements needed to implement an MCTS algorithm. In our algorithm these are as follows.

Tree Policy: The tree policy is used to traverse the tree from the root to a leaf and then select a new node to possibly add to the tree. It is implemented by applying recursively the following strategy inspired by UCT. Assuming v_i is the current vertex, to each of its neighbors[1] v_j we associate the following quantity (UCTF stands for UCT with Failure):

$$UCTF(v_j) = Q[v_j](1 - F[v_j]) + z\sqrt{\frac{\log t}{N[v_j]}}$$

where t is the sum of the number of times that the descendants of v_i have been explored already, and z is a constant. The node with the highest UCTF value is then selected and this process is repeated until a vertex not yet in the tree is selected. As commonly done in the basic UCT strategy, if node v_j has not yet been visited its $N[v_j]$ counter is 0 and we then set its UCTF value to ∞, to make sure all neighbors are visited at least once. The novel term $Q[v_j](1 - F[v_j])$ is the *expected utility* of moving to v_j, obtained by multiplying the estimated utility $Q[v_j]$ by the probability of success $(1 - F[v_j])$. Note that as failure is a binary random variable, this is the expected utility. In this way, given two vertices with similar Q values, the criterion favors the one with the lowest failure probability. The last term in the UCTF formula is borrowed from UCT and encourages the selection of nodes that have been formerly selected few times.

Rollout Policy: when a new vertex v_j is added as a descendant of a node v_i already in the tree it is necessary to estimate and how much reward one will collect by expanding the path through v_j, as well as to estimate of the probability of exceeding the available budget before reaching the end vertex v_g. These values will be stored in $Q[v_j]$ and $F[v_j]$, respectively. Figure 2 shows how the rollout process is implemented.

We generate S paths from v_j using the following strategy. The first step is to sample the time ψ it takes to proceed from the root node in $\mathcal{T}$ to v_j. We then set $B' = B - \psi$ as the residual budget available when starting from v_j. Then we execute the following two steps:

[1]From the set of neighbors we exclude the set of vertices already visited, because revisiting an already visited vertex does not give any reward and is therefore useless. In this context the terms neighbor and descendant are to considered synonyms.

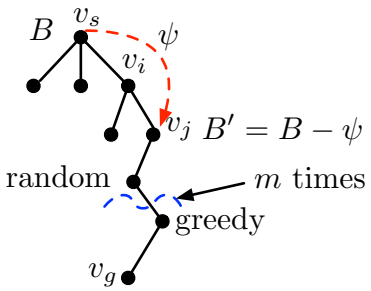

Fig. 2: Assuming the tree is rooted in v_s and the available budget is B, vertex v_j is selected by UCTF as the action to execute from v_i. The following process is then repeated S times. First sample the time ψ to move from v_s to v_j and set $B' = B - \psi$. Then, create a path from v_j first adding a random node and then adding nodes using the greedy criterion. By sampling the time to traverse the edges from v_j to v_g one can then estimate the probability of exceeding the residual budget B'.

1) first pick a random vertex among those that are neither visited, nor among the path from the root v_s to v_j;
2) next, repeatedly pick a vertex using the greedy criterion described below until the end vertex v_g is selected.

Once such a path p is obtained, one can compute its reward as the sum of the rewards of the vertices along p. The average of these S rewards provides an estimate for $Q[v_j]$. Similarly, one can use the pdfs of the edges along the path to generate samples for the costs of the edges along the path. If the sum of these samples exceeds the residual budget B', then the path is considered a failure. Dividing the number of failures by S, we get an estimate of the failure probability $F[v_j]$. The greedy strategy in step 2 above works as follows. For all vertices v_k not yet visited, compute $r(v_k)/c(v_l, v_k)$ where v_l is the last vertex added to the path p being built. Then, discard the vertices such that $\Pr[c(v_l, v_k) + c(v_k, v_g) > B'] > P_f$, and pick the one with the highest ratio. To determine the vertices to discard, we generate S samples for the costs of the edges use these values to estimate the probability of exceeding the budget B'. As the name suggests, the greedy step adds to the path the vertex v_k with the highest ratio between reward and cost, but constrained on having estimated that the probability of moving from v_k to the terminal v_g does not exceed the failure probability P_f. Note that the greedy step may select v_g as the most suitable node to visit next. When this happens, the rollout stops.

Backup Policy: After the S paths from v_j have been generated by the rollout procedure, one can compute the expected return $Q[v_j]$ and estimate $F[v_j]$, i.e., the probability of failure of expanding the route from v_i through v_j. Such values must then be propagated backwards thorough the tree towards the root, as the paths generated from v_j are the end points of paths starting from the root and therefore influence the values for the labels Q and F associated with the children of the root note. In MCTS without constraints this often done by averaging the returns, but the case considered in this work is different because of our need to generate actions that will lead to paths not violating the budget. The backup step is then applied from node v_j backwards towards the root always considering the relationship between the Q and F labels of the newly added node v_j and the Q and F labels of its parent node v_i. The step also involves the parent of v_i, if it exists (called v_k in the following). We refer to figure 3 while explaining the process.

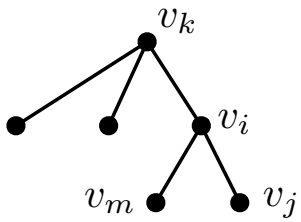

Fig. 3: When backing up the values $F[v_j]$ and $Q[v_j]$ associated with node v_i, it is necessary to consider their relationships with the values $F[v_i]$ and $Q[v_i]$ associated with v_k.

First, if v_i is the root then v_k does not exist and the backup step simply stores the values $F[v_j], Q[v_j]$ and $N[v_j]$ with v_i and stops. Next, let us assume that v_k exists, and has its labels $F[v_i], Q[v_i]$ and $N[v_i]$ initialized already. We distinguish the following cases:

- if $F[v_i] \geq F[v_j]$ and $Q[v_i] \leq Q[v_j] + r(v_i)$ then a better solution from v_k through v_i has been found. $F[v_i]$ is set to $F[v_j]$ and $Q[v_i]$ is set to $Q[v_j] + r(v_i)$. The update is made because we found a new path that has higher reward and lower failure probability.[2]
- if $F[v_i] < F[v_j]$ and $Q[v_i] \leq Q[v_j] + r(v_i)$, and $F[v_j] < P_f$, then we also set $F[v_i]$ to $F[v_j]$ and $Q[v_i]$ is set to $Q[v_j] + r(v_i)$. In this case the new path from v_k to v_i to v_j has higher failure probability than the former best path from v_k to v_i, but this is still below the constant P_f. Since its reward is higher than the previously best known path through v_i the value is propagated backwards.
- in all other cases, the values $F[v_j]$ and $Q[v_j]$ are stored at v_i but not propagated backwards.

When the values are updated, the process is repeated with the same rules backwards towards the root, until it is eventually reached and the process terminates, or the third case above applies. Finally, irrespective of which of the above cases applies, all the N values from v_i backards to the root are increased by 1 to record that those actions have been tried.

Action Selection: after the tree $\mathcal{T}$ has been built, the best action available from the root node v is selected. The best action is defined as the child node v_j with the highest value $Q[v_j]$ subject to the constraint that $F[v_j] \leq P_f$. If no such node exists (i.e., all nodes connected to the root have an F value exceeding the failure probability P_f), then action selection returns v_g, i.e., it tries to move the robot to the final vertex v_g in the orienteering graph V.

Algorithm 2 shows how the components described above are put together. As usual in MCTS algorithms, the tree is

[2] $F[v_i]$ may be larger than $F[v_j]$ because if was formerly set while exploring a node sibling to v_j like for example v_m in the figure.

expanded through a fixed number of iterations K. At each iteration, the UCTF criterion is used as a tree policy to move from the root of the tree to a node v_j (line 3) that is added to the tree it it is not already present. Then, S paths are generated from v_j (line 5) using the rollout process formerly described (line 8). Note that at each rollout the algorithm considers a different residual budget B' obtained by sampling the time ψ to move from the root of the tree to the new node v_j (line 6). After the S samples are collected, the values $Q[v_j]$ and $F[v_j]$ for v_j can be computed (line 10) and propagated back to the root with the backup procedure (line 11). Finally, the algorithm returns the action from the root with the highest Q value among those not exceeding the failure probability F (line 13).

Algorithm 2: SOPCC

```
   Data: start vertex v, B
1  Initialize tree T with root equal to v
2  for K iterations do
3      v_j ← UCTF(v)
4      add v_j to the tree if not present
5      for S iterations do
6          ψ ← SampleTraverseTime(v, v_j)
7          B' ← B − ψ
8          path ←rollout(v_j, B')
9      end
10     compute Q[v_j] and F[v_j] based on the S paths
11     Backup(v_j, Q[v_j], F[v_j])
12 end
13 return ActionSelection(root(T))
```

A. Properties

True to the MCTS spirit, the presented algorithm is an anytime algorithm, i.e., by increasing the value of the parameters K and S the quality of the returned solution increases. This is in contrast to our former works where a solution is produced only after the whole state space for the CMPD has been constructed and the associated linear program solved. In section V we will assess how K influences the quality of the solution and the computation time. The other relevant parameter is S, the number of samples used to estimate the failure probability of a path from the root to a leaf. Obviously, the larger the number of samples, the more accurate the estimate and the associated computation time.

Among the trajectories generated from the root, SOPCC is guaranteed to return an action associated with a trajectory that is estimated not to violate the failure constraint P_f. This is ensured by ActionSelection. If none of the generated trajectories satisfy the failure probability, for simplicity we return the terminal action v_g (move to the last vertex), but alternatively one could further extend the tree by running SOPCC again and extend the current tree rather than restarting from scratch.

Finally, as common for these type of algorithms, the probability of not finding the solution tends to 0 as the values of K and S increases. The convergence velocity is influenced by the size of the search space, i.e., by the average branching factor in the tree.

V. Results

In this section we provide two types of results. First we assess the sensitivity of the proposed algorithm to the parameters K (number of iterations) and S (number of simulations to estimate failure probability). In all our simulations we kept the parameter z (coefficient in the UCTF formula) equal to 3. Next, we make some comparisons with our recently proposed algorithms. To ease the comparison with our previous work, we consider the same setup to generate test cases. The n vertices in graph are randomly generated inside the unit square and rewards are sampled from a uniform distribution with support $[0, 1]$. All graphs are complete, i.e., $(v_i, v_j) \in E$ for each $v_i \neq v_j$. The random cost associated with edge (v_i, v_j) is

$$\alpha d_{i,j} + \mathcal{E}\left(\frac{1}{(1-\alpha)d_{i,j}}\right)$$

where $d_{i,j}$ is the Euclidean distance between v_i and v_j and $\mathcal{E}(\lambda)$ is random sample from the exponential distribution with parameter λ. This formulation ensures that the expected cost to traverse (v_i, v_j) is equal to $d_{i,j}$ and the cost is non-negative. In all our experiments the parameter α is set to 0.5. Before discussing the results, it is worth outlining that complete graphs are the most challenging to deal with because every node in the tree has the maximum possible branching factor, and for n vertices there are $\mathcal{O}(n!)$ possible paths[3] in the space of possible policies.

Figure 4 shows how the computational time grows with the number of iterations K (red line) as well as the standard deviation. The figure was collected for a problem instance with 10 vertices and data were averaged over 50 independent executions. The trend is roughly linear, as expected, and therefore one can accordingly tune the number of iterations based on the available time. Variations in the computation time emerge because the produced paths may have more or less vertices depending on the realizations of the stochastic travel times. It is worth observing that this time is not all spent upfront, but rather distributed along the path (see algorithm 1).

Next, we investigate how the number of iterations K in algorithm 2 influences the collected reward. Figure 5 shows how the amount of collected rewards changes with K (with all other parameters fixed.) for a graph with 20 vertices (red line) 30 vertices (blue line) and 40 vertices (orange line). Data is averaged over 50 runs with K varying from 1000 to 20000. In all instances the reward barely grows with the number of iterations, showing that already with a value of K below 4000 the algorithm displays a good performance. For larger graphs with a larger branching factor, with larger values of K one can expect a continued increase in the accrued reward, but this comes at the cost of increased computational time, as shown in figure 4. Overall, this figure seems to indicate that the algorithm is not too sensitive to the value of K (this would of course not be the case when K is decreased to smaller values now shown in the figure,

[3]The number is less than $n!$ because the start and final vertices are assigned.

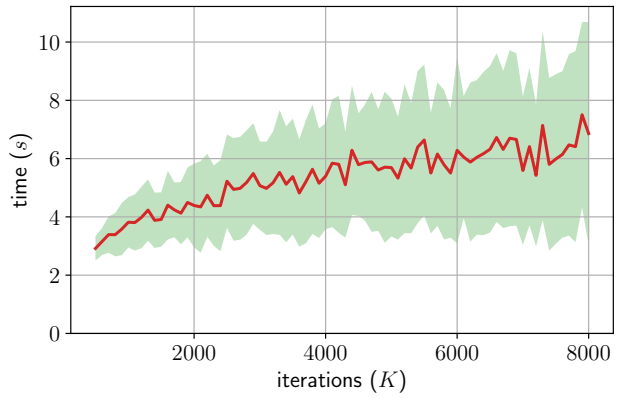

Fig. 4: Computation time as a function of the number of iterations K (red line). The green area shows the standard deviation (data averaged over 50 trials for each value of K). The chart refers to a graph with 10 vertices and $S = 100$ samples.

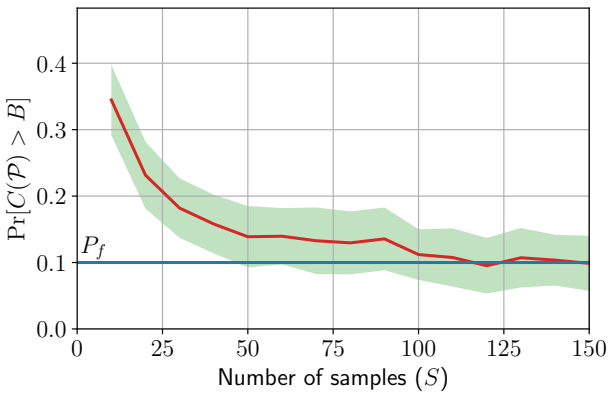

Fig. 6: Probability of failure as a function of the samples size S for a graph with 20 vertices, $K = 1000$ and $P_f = 0.1$. The green area shows the standard deviation (data averaged over 50 trials for each value of S).

as it would not have the ability to sufficiently explore the set of paths.)

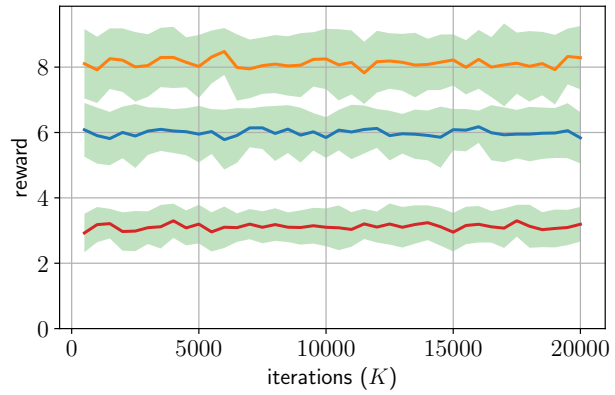

Fig. 5: Reward as a function of the number of iterations K. The orange line shoes the shows the trend for 40 vertices, the blue shows the reward trend for 30 vertices and the red shows the trend for 20 vertices. The green area shows the standard deviation (data averaged over 50 trials for each value of K). In all instances $S = 100$ samples.

Finally, figure 6 shows how the probability of exceeding the budget B varies with the number of samples S used to estimate the time to traverse a path. In this specific case, the assigned failure probability was $P_f = 0.1$. As the number of samples increases, the probability of failure decreases, as expected. This will further decrease as the parameter K increases, because a larger part of the search space is searched (the chart was produced with $K = 1000$ which is a rather low number.)

Having assessed the sensitivity of the algorithm to its parameters, we next compare its performance with the CMDP based planner we formerly proposed and discussed earlier. Tables I and II compare the performance of the two approaches. Based on the results outlined so far, in all tests we set $K = 2000$ and $S = 100$, although one could further fine tune them using higher numbers when the number of vertices grow or the failure probability P_f decreases. The table displays average results (50 executions for MCTS, 20 for CMDP). Detailed time comparisons between the two algorithms are not provided because one is implemented in Python and the other Matlab, and therefore a comparison would not be principled. Both are however suitable for real-time performance, and both algorithms satisfy the given failures probabilities, i.e., on average their failure probability does not exceed P_f. The tables clearly show that MCTS exceeds the performance of the CMDP algorithm and the gain is at times notable. Gains are particularly notable for $P_f = 0.05$ and $B = 2$ which is the hardest combination (low failure probability, small budget). Note that for the case $n = 10$ and $B = 3$ the two algorithms perform more or less the same because they both manage to find the same class of paths. Indeed, in that case it is most of the time possible for the agent to visit all 10 vertices and collect all the reward, and both algorithms determine that solution. In all other cases, however, the MCTS approach is the clear winner.

	$P_f = 0.05$		$P_f = 0.1$	
n	MCTS	CMDP	MCTS	CMDP
10	2.4366	1.7766	2.6738	2.3187
20	2.8492	2.3127	3.0796	2.9767
30	5.3748	2.3583	5.9764	5.0151
40	7.6865	5.9890	8.3003	7.1816

TABLE I: Average reward collected for a budget $B = 2$ and different failure probabilities.

	$P_f = 0.05$		$P_f = 0.1$	
n	MCTS	CMDP	MCTS	CMDP
10	3.0827	3.0545	3.0901	3.0802
20	4.6245	4.0613	5.1480	4.5862
30	7.7808	6.8688	7.9830	7.6081
40	10.5332	9.0803	10.6084	9.6643

TABLE II: Average reward collected for a budget of $B = 3$ and different failure probabilities.

VI. Conclusions and Future Work

In this paper we have presented a new algorithm to solve the stochastic orienteering problem with chance constraints. The main novelty is in using an MCTS approach to explore the space of possible policies and a generative model for the travel times across the edges to prune away realizations that violate the assigned bound on the probability of failure. To the best of our knowledge, this approach is novel. The proposed algorithm offers various advantages. By being an anytime algorithm, it can produce results within a given pre-assigned computational time, while previous methods will not produce any results until the associated linear program is built and solved. In addition, we have shown that on a variety of test cases our method outperforms our previous solution.

There are various venues for further research. First, the current rollout policy relies on a simple greedy strategy. It would be interesting to explore whether different rollout policies (e.g., policies using some of the heuristic methods fomerly proposed for the orienteering problems) would prouce better results (either in terms of speed or collected rewards). Next, it will be interesting to investigate whether this approach could be extended to study other planning problems with chance constraints where building a finite state space is disadvantageous because of the need to discretize continuous dimensions, such as time.

References

[1] B.J. Ayton and B.C. Williams. Vulcan: A Monte Carlo algorithm for large chance constrained MDPs with risk bounding functions. *arXiv*, (1809.01220), 2018.

[2] D.P. Bertsekas. *Rollout, Policy Iteration, and Distributed Reinforcement Learning*. Athena Scientific, 2020.

[3] C.B. Browne, E. Powley, D. Whitehouse, S.M. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[4] A.M. Campbell, M. Gendreau, and B.W. Thomas. The orienteering problem with stochastic travel and service times. *Annals of Operations Research*, 186(1):61–81, 2011.

[5] C. Chekuri, N. Korula, and M. Pál. Improved algorithms for orienteering and related problems. *ACM Transactions on Algorithms*, 8(3), July 2012.

[6] R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and Games*, pages 72–83, 2007.

[7] M. Fischetti, J. J. S. Gonzalez, and P. Toth. Solving the orienteering problem through branch-and-cut. *INFORMS Journal on Computing*, 10(2):133–148, 1998.

[8] B. L. Golden, L. Levy, and R. Vohra. The orienteering problem. *Naval Research Logistics*, 34:307–318, 1987.

[9] A. Gunawan, H. C. Lau, and P. Vansteenwegen. Orienteering problem: A survey of recent variants, solution approaches, and applications. *European Journal of Operational Research*, 255(2):315–332, 2016.

[10] S. Jorgensen, R. H. Chen, M.B. Milam, and M. Pavone. The team surviving orienteers problem: Routing robots in uncertain environments with survival constraints. In *International Conference on Robotic Computing*, pages 227–234, 2017.

[11] P. Khandelwal, E. Liebman, S. Niekum, and P. Stone. On the analysis of complex backup strategies in Monte Carlo Tree Search. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 1319–1328, 2016.

[12] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*, pages 282–293, 2006.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A.A. Rusu, J. Veness, M.G. Bellemare, A. Graves, M. Riedmiller, A.K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[14] F. Betti Sorbelli, S. Carpin, F. Coró, A. Navarra, and C.M. Pinotti. Optimal routing schedules for robots operating in aisle-structures. In *Proceedings of the IEEE Conference on Robotics and Automation*, pages 4927–4933, 2020.

[15] R.S. Sutton and A.G. Barto. *Reinforcement Learning – an introduction*. MIT Press, 2nd edition, 2018.

[16] T. Thayer and S. Carpin. An adaptive method for the stochastic orienteering problem. *IEEE Robotics and Automation Letters*, 6(2):4185–4192, 2021.

[17] T. Thayer and S. Carpin. A fast algorithm for stochastic orienteering with chance constraints. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 7398–7945, 2021.

[18] T. Thayer and S. Carpin. A resolution adaptive algorithm for the stochastic orienteering problem with chance constraints. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 6388–6395, 2021.

[19] T. Thayer, S. Vougioukas, K. Goldberg, and S. Carpin. Routing algorithms for robot assisted precision irrigation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2221–2228, 2018.

[20] T. Thayer, S. Vougioukas, K. Goldberg, and S. Carpin. Multi-robot routing algorithms for robots operating in vineyards. *IEEE Transactions on Automation Science and Engineering*, 17(3):1184–1194, 2020.

[21] P. Varakantham and A. Kumar. Optimization approaches for solving chance constrained stochastic orienteering problems. In *Algorithmic Decision Theory: Third International Conference, ADT 2013, Bruxelles, Belgium, November 12-14, 2013, Proceedings*, page 387–398. Springer-Verlag, 2013.

[22] P. Varakantham, A. Kumar, H. C. Lau, and W. Yeoh. Risk-sensitive stochastic orienteering problems for trip optimization in urban environments. *Transactions on Intelligent Systems and Technology*, 9(3), 2018.

[23] J. Yu, M. Schwager, and D. Rus. Correlated orienteering problem and its application to persistent monitoring tasks. *IEEE Transactions on Robotics*, 32(5):1106–1118, 2016.