Multi-goal path planning using multiple random trees

Jaroslav Janoš [ID], Vojtěch Vonásek [ID] and Robert Pěnička [ID]

Abstract—In this paper, we propose a novel sampling-based planner for multi-goal path planning among obstacles, where the objective is to visit predefined target locations while minimizing the travel costs. The order of visiting the targets is often achieved by solving the Traveling Salesman Problem (TSP) or its variants. TSP requires to define costs between the individual targets, which — in a map with obstacles — requires to compute mutual paths between the targets. These paths, found by path planning, are used both to define the costs (e.g., based on their length or time-to-traverse) and also they define paths that are later used in the final solution. To enable TSP finding a good-quality solution, it is necessary to find these target-to-target paths as short as possible. We propose a sampling-based planner called Space-Filling Forest (SFF*) that solves the part of finding collision-free paths. SFF* uses multiple trees (forest) constructed gradually and simultaneously from the targets and attempts to find connections with other trees to form the paths. Unlike Rapidly-exploring Random Tree (RRT), which uses the nearest-neighbor rule for selecting nodes for expansion, SFF* maintains an explicit list of nodes for expansion. Individual trees are grown in a RRT* manner, i.e., with rewiring the nodes to minimize their cost. Computational results show that SFF* provides shorter target-to-target paths than existing approaches, and consequently, the final TSP solutions also have a lower cost.

Index Terms—Motion and Path Planning; Planning, Scheduling and Coordination

I. Introduction

The task of multi-goal path planning is to find a collision-free path connecting several targets [27], which is required in

data collection [8], [19], active perception [2], [19], and manufacturing [27], [26]. Visiting multiple goals in the shortest possible time is crucial for systems with limited operational time like flying vehicles [22], e.g., for their recharging [18].

In the general version of multi-goal path planning, both the sequence of visiting the targets and also the trajectory connecting them have to be found such that the travel cost (e.g., traveled distance or execution time) is minimized. The classical approach, which is also considered in this paper, is to decouple the task to the combinatorial part (finding the sequence of targets to be visited) and to path planning part that connects the targets in the found order. The combinatorial phase is usually considered as an instance of Traveling Salesman Problem (TSP) and can be solved heuristically [19], [14].

Formulation of multi-goal path planning using the TSP requires knowledge about mutual reachability and trajectory cost between the targets. For robots moving among obstacles, path planning between all pairs of targets is necessary to obtain the cost of their connection. To enable TSP finding a low-cost solutions, it is furthermore desired that this target-to-target path planning provides good quality paths.

In theory, paths between all pairs of targets should be computed to define the distance matrix for TSP, but it is time-consuming. Practically, paths between near targets are likely to be used in the final TSP sequence due to their low cost, while paths between distant targets may be ignored. For example in Fig. 1a, it is more useful to find paths between targets A, B, C and D as they are close to each other rather than finding paths between D and E as the final TSP solution (Fig. 1d) would rather connect the near targets. This observation motivates the search proposed in this planner: instead of finding paths between all pairs of targets (to define costs of visiting the targets for TSP), we propose to find good-quality paths only between near targets.

In this paper, we propose a novel tree-based randomized

| (a) Goals | (b) Partially grown trees | (c) Connected trees | (d) Final path |

Fig. 1: Example of multi-goal path planning with target locations (black) (a). We first build a set of trees from each goal (b) and expand them until they touch obstacles or each other (b). Paths between the targets are found by searching the connected trees (c). Finally, TSP computes final sequence of goals (d). Visualization of SFF* is available at https://youtu.be/vBQVO_GP5Sc

planner for finding trajectories between multiple targets concurrently (Fig. 1a). The planner, called Space-Filling Forest (SFF*), grows multiple trees simultaneously, starting from the given targets (Fig. 1b). The trees are expanded in the configuration space in a randomized manner until they approach each other or get close to an obstacle. The nodes of two different trees are connected if they are close to each other. This forms a roadmap where the path connecting the targets can be found (Fig. 1c). The paths are then used in the subsequent TSP computation to obtain the final sequence of visiting the targets and minimizing the overall cost (Fig. 1d). SFF* will be released as an open-source[1].

II. RELATED WORK

Multi-goal path planning requires, besides constructing the final trajectory, also finding the order of the goals (targets) to be visited [27]. In the case of one vehicle, the problem can be considered as an instance of TSP. Solving the combinatorial part, i.e., TSP, requires to define costs between the individual targets. Dubins-TSP is a variant of TSP, where the robots are considered as Dubins vehicles moving in an obstacle-free environment [21], [25]. In Dubins-TSP, the trajectory between targets can be found analytically in a short time.

However, the approach cannot be used in the case of robots moving among obstacles, where more general planners need to be used to obtain trajectories avoiding the obstacles. This is formulated in the Physical TSP problem [23] and used, e.g., for the mine countermeasure missions [19]. Generally, finding paths among obstacles for robots of arbitrary shape can be solved using sampling-based planners like Rapidly-exploring Random Tree (RRT) [16] and Probabilistic Roadmaps (PRM) [12]. A number of planners were derived from basic RRT and PRM, e.g., their asymptotically optimal variants RRT* and PRM* [11]. We refer to the survey [6] about many other variants of these planners.

A widely used approach to utilize sampling-based motion planning in multi-goal path planning is to derive target-to-target paths between all pairs of targets. The final sequence of visiting the waypoints is achieved using TSP [27], [26], [7]. Alternatively, other TSP-related formulations like Watchman Routing Problem [4] or Vehicle Routing Problem [24] can be used in special cases. The work [20] uses a finite automaton to determine the sequence.

Finding all target-to-target trajectories is computationally demanding due to its $O(n^2)$ complexity for n targets. For scenarios with tens or a few hundreds of targets, the runtime of the TSP solution is minor compared to the runtime of all target-to-target path planning [26] and therefore, the speed of path planning is crucial.

Lazy-TSP [7] reduces the load of path planning. For a set of targets in the environment with obstacles, the costs of their connection is simply defined by their Euclidean distance, i.e., without considering the obstacles. An initial tour is computed using TSP and the connection between the consecutive targets is verified by a time-consuming RRT-based planner. If the connection cannot be found, the TSP is iteratively refined

until a valid sequence and a corresponding trajectory is found. The number of paths computed by the RRT-based planner is therefore reduced, which also decreases the computational time. The approach is faster than finding all target-to-target connections using the PRM approach [4]. Additional speedup can be achieved using online learning methods to predict collisions of edges [13].

The speed-up of the multi-goal planning can also be achieved by improving the underlying sampling-based planner, e.g., by using multiple RRT trees. Bidirectional-RRT alternately grows two trees rooted at the start and goal, respectively, towards the random samples [15]. One global tree (rooted at the initial configuration) and several local trees are grown in [28]. The random sample is first tested for connection to the global tree, and if it fails, all local trees attempt to connect to the sample. If none of the trees can connect to the sample, a new tree is set up at the sample with a given probability. A similar approach is presented in [3], but in contrast to [28], the new tree is always set up at a random sample if the sample is not reachable from any other tree. Instead of creating new trees everywhere (as in [28], [3]), the method [30] establishes a new tree only if the random sample is estimated to be in a narrow passage.

Multiple RRT trees for multi-goal path planning was introduced in [5]. The trees are rooted at the targets, and they are selected for expansion using a round-robin, i.e., in each iteration, only a single tree expands toward the random sample. If two trees approach each other close enough, they are connected (if possible), and the trajectory between the roots of the trees is retrieved. Then, the connected trees are merged, so they continue to grow as a single tree. Due to the merging of the connected trees, the algorithm can provide at most one path between each pair of waypoints and no alternative path can be found even if the number of samples increases. Moreover, the trajectory between each pair is not optimal due to the non-optimality of the underlying planner [5].

In our previous work [29], we proposed to grow multiple random trees and connect them at any two nodes being close enough. In contrary to [5], where the trees are merged into a single one after they approach each other, our approach [29] considers the connection as 'virtual', i.e., the trees remain separated and grow further independently. This results in a roadmap of trees that are connected by multiple virtual edges, so more than one path can be found between two targets.

In this paper, we further extend our method [29] for multi-goal path planning in environments with obstacles; the planner proposed in this paper is referred to as Space-Filling Forest (SFF*). In comparison to [29], we improve the quality of the paths by using the rewiring technique while growing the trees. The growth of the trees towards other targets is boosted via a priority queue to bias finding connections between near trees. For each tree, a list of priority queues (for every target location) is maintained; the priority queues define the priority of nodes for the expansion. Therefore, nodes that are believed to be close to a target location are more likely expanded. Priority queue provides an efficient way to prioritize expansion towards promising areas and it has been already used in sampling-based motion planning [10].

---

[1] At https://github.com/ctu-mrs/space_filling_forest_star

III. Problem Formulation

The multi-goal path planning problem being solved in this paper focuses on finding collision-free minimal-cost path over multiple target locations. Such a problem includes two challenging parts. The first one is the finding of collision-free paths with a minimal cost between all target locations. The second one contains a combinatorial optimization problem of TSP that finds the appropriate sequence to visit the targets to minimize the overall path cost using the already found collision-free target-to-target paths. This paper proposes the SFF* method for the first part, while the second part is solved by an existing state-of-the-art algorithm. In the rest of this section we summarize the problem and used notation.

Let $\mathcal{C}$ denote the configuration space and let $\mathcal{C}_{free} \subseteq \mathcal{C}$ is the collision-free region of the configuration space where the robot can move. The distance between two configurations is denoted as $\varrho(a,b), a,b \in \mathcal{C}$.

Multiple target locations are specified $R = \{r_1, \ldots, r_n\}$, $r_i \in \mathcal{C}_{free}$ and need to be visited by the robot. The sequence to visit the target locations can be described by a vector of their indexes $\Sigma = (\sigma_1, \ldots, \sigma_n)$, $1 \leq \sigma_i \leq n$, $\sigma_i \neq \sigma_j$ for $i \neq j$. The combinatorial TSP optimization part, therefore, finds the appropriate Σ.

However, the collision-free paths connecting the targets in R together with their costs have to be known before finding Σ. A path between targets r_i and r_j can be described as $\tau_{ij} : [0,1] \rightarrow \mathcal{C}_{free}$ with $\tau_{ij}(0) = r_i$ and $\tau_{ij}(1) = r_j$. Its cost is denoted $\varrho(r_i, r_j) = \varrho(\tau_{ij}) = \int_0^1 |\tau_{ij}(t)| dt$. Additionally, the shortest possible path τ_{ij}^* is required such that $\varrho(\tau_{ij}^*) = \min\{\varrho(\tau_{ij}) | \tau_{ij} \in \mathcal{C}_{free}\}$. Therefore, the necessary task is to find set $T^* = \{\tau_{ij}^* | i = 1, \ldots, n, j = 1, \ldots, n\}$ of all collision-free paths with minimal costs between the targets.

Both parts of the multi-goal path planning problem using the TSP formulation can be specified as a single optimization problem

$$\underset{\Sigma, T^*}{minimize} \sum_{i=2}^{n} \varrho(r_{\sigma_{i-1}}, r_{\sigma_i}) + \varrho(r_{\sigma_n}, r_{\sigma_1}),$$
$$\text{s.t. } \varrho(r_i, r_j) = \varrho(\tau_{ij}^*), i = 1, \ldots, n, j = 1, \ldots, n, \quad (1)$$
$$\sigma_i \in \Sigma, i = 1, \ldots, n.$$

Notice that the minimality of τ_{ij} is particularly important for the paths used by Σ, which motivates the proposed SFF* to minimize mainly the path between neighboring targets as they are more often used in a TSP solution.

IV. Proposed Method

The proposed SFF* is a sampling-based method that aims to find short paths between multiple target locations. The method has three key features. First, it grows multiple trees simultaneously starting from the target locations. If two trees approach each other, their connection is tested. The feasible connections are considered as 'virtual' edges and stored separately. Later, they are used to find paths between the connected trees. Second, the nodes for expansion are explicitly defined in an open and close lists. New nodes are added to the open list, while nodes that were difficult to expand are moved into the

---

Algorithm 1: SFF

Input: root nodes $R = \{r_1, \ldots, r_n\}$, priority queue bias p_q;
Output: path between each target $r_i, r_j \in R$

1   $T = \{t_1, \ldots, t_n\}$ initialize the trees at targets $r_1, \ldots, r_n$;
2   $O = \{r_1, \ldots, r_n\}$;   // open list
3   $C = \emptyset$;   // closed list
4   $E = \emptyset$;   // virtual edges between trees
5   initialize $Q_{i,j}, i, j = 1, \ldots, n; i \neq j$;
6   **for** $iteration = 1, \ldots, I_{max}$ **do**
7   **if** $O \neq \emptyset$ **then**
8   **if** $rand(0,1) < p_q$ **then**
9   i = index of a random tree, $i \in 1, \ldots, n$;
10   q = random queue, $q \in Q_{i,j}, j = 1, \ldots, n$;
11   e = best node from q;
12   **else**
13   e, i = random node $e \in O$, index of its tree t_i;
14   **else**
15   e, i = random node $e \in C$, index of its tree t_i;
16   **if** $ExpandNode(e,i)$=*failed* **and** $e \notin C$ **then**
      // Alg. 2
17   remove e from open list O;
18   remove e from all priority queues $Q_{i,j}, j = 1, \ldots, n$;
19   **if** $O = \emptyset$ **and** $(T + E)$ *forms a single component* **then**
20   **break** ;   // we can find paths between each r_i and r_j

---

close list. The algorithm primarily selects nodes for expansion from the open list, as these nodes are believed to be easily expandable. However, the method can also draw nodes for expansion from the close list. A node is expanded in a random direction, but expansion towards other nodes of the same tree are prohibited. This boosts the growth towards unexplored areas and prevents each tree to grow towards itself. Due to the usage of open/close lists, the trees are not expanded using the Voronoi-bias [17] (as in the case of RRT and its derivatives), yet, the trees can spread in $\mathcal{C}_{free}$. Third, a priority queue is used to boost the growth of each tree towards near targets. Furthermore, the rewiring technique known from RRT* [11] is used to minimize costs of the nodes.

The method is summarized in Alg. 1. First, n trees $T = \{t_1, \ldots, t_n\}$ are created and rooted in the target locations R and the roots of the trees are added to the open list O. In each iteration, the algorithm selects a random node (and its corresponding tree) for expansion and attempts to expand it by randomly searching its vicinity for new collision-free nodes. If the expansion fails, the selected node is considered as difficult to expand and it is moved to the close list C (lines 17–18 in Alg. 1). The algorithm terminates after a predefined number of iterations I_{max} or when the open list is empty and all target locations have been connected to a single component, i.e., when a path between each pair of targets can be found using all created trees and considering their connection.

The key part of SFF* is the selection of the nodes for expansion. Primarily, the nodes from open list are selected either randomly or by considering their distance towards roots of other trees. The latter case is introduced to boost the expansion towards target locations that are believed to be easily reachable. This boosting is achieved via priority queues $Q_{i,j}, i, j = 1, \ldots, n, i \neq j$, where i denotes the index of the tree and j denotes the index of a target. For each tree, $n - 1$ queues are maintained. The queue $Q_{i,j}$ is an ordered list of

(a) Sampling around e in distance l

(b) Expansion of e by r_{new}

(c) Expansion of e by r_{new} + connect. t_1 and t_3 via r_{new}

Fig. 2: Example of SFF* expansion for trees t_1, t_2, t_3 rooted at target locations (red). The nodes in the open list are in green, the nodes in the close list are in black. Let assume the node e is going to be expanded, so its vicinity is sampled in distance l from e (brown) (a). The candidate 1 is discarded as it approaches other nodes of the tree closer than e. The candidate 4 is discarded as it is not collision-free (b). If the candidate $r_{new} = 2$ is added to the tree, it becomes member of the open list and no other action is made (b). However, when candidate $r_{new} = 3$ is added to the tree, it is already too close to the node $f \in t_3$ because $\varrho(r_{new}, f) < d$. Therefore, the trees t_1 and t_2 are virtually connected (blue edge) via new node r_{new}, i.e., edge (r_{new}, f) is added to the list of connections E (c).

nodes of the tree i according to their distance towards the target r_j, i.e., according to $\varrho(e, r_j), e \in t_i$. The binary heap data structure can be used to efficiently implement the queues.

The node for expansion is selected as follows. If the open list is not empty, a node from the open list is selected randomly with the probability $1 - p_q$. Otherwise, with the probability p_q, a tree is selected randomly, and then one of its queues is selected randomly as well. From this queue, the node with the shortest distance to a particular target is selected for expansion (lines 8–11 in Alg. 1). If the open list is empty, the node for the expansion is selected randomly only from the close list.

The task of the expansion (Alg. 2) is to find a new collision-free configuration in the vicinity of the node being expanded such that the tree grows toward other trees and does not grow toward itself. We create up to k samples in the distance l from e (node for expansion), and discard the colliding ones. Samples that approach the same tree to the distance less than the distance to the node e, are also discarded, as they would cause growing the tree toward itself. If the expansion of a node fails, i.e., its vicinity cannot be sampled, it is removed from open list and moved to the close list.

If a new sample r_{new} is found, it is added to the open list, to the tree and all its queues. The expansion process is illustrated in Fig. 2. After new node r_{new} is added to the tree, the rewiring at r_{new} according to RRT* rules [11] are used (lines 12–17 in Alg. 2), where $cost(e)$ denotes the length of the path from the root of the tree to the node e. The task of the rewiring is to minimize the cost of reaching the tree nodes, and consequently the cost between multiple targets when the trees are connected.

After the new node r_{new} is added to the tree t_i, the connection with other trees is checked. The connection is possible if the distance $\varrho_{tree}(r_{new}, t_j) < d$ for some tree $t_j, j \neq i$ and if this connection is collision free (lines 18–21 in Alg. 2), where $\varrho_{tree}(q, t), q \in \mathcal{C}, t \in T$ is the distance between a node and its nearest node in the tree t. In such a case, the connecting virtual edge is remembered in the set E and later used to decide if all trees form a single component or

Algorithm 2: ExpandNode

Input: index i of tree t_i for expansion, node for expansion $e \in t_i$;

Global params.: sampling distance l, distance of two trees to be connected d, open list O, list of virtual edges E, all trees T ;

Output: failure or success

1 $r_{new} = \emptyset$;
2 **for** $1, \ldots, k$ **do**   // expansion around e
3      $r' =$ sample random node in the distance l from e;
4   **if** *canConnect(e, r')* **and** $\varrho_{tree}(t_i, r') > \varrho(r', e)$ **then**
5         $r_{new} = r'$;
6   **break**;

7 **if** $r_{new} = \emptyset$ **then**
8   **return** failure ;   // expansion failed

9 add r_{new} to tree t_i;
10 add r_{new} to open list O;
11 add r_{new} to all priority queues $Q_{i,j}, j = 1, \ldots, n$;
12 $X_{near} = k$-nearest neighbors of r_{new} using [11], sec. 3.3.3 ;
13 **foreach** $h \in X_{near}$ **do**   // see Alg. 6 in [11]
14   **if** *canConnect(r, h)* **and** $cost(r) > cost(h) + \varrho(r, h)$ **then**
15   set h as parent of r and update their costs
16   **if** *canConnect(r, h)* **and** $cost(h) > cost(r) + \varrho(r, h)$ **then**
17   set r as parent of h and update costs

18 **for** $t' \in T \setminus \{t_i\}$ **do**   // connection of trees
19     $f =$ find nearest node in tree t' towards r_{new} ;
20   **if** $\varrho(f, r_{new}) < d$ **and** *canConnect(f, r_{new})* **then**
21   add edge $(f, r_{new}), f \in t', r_{new} \in t_i$ to E;

22 **return** success;

not (line 19 in Alg. 1). This also allows evaluation of multiple connections between the same trees (each of them between different pairs of nodes), resulting in a different cost between targets.

A. Discussion

As SFF* connects two trees between multiple nodes, we can find more paths between the same pair of target locations and considering only the shortest of them for TSP. This feature brings an advantage over state-of-the-art planner [5], which can connect two trees at most once. The comparison between SFF* and method from [5] is illustrated in Fig. 3c,d.

Another advantage in comparison to [5] is that the growth of the trees in SFF* is not driven by Voronoi-bias, but it is maintained using the open/close lists. Voronoi-bias boosts the growth of the RRT-based trees towards unexplored areas of the configuration space, but it can be counteractive when connecting two near nodes located in large configuration space. In such a situation, trees of RRT-based methods likely grow to the open areas of the configuration space instead of approaching each other, which may result in finding long paths (Fig. 3a,b,c)

The most time-consuming operations in SFF* are collision detection (CD) and the nearest-neighbor search. The time complexity of one iteration of SFF* is $\mathcal{O}(k|t_i| \log |t_i| + kCD + REW(|t_i|) + n|t_j| \log |t_j| + nCD)$, where the first two terms represent the nearest-neighbor search and CD when expanding the tree t_i using k attempts (lines 2–6 in Alg. 2), where $|t_i|$ is the number of nodes in the tree t_i. We assume that the

Fig. 3: The difference between Multi-T-RRT [5] and SFF* in a map with the target locations r_1 and r_2. In classic RRT (and also in [5]), the trees are expanded towards the random samples due to Voronoi-bias. In the depicted scenario, more samples are generated on the left from targets than in the right (a). Consequently, the RRT-based method, e.g., [5], prefers to grow both trees towards the left zone. The connection of these trees more likely happens also in the left zone, which results in a long path between the targets (blue) (b). Moreover, as [5] connects the trees only once, no other path can be found even if the number of samples is increased. Contrary, SFF* can find multiple connections between the trees and therefore, it can also discover the shorter path (red) (c).

KD-tree is used for the nearest-neighbor search. $REW(t_i)$ is the complexity of the rewiring procedure that depends only on the size of the tree t_i. The last two terms are for testing the connections between the tree t_i and other trees, which also requires to compute the nearest-neighbor and perform collision detection (lines 18–21 in Alg. 2). Therefore, the number of targets n influences only the complexity of the connection of the trees.

SFF* explores the free region $\mathcal{C}_{free}$ by the multiple trees whose nodes cannot be closer than l. Therefore, the number of nodes required to cover $\mathcal{C}_{free}$ is given by the volume $vol(\mathcal{C}_{free})$ of $\mathcal{C}_{free}$. Let $\bar{t} \sim vol(\mathcal{C}_{free})/vol(node)$ denote the maximum number of nodes that are required to cover $\mathcal{C}_{free}$ assuming that each node has volume $vol(node)$. The nearest-neighbor search between $\bar{t}$ nodes has time complexity $\mathcal{O}(\bar{t} \log \bar{t})$.

Now let's assume that $\mathcal{C}_{free}$ is covered by n trees and each of them has $\bar{t}/n$ nodes. The time complexity of connecting one tree to the others is $\mathcal{O}(n(\bar{t}/n) \log(\bar{t}/n)) \leq \mathcal{O}(\bar{t} \log \bar{t})$. Therefore, the nearest-neighbor search for connecting the trees does not depend on the number of targets, but rather on the volume of $\mathcal{C}_{free}$. However, the expansion procedure attempts to connect actual tree t_i with all other n trees, which requires $n \cdot CD$ queries. Therefore, the overall time complexity of one expansion step can be expressed as $\mathcal{O}(k|t_i| \log |t_i| + kCD + REW(|t_i|) + \bar{t} \log \bar{t} + nCD)$, which increases linearly with the number of targets n due to collision detection, as verified experimentally in Fig. 4b.

The behavior of SFF* is controlled via several parameters. The parameter l specifies the size of the neighborhood where new samples are generated around a node being expanded. With increasing l, the tree grows faster (spreading more with few iterations), but it can miss the entrance to more difficult regions, e.g. to narrow passages. Therefore, the proper value depends mainly on the map, and we recommend to start with l according to the width of the expected narrow passages in the environments. The number of samples per expansion is determined by k. Low values speed up the expansion step, but the performance may be decreased in the narrow passages. We recommend to set up $k \approx 10$. Two trees are virtually

Fig. 4: Behavior of SFF* depending on the parameter p_q and with the increasing number of targets. Graphs are made using 20 measurements in V-Dense (a) and Dense (b) maps.

connected if they approach each other to the distance d. Larger d means that more connection tests will be performed, which increases the runtime as these connections necessarily rely on collision detection. We recommend to set d as a small multiple of l, e.g. as $d = 2 \cdot l$. With the higher probability of using the priority queues, p_q, SFF* prefers to connect near trees in early stages of the method, which also decreases the number of iterations required to find all target-to-target paths (Fig. 4a). Our experiments show that $p_q = 0.9$–0.95 yield the best results.

SFF* can also be used in other planning problems that require search in the configuration space. However, the design of SFF* assumes that several trees are grown simultaneously and that discovering multiple connections between the trees brings an advantage in the given task. Applying SFF* on a planning problem with a single start/goal is possible, but in such a case, the method reduces to RRT*, and the proposed features, e.g., priority heap, are not in use. SFF* can principally search high-dimensional configuration space. It only requires to employ a suitable nearest-neighbor data structure. For non-holonomic systems, a local planner allowing exact connection of two near configuration is required. These requirements are common also for other sampling-based planners.

V. RESULTS

The SFF* was compared with the Multi-T-RRT planner [5], Lazy-TSP [7], and with the Simple-SFF [29]. The contribution of the tree rewiring was further examined by using NR-SFF* version, which is SFF*, where rewiring is disabled (i.e., lines 12–17 in Alg. 2 are disabled). All methods were implemented in C++, and the experiments were performed on eight 16-core AMD Opteron 6376 2.3 GHz processors with 48 GB of RAM.

A. Performance in 2D workspace

In the first set of experiments, paths between 5, 10 and 20 targets has to be found for a holonomic hexagonal robot of radius 10 units. Three scenarios were considered: Dense, V-Dense and Triangles with size 2000×2000 units (Fig. 6).

On average, finding the multi-goal paths on problems with 20 targets took 0.58 s for Multi-T-RRT, 13.49 s for Simple-SFF, 13.83 s for NR-SFF*, and 16.19 s for SFF*. Lazy-TSP is considerable slower, which is caused by the RRT* planner that is internally used in Lazy-TSP to evaluate target-to-target distances. On the problems with 5 and 10 targets, Lazy-TSP

took in average 18 s, but > 800 s was required to solve problems with 20 targets.

The result of each planner is a roadmap in which target-to-target paths were found using Dijkstra's algorithm, and costs of these paths were used in TSP. TSP was calculated using the state-of-the-art TSP Concorde solver [1]. The runtimes of TSP are, in our scenarios with few tens of nodes, negligible compared to the runtime of the planners. Namely, TSP with 5 targets is solved in ~ 20 ms, ~ 26 ms for 10 targets, and ~ 125 ms for 20 targets.

From the runtime point of view, Multi-T-RRT outperforms the proposed SFF*. However, as will be demonstrated in the following text, the proposed SFF* (and also Simple-SFF and NR-SFF*) provides significantly better paths than Multi-T-RRT. A practitioner may want to know whether running Multi-T-RRT repeatedly and using the best solution (i.e., a solution with the shortest paths between the targets) can yield similar results as SFF*, but with a shorter computational time. We investigated this alternative by considering another method called Multi-T-RRT-20, which represents the best results out of 20 trials of Multi-T-RRT. We decided to use 20 trials as a single run of Multi-T-RRT is approximately 28 times faster than SFF*, therefore running 20 times Multi-T-RRT would still result in a faster planner than SFF*.

SFF* and NR-SFF* was run with $p_q = 0.95$, $d = 50$ units and $l = 40$ units and $k = 12$. Collision detection was called $3(n + k)$ times per iteration of SFF*, because each iteration results in $n + k$ edges that are tested for collisions at three points (start/middle/end). Multi-T-RRT was run in a similar manner: the expansion step was $l = 40$ units and the distance for connecting two trees was $d = 50$ units. All methods were terminated after $I_{max} = 100 \times 10^3$ iterations.

B. Target-to-target distance comparison

We first compare the costs of the paths between the targets as follows. After each algorithm finishes, we find all target-to-target paths. The cumulative cost of these paths is then used for the comparison. For each map and the number of target locations, each algorithm was run 10^4 times.

The results are depicted in Fig. 5 in the form of a histogram of the cumulative costs (in units). In all tested cases, SFF* outperformed other methods because it provides smaller cumulative costs, i.e., shorter paths, than other methods. The planner Multi-T-RRT provides the highest cumulative costs. Repeated runs of Multi-T-RRT with selecting the best solution out of 20 trials (algorithm Multi-T-RRT-20) decreases the cumulative costs, but not significantly, so even Multi-T-RRT-20 is outperformed by SFF*.

Multi-T-RRT is also outperformed by the Simple-SFF (which uses no rewiring and no priority heap but connects the trees multiple times). This indicates that connecting the trees multiple times already improves the quality of the solutions. Contrary, Multi-T-RRT connects the trees only once, which results in only one path connecting each target location. Due to the stochastic behavior of Multi-T-RRT, this connection varies between different runs, which also results in a higher deviation of the cumulative costs. Contrary, the deviation of SFF*,

(a) Dense, 10 targets (b) Dense, 20 targets

(c) Triangles, 10 targets (d) Triangles, 20 targets

(d) V-Dense, 10 targets (f) V-Dense, 20 targets

Fig. 5: Histograms of cumulative costs of target-to-target paths, the cost path (horizontal axis) is the map units.

Simple SFF and NR-SFF* is smaller. The comparison between SFF and NR-SFF* shows the positive effect of the rewiring procedure: SFF* provides better results (lower values of the cumulative costs) than NR-SFF*. Lazy-TSP is not included in Fig. 5 as it does not compute all target-to-target paths.

C. Multi-goal paths

Example of the final multi-goal tours are depicted in Fig. 6. The progress of SFF* trees is depicted in Fig. 1. The results are summarized in Tab. I, where the column 'TSP' is the total length of the final paths (in units), the column 'Iterations' denotes how many sampling iterations were needed to connect all targets to a single component. The shortest paths are provided by SFF* and Lazy-TSP. The cases where Lazy-TSP and SFF* provided statistically same results (using t-test with $\alpha = 0.05$) are marked '=' in the table. The biggest advantage of the proposed SFF* over Lazy-TSP is the number of iterations required to find the solution: Lazy-TSP requires 5–10$\times$ more iterations than the SFF*. Consequently, also the runtime of Lazy-TSP is significantly higher than the runtime of SFF*. This enables SFF* to be run repeatedly and using the best result with the minimum TSP cost. In this case, the best result provided by SFF* is better than the best result provided by Lazy-TSP (column 'TSP best' in Tab. I).

The cost of TSP solutions obtained using paths from Multi-T-RRT is from 45.26 % to 75.23 % higher than the costs of solutions achieved using SFF*. When repeated runs of Multi-T-RRT are used, the cost of TSP solutions is from 32.54 % to 63.48 % higher than the cost of TSP solutions obtained using SFF*. This indicates that running a planner providing low-quality paths repeatedly, albeit it is faster, does not necessarily improve the cost of the final solution. We have also examined

TABLE I: Comparison of the planners using costs of TSP solutions. The columns 'TSP' and 'Iterations' are in format mean | std. dev. The results are based on 100 measurements. 'TSP best' is the smallest achieved TSP cost. TSP costs are computed using Concorde [1] solver. Computing TSP using LKH [9] led to same results in all cases except ones denoted [a], where the costs of LKH solution is by 0.21 % smaller than the solution of Concorde; and in case [b], where the cost of LKH solution is by 0.33 % smaller than the solution of Concorde. The marks $\neq$ and $=$ denote if the cost of SFF* solution is different, or same, as solution of Lazy-TSP using t-test and $\alpha = 0.05$.

Num. of goals:	5			10			20		
	TSP $\times 10^3$	TSP best $\times 10^3$	Iterations $\times 10^3$	TSP $\times 10^3$	TSP best $\times 10^3$	Iterations $\times 10^3$	TSP $\times 10^3$	TSP best $\times 10^3$	Iterations $\times 10^3$
Dense									
SFF*	**7.00** | **0.53**	*6.46*	5.60 | 0.32	12.63 | 1.34[a]	*9.99*	10.72 | 10.74	15.18 | 1.23 [a]	*12.25*	12.25 | 12.53
NR-SFF*	9.09 | 0.61	*7.60*	6.21 | 0.36	15.04 | 1.50[a]	*11.82*	10.93 | 10.30	17.44 | 1.33 [a]	*13.87*	12.29 | 11.97
Simple-SFF	9.33 | 0.74	*7.64*	6.01 | 0.35	15.00 | 1.45[a]	*11.87*	11.40 | 10.94	17.60 | 1.40[b]	*14.30*	13.76 | 13.49
Multi-T-RRT	10.85 | 0.77	*8.93*	0.47 | 0.14	20.03 | 1.38	*15.65*	2.10 | 0.60	25.33 | 1.38	*21.46*	2.29 | 0.61
Multi-T-RRT-20	9.66 | 0.23	*8.85*	0.40 | 0.11	17.76 | 0.54	*15.59*	1.91 | 0.57	23.08 | 0.52	*20.81*	2.20 | 0.59
Lazy TSP	7.18 | 0.40 $\neq$	*6.59*	20.99 | 8.08	**10.30** | **0.50** $\neq$	*9.34*	84.53 | 28.47	**12.73** | **0.39** $\neq$	*11.95*	125.94 | 32.99
Triangles									
SFF*	1.82 | 0.19	*1.59*	2.84 | 2.99	2.42 | 0.18	**2.07**	2.61 | 1.28	3.22 | 0.20	**2.65**	2.45 | 1.25
NR-SFF*	2.25 | 0.22	*1.74*	2.97 | 2.13	2.74 | 0.19	*2.24*	2.76 | 1.95	3.36 | 0.21	*2.85*	2.53 | 1.44
Simple-SFF	2.23 | 0.23	*1.73*	2.90 | 2.41	2.75 | 0.22	*2.23*	2.69 | 1.03	3.42 | 0.23	*2.85*	2.54 | 1.48
Multi-T-RRT	2.78 | 0.21	*2.18*	0.10 | 0.03	4.07 | 0.24	*3.38*	0.13 | 0.03	5.42 | 0.24	*4.65*	0.14 | 0.03
Multi-T-RRT-20	2.42 | 0.08	*2.15*	0.09 | 0.02	3.66 | 0.10	*3.31*	0.12 | 0.03	5.00 | 0.11	*4.48*	0.13 | 0.02
Lazy TSP	**1.81** | **0.09** $=$	*1.64*	1.74 | 0.79	**2.35** | **0.10** $\neq$	*2.11*	3.52 | 7.12	**3.00** | **0.10** $\neq$	*2.75*	3.29 | 0.83
V-Dense									
SFF*	8.55 | 0.80	*7.61*	5.49 | 0.60	**9.17** | **0.50**	*8.46*	5.58 | 0.51	12.35 | 0.56	**11.13**	5.67 | 0.99
NR-SFF*	11.37 | 0.93	*9.33*	6.10 | 0.65	11.37 | 0.58	*9.92*	6.09 | 0.81	14.61 | 0.64	*12.63*	6.10 | 1.23
Simple-SFF	11.00 | 0.94	*9.00*	5.89 | 1.03	11.39 | 0.65	*9.85*	5.94 | 1.09	14.51 | 0.71	*12.67*	6.00 | 1.33
Multi-T-RRT	12.42 | 0.70	*10.67*	0.45 | 0.21	15.99 | 0.89	*13.51*	0.48 | 0.18	21.64 | 0.84	*19.14*	0.49 | 0.15
Multi-T-RRT-20	11.36 | 0.21	*10.41*	0.36 | 0.16	14.58 | 0.31	*13.26*	0.44 | 0.16	20.19 | 0.36	*18.42*	0.44 | 0.13
Lazy TSP	**8.49** | **0.29** $=$	*7.81*	14.82 | 5.76	9.50 | 0.38 $\neq$	*8.81*	34.12 | 9.62	**12.28** | **0.36** $=$	*11.30*	51.39 | 12.62

(a) Dense, SFF*

(b) Dense, NR-SFF*

(c) V-Dense, SFF*

(d) Triangles, SFF*

Fig. 6: Final TSP solution on the V-Dense map (a) and on the Triangles map (b).

(a) Dense 3D   (b) Building   (c) Triangles 3D

Fig. 7: 3D workspace with partially grown SFF* trees. Spheres denote the target locations.

the influence of TSP solver. Besides Concorde, that was used to compute TSP solutions in Tab. I, also LKH [9] was used. In our scenarios with few tens of targets, Concorde and LKH provided the same results, except few cases that are marked in Tab. I.

D. Performance in 3D workspace

Multi-goal path planning in 6D configuration space (3D robot in 3D workspace) was tested using three scenarios (Fig. 7) and compared to Multi-T-RRT. The size of the workspace is $100 \times 100 \times 100$ units and robot was 10 units long with radius 3 units. In average, SFF* takes 10 minutes on the scenarios with 20 targets and Multi-T-RRT takes 5 minutes. The speed of Lazy-TSP was too high (> 50 minutes per trial for more than 10 and 20 targets), and therefore, Lazy-TSP was

not tested in this case. The results are summarized in Tab. II. SFF* utilized all allowed iterations ($I_{max} = 100 \times 10^3$) but still found multiple connections between the trees. The reason for so many iterations is the growth of the trees outwards from the obstacles (towards the open space), not the inability to connect individual trees. This is confirmed by the quality of TSP paths, which is lower for SFF* and higher for Multi-T-RRT.

VI. CONCLUSION

This paper focused on multi-goal path planning, where the task is to visit several target locations in an environment with obstacles. The order of visiting the targets is obtained by solving the related Traveling Salesman Problem. The core of the paper is the novel path planner called Space-Filling Forest (SFF*) that finds high-quality paths between the individual targets. SFF* builds multiple trees in the configuration space starting from the target locations. The trees are grown in the free space until they approach each other or their growth is locally stopped by an obstacle. Multiple connections are found between the trees that approached each other to a predefined distance. Paths between the targets are found in the connected trees. To find good-quality paths, the trees are grown in the RRT* manner, i.e., with rewiring the nodes during the

TABLE II: Comparison of the planners using costs of TSP solutions. The columns 'TSP' and 'Iterations' are in format mean | std. dev. The results are based on 100 measurements. 'TSP best' is the smallest achieved TSP cost. TSP costs are computed using Concorde [1] solver and the same TSP costs were obtained also using LKH solver [9].

Num. of goals:	5			10			20		
	TSP $\times 10^3$	TSP best $\times 10^3$	Iterations $\times 10^3$	TSP $\times 10^3$	TSP best $\times 10^3$	Iterations $\times 10^3$	TSP $\times 10^3$	TSP best $\times 10^3$	Iterations $\times 10^3$
Dense 3D									
Multi-T-RRT	14.99 | 1.81	*11.05*	1.44 | 0.52	29.20 | 2.35	*23.80*	2.15 | 0.53	46.58 | 2.96	*39.82*	2.45 | 0.50
SFF*	**6.79** | **0.14**	*6.50*	34.05 | 0.39	**10.63** | **0.50**	*9.96*	32.77 | 0.33	**14.29** | **0.49**	*13.48*	32.64 | 2.09
Building									
Multi-T-RRT	0.16 | 0.01	*0.12*	27.74 | 13.88	0.27 | 0.02	*0.22*	18.28 | 10.00	0.40 | 0.02	*0.34*	19.27 | 9.64
SFF*	**0.06** | **0.00**	*0.06*	100.00 | 0.00	**0.09** | **0.00**	*0.08*	100.00 | 0.00	**0.13** | **0.00**	*0.12*	100.00 | 0.00
Triangles 3D									
Multi-T-RRT	0.11 | 0.01	*0.08*	3.82 | 1.21	0.20 | 0.02	*0.16*	5.07 | 1.31	0.31 | 0.02	*0.25*	6.04 | 1.33
SFF*	**0.04** | **0.00**	*0.04*	100.00 | 0.00	**0.06** | **0.00**	*0.06*	100.00 | 0.00	**0.09** | **0.00**	*0.08*	100.00 | 0.00

growth. The experiments have shown superior performance in comparison to state-of-the-art methods.

REFERENCES

[1] Concorde TSP Solver. http://www.math.uwaterloo.ca/tsp/concorde.html Accessed 2020-10-09.

[2] G. Best, J. Faigl, and R. Fitch. Multi-robot path planning for budgeted active perception with self-organising maps. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2016.

[3] M. Clifton, G. Paul, N. Kwok, D. Liu, and D. Wang. Evaluating performance of multiple RRTs. In *IEEE/ASME International Conference on Mechtronic and Embedded Systems and Applications*, pages 564–569, 2008.

[4] T. Danner and L. E. Kavraki. Randomized planning for short inspection paths. In *IEEE International Conference on Robotics and Automation*, volume 2, pages 971–976 vol.2, 2000.

[5] D. Devaurs, T. Siméon, and J. Cortés. A multi-tree extension of the transition-based RRT: Application to ordering-and-pathfinding problems in continuous cost spaces. In *IEEE/RSJ IROS*, 2014.

[6] M. Elbanhawi and M. Simic. Sampling-based robot motion planning: A review. *IEEE Access*, 2:56–77, 2014.

[7] B. Englot and F. S. Hover. Three-dimensional coverage planning for an underwater inspection robot. *The International Journal of Robotics Research*, 32(9-10):1048–1073, 2013.

[8] J. Faigl and G. A. Hollinger. Unifying multi-goal path planning for autonomous data collection. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2937–2942, 2014.

[9] Keld Helsgaun. An extension of the lin-kernighan-helsgaun tsp solver for constrained traveling salesman and vehicle routing problems. *Roskilde: Roskilde University*, 2017.

[10] J. D. Hernández, M. Moll, and L. E. Kavraki. Lazy evaluation of goal specifications guided by motion planning. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 944–950, 2019.

[11] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30(7):846–894, 2011.

[12] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12:566–580, 1996.

[13] C. Kew, B. A. Ichter, M. Bandari, E. Lee, and A. Faust. Neural collision clearance estimator for batched motion planning. In *The 14th International Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2020.

[14] G. Kizilateş and F. Nuriyeva. On the nearest neighbor algorithms for the traveling salesman problem. In Dhinaharan Nagamalai, Ashok Kumar, and Annamalai Annamalai, editors, *Advances in Computational Science, Engineering and Information Technology*, pages 111–118, Heidelberg, 2013. Springer International Publishing.

[15] J. J. Kuffner and S. M. LaValle. Rrt-connect: An efficient approach to single-query path planning. In *IEEE International Conference on Robotics and Automation*, volume 2, pages 995–1001. IEEE, 2000.

[16] S. M. LaValle. Rapidly-exploring random trees: A new tool for path planning, 1998. Technical report 98-11.

[17] S. R. Lindemann and S. M. LaValle. Incrementally reducing dispersion by increasing voronoi bias in RRTs. In *IEEE International Conference on Robotics and Automation*, volume 4, pages 3251–3257. IEEE, 2004.

[18] N. Mathew, S. L. Smith, and S. L. Waslander. A graph-based approach to multi-robot rendezvous for recharging in persistent tasks. In *2013 IEEE International Conference on Robotics and Automation*, pages 3497–3502, 2013.

[19] J. McMahon and E. Plaku. Autonomous underwater vehicle mine countermeasures mission planning via the physical traveling salesman problem. In *OCEANS 2015 - MTS/IEEE Washington*, pages 1–5, 2015.

[20] J. McMahon and E. Plaku. Mission and motion planning for autonomous underwater vehicles operating in spatially and temporally complex environments. *IEEE J. of Oceanic Engineering*, 41(4):893–912, 2016.

[21] J. Ny, E. Feron, and E. Frazzoli. On the dubins traveling salesman problem. *IEEE Transactions on Automatic Control*, 57(1):265–270, 2012.

[22] A. Otto, N. Agatz, J. Campbell, B. Golden, and E. Pesch. Optimization approaches for civil applications of unmanned aerial vehicles (uavs) or aerial drones: A survey. *Networks*, 72(4):411–458, 2018.

[23] D. Perez, P. Rohlfshagen, and S. M. Lucas. The physical travelling salesman problem: Wcci 2012 competition. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2012.

[24] R. Pěnička, J. Faigl, and M. Saska. Physical orienteering problem for unmanned aerial vehicle data collection planning in environments with obstacles. *IEEE Robotics and Automation Letters*, 4(3):3005–3012, 2019.

[25] R. Pěnička, J. Faigl, P. Váňa, and M. Saska. Dubins orienteering problem. *IEEE Robotics and Automation Letters*, 2(2):1210–1217, 2017.

[26] M. Saha, G. Sanchez-Ante, and J.-C. Latombe. Planning multi-goal tours for robot arms. In *IEEE International Conference on Robotics and Automation (ICRA)*, volume 3, pages 3797–3803, 2003.

[27] S. N. Spitz and A. A. G. Requicha. Multiple-goals path planning for coordinate measuring machines. In *IEEE International Conference on Robotics and Automation*, volume 3, pages 2322–2327 vol.3, 2000.

[28] M. Strandberg. Augmenting RRT-planners with local trees. In *IEEE International Conference on Robotics and Automation*, volume 4, pages 3258–3262 Vol.4, 2004.

[29] V. Vonásek and R. Pěnička. Space-filling forest for multi-goal path planning. In *24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1587–1590, 2019.

[30] W. Wang and Y. Li. A multi-RRTs framework for robot path planning in high-dimensional configuration space with narrow passages. In *International Conference on Mechatronics and Automation*, pages 4952–4957, 2009.