Informed Steiner Trees: Sampling and Pruning for Multi-Goal Path Finding in High Dimensions

Anonymous submission

Abstract

We interleave sampling based motion planning methods with pruning ideas from minimum spanning tree algorithms to develop a new approach for solving a Multi-Goal Path Finding (MGPF) problem in high dimensional spaces. The approach alternates between sampling points from selected regions in the search space and de-emphasizing regions that may not lead to good solutions for MGPF. Our approach provides an asymptotic, 2-approximation guarantee for MGPF. We also present extensive simulation results to illustrate the advantages of our proposed approach over prior works and a baseline using uniform sampling in terms of the quality of the solutions found and computation speed.

Introduction

Multi-Goal Path Finding (MGPF) problems aim to find a least-cost path for a robot to travel from an origin (s) to a destination (d) such that the path visits each node in a given set of goals ($\bar{T}$) at least once. In the process of finding a least-cost path, MGPF algorithms also find an optimal sequence in which the goals must be visited. When the search space is discrete ($i.e.$, a finite graph), the cost of traveling between any two nodes can be computed using an all-pairs shortest paths algorithm. In this case, the MGPF encodes a variant of the Steiner[a] Traveling Salesman Problem (TSP) and is NP-Hard (Kou, Markowsky, and Berman 1981). In the general case, the search space is continuous and the least cost to travel between any two nodes is not known a-priori. This least-cost path computation between any two nodes in the presence of obstacles, in itself, is one of the most widely studied problems in robot motion planning (Kavraki et al. 1996; Kuffner and LaValle 2000). We address the general case of MGPF as it naturally arises in active perception (Best, Faigl, and Fitch 2016; McMahon and Plaku 2015), surface inspection (Edelkamp, Secim, and Plaku 2017) and logistical applications (Janoš, Vonásek, and Pěnička 2021; Otto et al. 2018; Macharet and Campos 2018).

MGPF is notoriously hard as it combines the challenges in Steiner TSP and the least-cost path computations in the presence of obstacles; hence, we are interested in finding approximate solutions for MGPF. Irrespective of whether the search space is discrete or continuous, Steiner trees spanning the origin, goals and the destination play a critical role in the development of approximation algorithms for MGPF. In the discrete case, doubling the edges in a suitable Steiner tree, and finding a feasible path in the resulting Eulerian graph leads to 2-approximation algorithms for MGPF (Kou, Markowsky, and Berman 1981; Mehlhorn 1988; Chour, Rathinam, and Ravi 2021). This approach doesn't readily extend to the continuous case because we do not a-priori know the travel cost between any two nodes in $T := \{s, t\} \bigcup \bar{T}$. One can appeal to the well-known sampling-based methods (Karaman and Frazzoli 2011; Kavraki et al. 1996; Gammell, Srinivasa, and Barfoot 2014, 2015) to estimate the costs between the nodes, but the following key questions remain: 1) How to sample the space so that the costs of the edges joining the nodes in T can be estimated quickly so that we can get a desired Steiner tree? 2) Should we estimate the cost of all the edges or can we ignore some edges and focus our effort on edges we think will likely end up in the Steiner tree?

We call our approach *Informed Steiner Tree** (IST*). IST* iteratively alternates between sampling points in the search space and pruning edges. Throughout its execution, a Steiner tree is maintained which is initially empty but eventually spans the nodes in T, possibly including a subset of sampled points. IST* relies on two key ideas. *First*, finding a Steiner tree spanning T commonly involves finding a Minimum Spanning Tree (MST) in the metric completion[b] of the nodes in T. For any two distinct terminals u, v, we maintain a lower bound and an upper bound[c] on the cost of the edge (u, v). Using these bounds and cycle properties of an MST, we identify edges which will never be part of the MST. This allows us to sample regions corresponding to *only* those edges that can be part of the MST. We further bias our sampling by assigning a suitable probability distribution over the search space based on the bounds on the cost of the edges (See Fig. 1). *Second*, as the algorithm progresses, a new set of points are added to the search graph in each iteration. Each new sample added may facilitate a lower-cost feasible path between terminals requiring us to frequently

[a] Any node that is *not* required to be visited is referred to as a *Steiner node*. A path may choose to visit a Steiner node if it helps in either finding feasible solutions or reducing the cost of travel.

[b] The metric completion here is a complete weighted graph on all the nodes in T where the cost of an edge between a pair of nodes in T is the minimum cost of a path between them.

[c] Upper bound is the cost of a feasible path from u to v.

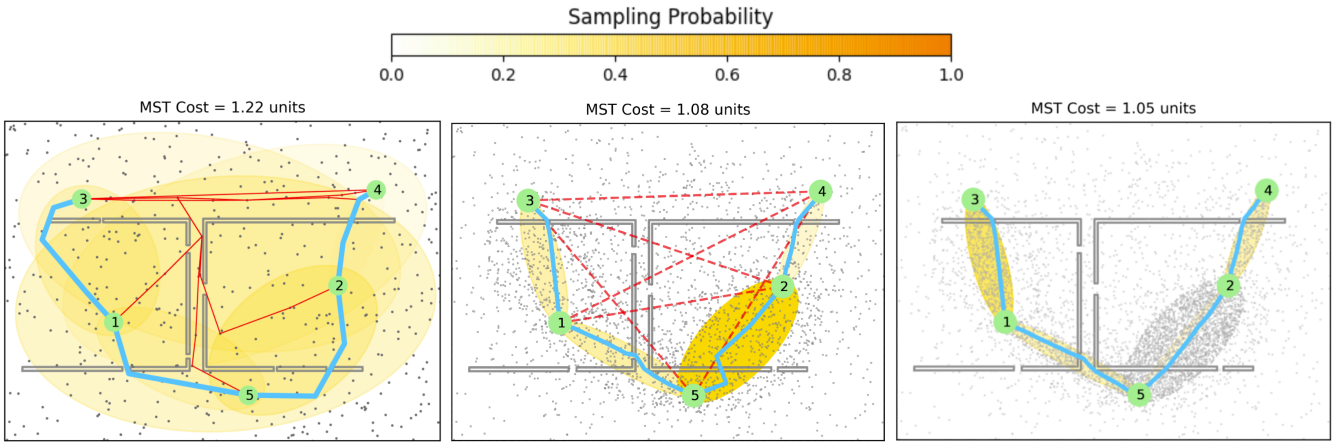


Figure 1: The Steiner tree (thick-blue lines) computed in three consecutive iterations of IST*, showing the advantage of pruning. The environment has 2 U-shaped obstacles with tiny openings. The numbered green circles denote the terminals in T that must be connected. A yellow-shade ellipse denotes an edge's informed set while its color intensity is the probability of that edge being sampled. Sampled points in free space are part of the roadmap (shown in grey). **Left**: The thin solid red lines are the actual paths corresponding to active non-MST edges under consideration (their edge costs act as upper bounds on the corresponding optimal costs of the edges). Until any pruning, these edges are also sampled in addition to the edges in the MST. **Middle**: Dashed red lines show the edges that have been discarded (**pruned**) from further consideration as they will not contribute to the optimal MST. Consequently, their regions will be not be sampled any further. **Right**: After pruning, strategic densification of the roadmap happens (for example, the region around the edge between nodes 2 and 5) based on the probabilities at the end of the previous iteration. We observe that the roadmap is further densified around active edges leading to an optimal MST faster.

update the Steiner tree. To address this efficiently, we develop an *incremental* version of the Steiner tree algorithm while maintaining its properties. Since this incremental approach correctly finds a Steiner tree, as the number of sampled points tends to infinity, IST* provides an asymptotic 2-approximation guarantee for MGPF.

We use the sampling procedure developed in *Informed RRT** (Gammell, Barfoot, and Srinivasa 2018) to choose points from selected regions in our approach. Informed sampling in synergy with pruning enables faster convergence to the optimal solution than uniform sampling. After describing IST* with its theoretical properties, we provide extensive computational results on high-dimensional problem instances.

Related Work

The MGPF and several variants of it have been addressed in the literature. Here, we discuss the most relevant work in continuous domains (detailed literature review present in the appendix). Un-supervised learning approaches[d](Faigl et al. 2011; Faigl 2016) using Self Organizing Maps (SOMs) have been used to solve MGPF. In (Faigl 2016), SOM is combined with a rapidly exploring random graph algorithm to find feasible solutions for the MGPF.

In (Devaurs, Siméon, and Cortés 2014), a meta-heuristic similar to *simulated annealing* is combined with multiple Rapidly-exploring Random Tree (RRT) expansions to solve MGPF. In (Englot and Hover 2013), LazyTSP was introduced for efficient path planning that combines RRT-based planning with TSP (Jünger, Reinelt, and Rinaldi 1995). It starts with TSP to compute an initial tour and then verifies the connections using RRT-based planning, iteratively refining the tour until a valid trajectory is found. In (Vonásek and Pěnička 2019), a method called *Space Filling Forests* (SFF) is proposed to solve MGPF. Multiple trees are grown from the nodes in T and unlike the approach in (Devaurs, Siméon, and Cortés 2014) where any two close enough (or neighboring) trees are merged into a single tree, in SFF, multiple virtual connections are allowed between two neighboring trees leading to multiple paths between any two nodes in T. Recently, in (Janoš, Vonásek, and Pěnička 2021), a generalization of SFF (called SFF*) has been proposed to also include rewiring of the edges in the trees similar to RRT*. Computational results show that SFF* performed the best among several previously known solvers for the simulation environments considered in (Janoš, Vonásek, and Pěnička 2021). We evaluated all the above methods with our proposed approach and report our findings in the results section.

A generalization of MGPF was considered in (Saha et al. 2006) where all the goals were partitioned into groups and the aim is to also visit one goal from each partition. They work under the assumption that a path between two nodes is computed at most once which may, in reality, be far from the optimal path. Their planner stops the instant it has found a feasible path connecting all the goals unlike our approach which keeps refining the paths.

[d]Since, the code for these learning approaches were not available, we could not directly test them on the problem instances considered in this paper.

Background and Preliminaries

Let $X \subseteq \mathbb{R}^n$ be the n-dimensional configuration space of a robot, and let $X_f \subset X$ be the set of obstacle-free configurations. Let $\sigma : [0,1] \rightarrow X_f$ be a collision free, feasible path such that the path starts at the origin ($\sigma(0) = s$), visits each of the goals in $\bar{T}$ at least once and ends at the destination ($\sigma(1) = d$). Let $c(\sigma) \in \mathbb{R}_{\geq 0}$ denote the cost of the path σ. The objective of MGPF is to find a collision-free feasible path σ such that $c(\sigma)$ is minimized.

Let $S \subseteq X_f$ be a set of sampled points (also referred to as nodes) from X_f. Let $V := S \bigcup T$. We refer to all the nodes in T as terminals[e]. Nodes which are within a radius of ρ of each other are treated as neighbors. Further, let $\mathcal{N}_V(u)$ denote the set of nodes which are neighbors to the node u. Edges are formed between neighbors if a feasible path is found between them using a local planner (for example, straight-line connection). Let $G = (V, E)$ denote the undirected graph thus formed where E denotes all the edges joining any pair of neighbors in V. Henceforth, we refer to G as the roadmap. Given a graph G, we use $E(G)$ to refer to all the edges present in G.

Let $e := (u, v)$ be an edge joining two distinct nodes u, v in G. Edge e represents a feasible path between u and v found by the local planner. The cost[f] of a feasible path joining vertices u and v is denoted as $dist(e)$. Let $h(e)$ be a lower bound on the length of any feasible path between u and v; typically, we set $h(e)$ to be the Euclidean distance between nodes u and v.

One approach for solving the MGPF is to sample as many points uniformly as possible from the obstacle-free space and form a roadmap spanning the terminals and the sampled points. At the end of the search process, the recently developed A^* based approach (Chour, Rathinam, and Ravi 2021) for discrete version of MGPF can be used to find a suitable Steiner tree. In this article, we consider this approach as the baseline against which IST* is compared.

Informed Steiner Trees

An overview of IST* is presented in Algorithm 1. At any iteration, IST* maintains three graphs to enable the search process:

1. *Roadmap graph:* $G = (V, E)$ includes all the terminals and the sampled points, and the edges between its neighboring nodes. Two nodes u, v are neighboring if $dist(u, v) \leq \rho$, where ρ is the connection radius. We provide a discussion on the role (and implementation) of ρ in the Appendix. Initially, G includes only the terminals and the edges between any pair of neighboring terminals.
2. *Shortest-path graph:* This graph is denoted as $\bar{G} := (V, \bar{E})$ and is a subgraph of the roadmap. It is a forest which contains a tree rooted at each terminal and maintains all the shortest paths from a terminal to each sampled point in its component. The shortest-path graph is used to find feasible paths between terminals. Initially, $\bar{G}$ is an empty forest ($i.e.$, contains no edges).
3. *Terminal graph:* This graph is denoted as $G_T = (T, \mathcal{A})$ and consists of only the terminals and any edges connecting them. Any edge in G_T corresponds to a path between two terminals in the roadmap. Therefore, any spanning tree S_T in the terminal graph corresponds to a feasible Steiner tree in G. For example, the leftmost subfigure in Fig. 1 shows the terminal graph. It is used in determining which regions to sample and in finding feasible Steiner trees for MGPF. $\mathcal{A}$ contains the set of all the edges that can possibly be part of S_T, and hence are currently actively explored when sampling. Initially, $\mathcal{A}$ consists of all the edges that join any pair of terminals in T. As the algorithm progresses, edges in $\mathcal{A}$ may be pruned. Also, the cost of an edge connecting any pair of distinct terminals u, v in G_T is denoted as $cost_T(u, v)$. Initially, $cost_T(u, v)$ is set to ∞ for all $u, v \in T$ and S_T is empty.

Algorithm 1: IST*

Input:
$X_f \subset X$ // Collision free space
$T \subset X_f$ // Discrete set of terminals
$h(e)$ for any $e = (u, v), u, v \in X_f$ // Heuristic cost between terminals u and v
n_s // Samples per batch
n_b // Number of batches
Output:
S_T // Steiner tree spanning the terminals in T
Initialization:
$V \leftarrow T$ // Add terminals to the roadmap
$\mathcal{N}_V(u) := \{v : dist(v, u) \leq \rho, v \neq u, v \in V\}$ for any $u \in V$ // nearest neighbors of u in V
$E \leftarrow \{(u, v) : v \in \mathcal{N}_V(u), u \in V\}$
$cost(e) = dist(u, v)$ for any $e = (u, v) \in E$
$G \leftarrow (V, E)$
$\mathcal{A} = \{(u, v) : u, v \in T, u \neq v\}$ // Set of active edges joining terminals
$cost_T(u, v) = \infty$ for any $(u, v) \in \mathcal{A}$
$Prob(e) = \frac{h(e)}{\sum_{e' \in \mathcal{A}} h(e')}$ for all $e \in \mathcal{A}$ // Initial probabilities using lower bounds
$S_T \leftarrow \emptyset$
// **Note:** $G, \bar{G}, cost, cost_T$ **are global variables.**
Main loop:
for $iteration = 1, \ldots, n_b$ **do**
  $S' \leftarrow AddSamples(Prob, \mathcal{A}, cost_T, n_s)$
  $Ripple(S')$
  $S_T \leftarrow UpdateTree(S_T, \mathcal{A})$
  $\mathcal{A} \leftarrow PruneEdges(S_T, \mathcal{A})$
  $Prob \leftarrow UpdateProbability(Prob, \mathcal{A}, S_T, h)$
return S_T

[e]This is a commonly used term in the optimization literature.

[f]We assume the costs are symmetric, $i.e.$, cost of traveling a given path from node u to node v is the same as the cost of traveling the path from v to u.

Algorithm 2: `AddSamples`$(Prob, \mathcal{A}, cost_T, n_s)$

```
1  S' ← ∅
2  for 1, ..., n_s do
3  |   Choose (u, v) ∈ A using Prob
4  |   x_rand ← Sample(u, v, cost_T(u, v))
5  |   S' ← S' ∪ {x_rand}
6  return S'
```

In each iteration of `IST*`, the algorithm (i) samples new points in X_f (line 21 of Algorithm 1), (ii) updates S_T based on a new incremental Steiner tree algorithm called `Ripple` (lines 22–23 of Algorithm 1), (iii) prunes edges from $\mathcal{A}$ (line 24 of Algorithm 1), and (iv) updates the probability distribution (line 25 of Algorithm 1). Each of these key steps are discussed in the following subsections.

Sampling The sampling procedure (Algorithm 2) uses the routine `Sample` developed in Informed RRT* (Gammell, Srinivasa, and Barfoot 2014, Algorithm 2). Informally speaking, that subroutine samples within a prolate hyperspheroid with the focal points being the start and goal configurations and the diameters defined by the cost. Any configurations out of this hyperspheroid cannot improve the current-best solution. In Fig. 1, this corresponds to sampling within the yellow ellipsoids as shown through the progressions.

In our sampling procedure, a fixed number of samples n_s is added to S' from X_f in each iteration. First, an edge $e := (u, v)$ is drawn from the probability distribution over $\mathcal{A}$ (line 3 in Algorithm 2). Initially, until a feasible S_T is obtained, the probability distribution is computed using the heuristic lower bounds (line 17 in Algorithm 1). Once a random sample x_{rand} is obtained for the edge (u, v) using `Sample`, it is added to S'.

Incremental Steiner tree algorithm This procedure is accomplished by first finding lower-cost, feasible paths between the terminals through `Ripple` (Algorithm 3), and then updating the spanning tree S_T. In `Ripple`, each sample is processed individually until S' becomes empty. Adding a new point s to the roadmap (consequently to G) can facilitate new paths between terminals. If s has a neighbor that is connected to a terminal (or is itself a terminal), then s and each of its neighbors are expanded in a *Dijkstra*-like fashion until the priority queue $\mathcal{Q}$ is empty (lines 13–21 of Algorithm 3). The variable $g(u)$ keeps track of the shortest path from u to its closest terminal which is stored in r_u. The key part of the `Ripple` algorithm lies in finding new feasible paths between the terminals. This happens (line 22 of Algorithm 3) during the expansion of a node u^* to its neighbor n when $g(u^*) + cost(u^*, n) \geq g(n)$ and $r_{u^*} \neq r_u$, *i.e.*, n is closer to its root r_n than to r_{u^*}. In this case, a feasible path from r_{u^*} to r_n has been discovered through the nodes u^* and n. If the cost of this feasible path is lower than $cost_T(r_{u^*}, r_n)$, then $cost_T(r_{u^*}, r_n)$ is updated accordingly.

After $cost_T(u, v)$ for any pair of terminals (u, v) is updated, it is relatively straightforward to check if any edge not in S_T should become part of S_T. First, we check if S_T is empty or not. If S_T is empty, then we just use Kruskal's algorithm (Kruskal 1956) to find a spanning tree (if it exists) for T (line 2 of Algorithm 4). If S_T is not empty, we appeal to the following well-known cycle property of minimum spanning trees to see if we should include (u, v) in S_T.

Algorithm 3: `Ripple`(S')

```
   // For any u ∈ V, let r_u denote the
      closest terminal to u in G. If u
      is a terminal, then r_u := u. The
      parent of any terminal is itself
1  while S' ≠ ∅ do
2  |   s ← S'.pop()
3  |   V ← V ∪ {s}
4  |   E ← E ∪ {(s, u) : u ∈ N_V(s)}
5  |   parent(s), r_s ← NULL
6  |   n* = arg min{g(n) + cost(s, n) : n ∈
   |    N_V(s), r_n ≠ NULL}
7  |   if n* ≠ NULL then
8  |   |   Ē ← Ē ∪ {(s, n*)}
9  |   |   g(s) ← g(n*) + c(s, n*)
10 |   |   r_s ← r_{n*}
11 |   |   parent(s) ← n*
12 |   |   Q.insert(s, g(s)) // Priority Queue
13 |   |   while Q ≠ ∅ do
14 |   |   |   u* ← Q.extractMin()
15 |   |   |   foreach n ∈ N_V(u*) do
16 |   |   |   |   if g(u*) + cost(u*, n) < g(n) then
17 |   |   |   |   |   g(n) ← g(u*) + cost(u*, n)
18 |   |   |   |   |   Q.insert(n, g(n))
19 |   |   |   |   |   r_n ← r_{u*}
20 |   |   |   |   |   Ē ←
   |   |   |   |   |    Ē\{(n, parent(n))∪{(n, u*)}}
21 |   |   |   |   |   parent(n) ← u*
22 |   |   |   |   else if r_n ≠ r_{u*} then
23 |   |   |   |   |   d ← g(n) + cost(n, u*) + g(u*)
   |   |   |   |   |    if d < cost_T(r_n, r_{u*}) then
24 |   |   |   |   |   |   cost_T(r_n, r_{u*}) ← d
25 return
```

Cycle Property. *Suppose that S_T is a MST in G_T and when a new edge $(u, v) \notin S_T$ is added to G_T, it forms a cycle $C = \Omega((u, v), S_T)$. Consider an edge $(u^*, v^*) \in C$ other than (u, v) that has the maximum of all the edge costs in C. If $cost_T(u^*, v^*) > cost_T(u, v)$, then $S_T \setminus \{(u^*, v^*)\} \cup \{(u, v)\}$ is an MST of the updated G_T.*

Thus, in our algorithm, if $cost_T(u^*, v^*) > cost_T(u, v)$, then edge (u^*, v^*) is deleted from S_T and edge (u, v) is added to S_T (lines 9 of Algorithm 4).

Pruning edges from $\mathcal{A}$ The pruning procedure (Algorithm 5) is similar to what we discussed in the previous tree update procedure except that we now appeal to the bounds on the cost of the edges. Consider an edge $(u^*, v^*) \in \Omega((u, v), S_T)$ other than (u, v) that has the maximum of all the edge costs in C. If $cost_T(u^*, v^*) < h(u, v)$, and

Algorithm 4: `UpdateTree`$(S_T, \mathcal{A})$

```
// Notation: Ω((u,v), S_T) is the cycle induced by adding edge (u,v) to S_T
if S_T is empty then
    S_T = Kruskal(cost_T)
    return S_T
for (u,v) ∈ A : (u,v) ∉ S_T do
    P := {(u',v') ∈ Ω((u,v), S_T) \ {u,v}}
    pathCost ← Σ{cost_T(u',v') : (u',v') ∈ P}
    (u*,v*) = arg max{cost_T(u',v') : (u',v') ∈ P}
    if cost_T(u*,v*) > cost_T(u,v) then
        S_T ← S_T \ {(u*,v*)} ∪ {(u,v)}
    else if pathCost < cost_T(u,v) then
        cost_T(u,v) ← pathCost // Update non-MST edge with a cheaper path via MST
return S_T
```

Algorithm 5: `PruneEdges`$(S_T, \mathcal{A})$

```
for (u,v) ∈ A : (u,v) ∉ S_T do
    (u*,v*) = arg max{cost_T(u',v') : (u',v') ∈ Ω((u,v), S_T) \ {u,v}}
    if h(u,v) > cost_T(u*,v*) then
        A ← A \ {(u,v)}
return A
```

since $h(u,v) \leq cost_T(u,v)$, we have $cost_T(u^*,v^*) < cost_T(u,v)$. Then, again applying the cycle property, edge (u,v) is pruned from $\mathcal{A}$ and is never considered by the algorithm henceforth (line 4 of Algorithm 5). This is can be seen in Fig. 1 where every non-MST edge gets pruned because its lower bound (straight line distance in this case) is more than the current cost of each of the edges it encloses in its MST cycle.

Update probability distributions Once a feasible S_T is found, the probability distribution is updated (Algorithm 6) in each iteration to reflect the changes in the costs and $\mathcal{A}$. We want to sample the region around an edge more if the uncertainty around the cost of the edge is higher. This is based on our assumption that the more we learn about the cost of an edge, the better we can decide on its inclusion in S_T. If an edge (u,v) already belongs to S_T, we assign a sampling probability for (u,v) proportional to the difference between the edge cost and its lower bound. On the other hand, if an edge (u,v) does not belong to S_T, we first aim to include it in S_T and later hope to drive its cost to its lower bound. Thus, we assign a sampling probability for (u,v) proportional to the difference between the edge cost and the cost of its next largest cost edge in $\Omega((u,v), S_T)$. An illustration of this can be seen in Fig. 1: In the middle subfigure, the MST edge 2 – 5 has highest deviation from its lower bound (compared to other MST edges). Thus, we would like to sample this edge's informed set more to reduce the uncertainity (deviation). This is reflected by the sampling probability which can be seen to be darker compared to rest. Consequently, in the next iteration, we significantly densify in its corresponding ellipsoid making the deviation from its lower bound approach 0.

Algorithm 6: `UpdateProbability`$(Prob, \mathcal{A}, S_T, h)$

```
if S_T is empty then
    return Prob
// Reset probability distribution
Prob(e) := 0 for all e ∈ {(u,v) : u,v ∈ T, u ≠ v}
gap_1(e) := cost_T(u,v) − h(u,v), ∀(u,v) ∈ S_T
gap_2(e) := cost_T(u,v) − max{c(e') : e' ∈ Ω(e, S_T) \ {e}}, ∀e ∈ A \ S_T
// Partition active edges
A_mst = {e ∈ S_T : gap_1(e) > 0}
A_non−mst = {e ∈ A \ S_T : gap_2(e) > 0}
foreach e ∈ A_mst do
    Prob(e) ← (|A_mst| / (|A_mst| + |A_non−mst|)) · gap_1(e) / Σ_{e'∈gap_1(e')}
foreach e ∈ A_non−mst do
    Prob(e) ← (|A_non−mst| / (|A_mst| + |A_non−mst|)) · gap_2(e) / Σ_{e'∈gap_2(e')}
return Prob
```

Theoretical Properties

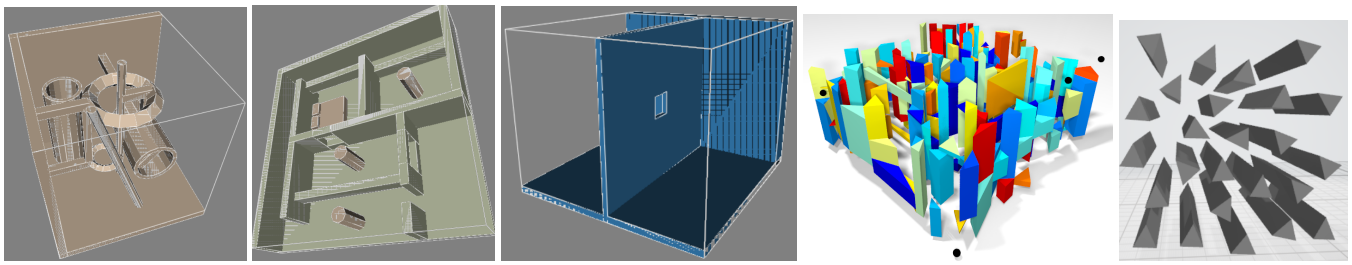
At any iteration of the algorithm, the algorithm maintains a Steiner tree S_T which is formed by expanding the paths of a MST on the terminal graph G_T. For any given state of the graph G that contains the terminals T along with the sampled points so far, we argue that the tree returned by `IST`* is such an MST.

Theorem 1. *`IST`* will return an MST over the shortest paths between terminals in G.*

We show this theorem by arguing that the edge pruning from the active set $\mathcal{A}$ of inter-terminal edges is correct and that the sampling procedure with updated probabilities coupled with the `Ripple` update will eventually find all relevant shortest paths. The former claim follows from the cycle property as discussed in the pruning step. To prove the latter claim, we use the following key lemma, that is proved in the Appendix.

Lemma 2. *Suppose a sample point $s \in S'$ is processed by `Ripple` so that r_s is a terminal t, then in G, t is (one of) the closest terminal to s among all terminals T.*

In this way, `Ripple` maintains every sample point in the correct so-called 'Voronoi' partition of the terminals (i.e. assigning it to its closest terminal). From this lemma, we see that the edges between terminals in G_T whose costs are updated by `Ripple` (line 24, Algorithm 3) are precisely those where an edge between a node u^* and its neighbor n such that $r_{u^*} \neq r_n$ discovers a new cheaper path between r_{u^*} and r_n. Thus, `Ripple` is able to find all shortest paths between



(a) *Abstract* (b) *Home* (c) *TwistyCool* (d) *Dense* (e) *Triangles*

Figure 2: The different 3D environments considered for evaluation with SFF* and Multi-T RRT. Figures (d) and (e) are taken from (Janoš, Vonásek, and Pěnička 2021) and those environments were designed by the respective authors while environments (a), (b), and (c) are part of the OMPL suite (Şucan, Moll, and Kavraki 2012).

pairs of terminals that are witnessed by a pair of neighboring boundary [g] nodes. We can now use a result of (Mehlhorn 1988) (Lemma in Section 2.2) that shows that this subgraph G_T of the subset of shortest paths between terminals in T that are witnessed by adjacent boundary nodes is sufficient to reconstruct an MST of G. This shows that S_T constructed as a MST of G_T is indeed an MST of the metric completion of T using all the edges in G as claimed.

Since the `UpdateTree` method in `IST*` always maintains a MST of the sampled graph G, in the limit with more sampling, the actual shortest paths corresponding to the true final MST will be updated to their correct lengths with diminishing error, and this MST will be output by `IST*`. Since the MST is a 2-approximation to the optimal Steiner tree in any graph and consequently, the MGPF problem (Kou, Markowsky, and Berman 1981), we get the following result.

Corollary 3. *`IST*` outputs a 2-approximation to the MGPF problem asymptotically.*

Results

We compared the performance of `IST*` with the existing approaches, namely SFF* (Janoš, Vonásek, and Pěnička 2021), Multi-T-RRT (Devaurs, Siméon, and Cortés 2014) and LazyTSP (Englot and Hover 2013). However, we found that these approaches timed out (took significantly greater time to find an initial solution) even in three-dimensional environments. Thus, for more holistic evaluation of `IST*`, we came up with a baseline, as described in the Background and Preliminaries section.

Specifically for the baseline, we densified a roadmap given by PRM* (Karaman and Frazzoli 2011) for a fixed amount of time via uniform random sampling after which S* (specifically, S*-BS variant) was run on the resulting roadmap to obtain an MST-based Steiner tree (Chour, Rathinam, and Ravi 2021). It was ensured that the combined time spent in growing the roadmap and running S* exhausted the input time limit.

The planners (`IST*` and the Baseline) were implemented in Python 3.7 using OMPL (Şucan, Moll, and Kavraki 2012) v1.5.2 on a desktop computer running Ubuntu 20.04, with 32 GB of RAM and an Intel i7-8700k processor. Further implementation details are available in the Appendix.

Due to the absence of a standard suite of instances for benchmarking planners for MGPF, we use the environments commonly considered in motion planning and extend them to our setting by randomly generating valid goal configurations. Specifically, we test our planners in rigid body motion planning instances available in the OMPL.app GUI and the environments considered by the authors of SFF*. A problem instance is uniquely identified by an environment (with its specific robot model), number of terminals, and the computational time given for planning. For each environment, a varying number of terminals ($|T| = 10, 30, 50$) was generated. The choice for the size of the problem instances was motivated by unmanned vehicle applications (Oberlin, Rathinam, and Darbha 2011). We now describe the testing environments and the corresponding results next.

Environments

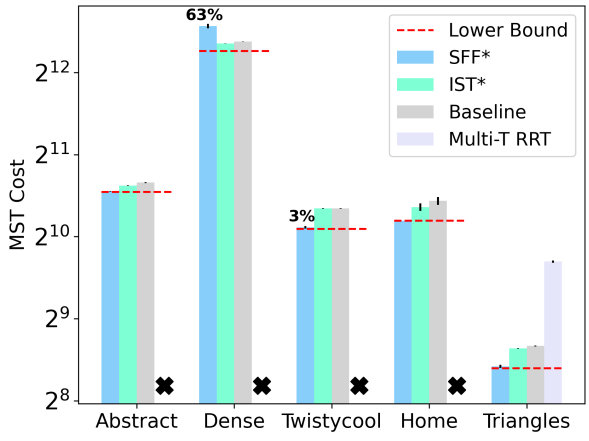
As the environment models in OMPL App are designed for single source and destination motion planning (with no goals), many of them have disconnected regions making them unsuitable for benchmarking multi-goal path planning under random generation of goal points. Thus, we used the $Home$, $Abstract$ and $Twisty\ Cool$ environment which admit one connected $SE(3)$ configuration space.

$Home$ (Fig. 2b) and $Abstract$ (Fig. 2a) are related to the classic Piano Mover's problem (Schwartz and Sharir 1983), admitting several narrow passages offering several homotopy classes of solution paths. Meanwhile, $Twisty\ Cool$ (Fig. 2c) represents a cubicle divided into two regions by a wall in between which has a narrow window connecting the two regions.

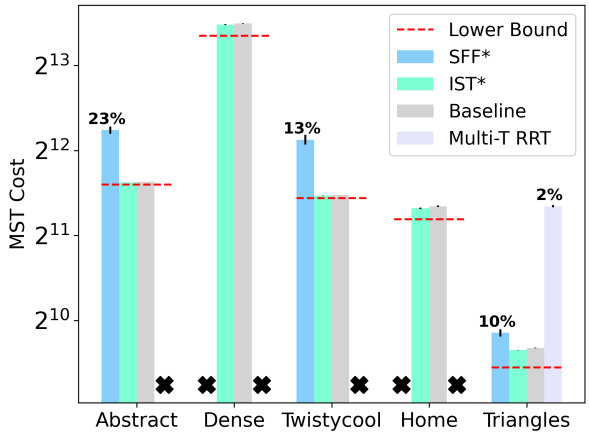
Apart from OMPL environments, we also considered $3D$ environments ($6D$ configuration space) introduced by the authors of SFF*; specifically, $Dense$ (Fig. 2d) and $Traingles$ (Fig. 2e). $Triangles$ has multiple triangular prisms of same size symmetrically placed as obstacles while $Dense$ represents a city-like environment with various free-form building-like structures, placed irregularly, acting as obstacles.

The spatial size (volume) of each of the environments is shwon in Table 1.

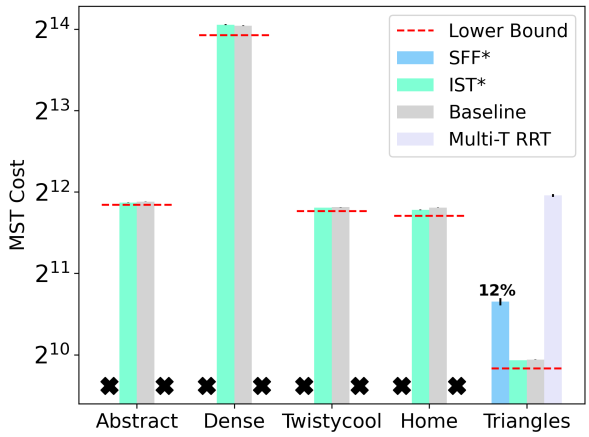
[g] A node u is said to be a *boundary* node if not all its neighbors have root r_u.



(a) 10 Terminals



(b) 30 Terminals



(c) 50 Terminals

Figure 3: Performance of the different approaches on the environments considered in Table 1. The plots represent the mean (across 30+ runs) with error bars (standard deviation). The number on top of a bar represents the percentage of **failure** while its absence indicates no failures. A cross ✗ represents 100% failure (timeout in all runs). *Note that the y-axis is in log scale.*

Table 1: Time (in seconds) given to planners as input in different three-dimensional problem instances.

Environment \ Terminals	10	30	50
HOME	$450s$	$1200s$	$1800s$
ABSTRACT	$300s$	$1200s$	$1800s$
TWISTY COOL	$600s$	$1200s$	$1800ss$
DENSE	$600s$	$1200s$	$1800s$
TRIANGLES	$10s$	$150s$	$300s$

Comparison with the State of the Art

Given the stochastic nature of sampling-based path planning methods, all the approaches considered for evaluation were ran 30+ times to obtain a distribution of the cost of the Steiner tree found by them. First, we note that IST* and the Baseline find a (initial) solution for all the problem instances considered within 5 minutes. Meanwhile, it was found that LazyTSP took more than 45 minutes across all the problem instances so we explicitly exclude it from further discussion. Next, Multi-R RRT, unlike SFF*, IST* and the Baseline, stops the moment it finds a solution and doesn't improve further. Thus, we tried to find a rough average of the time taken by Multi-T-RRT to solve a problem instance. In the cases where it finished under an acceptable threshold (30 minutes[h]), we recorded its duration and provided the same time to SFF*, IST* and the Baseline for *fair* comparison. Otherwise, they were given (the same) time based on the number of terminals and difficulty of planning in that environment. The exact time for each instance finally used is shown in Table 1.

The results are shown as bar plots for each planner across environments in Fig. 3. To make them more insightful, we also calculate a lower bound on the cost of the MST (described in Appendix) in each problem instance and show it as a dashed red bar in the figures. It represents the best possible cost achievable for that instance.

We find that Multi-T-RRT was only able to find solutions in the $Triangles$ environment which has relatively less volume compared to other environments. This suggests that Multi-T-RRT is only suitable for low-dimensional environments as it timed out even for 10 terminals in other environments. Furthermore, the solutions it found were 4 times worse than other planners which is not surprising given its nature of stopping at the first solution found.

SFF* is more promising that Multi-T-RRT, finding solutions across all environments for 10 terminals, across more than half of the environments for 30 terminals but only in $Triangles$ when given 50 terminals. Further, it excels in 10 terminals case, reaching the optimal solution in 4 out of the 5 environments consequently outperforming IST*. However, when considering more number of terminals, it either finds a poor solution, worse than our Baseline (30 terminals) or

[h]Given both IST* and the Baseline finds a solution within 5 minutes, a significantly larger time period (30 minutes) was chosen to be the upper limit.

times out (50 terminals). SFF* further has more variance and higher rate of failure. Meanwhile, our Baseline consistently finds good solutions across all instances and performs only marginally worse to IST* making it a competitive alternative for future evaluations.

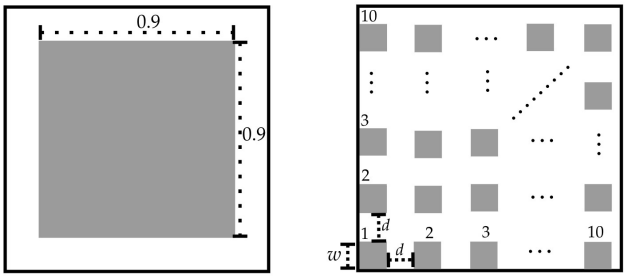
Overall, we find SFF* is best suited for the cases when the number of terminals is low. In all other cases, IST* is superior to it, consistent, asymptotically optimal, and also performs good with low number of terminals. A final point of distinction we would like to emphasize is that the available implementation of Multi-T-RRT and SFF*[i] only supports $SE(3)$ (and its subspace), and thus, cannot be evaluated on environments like $\mathbb{R}^4$ and $\mathbb{R}^8$ presently. Further, SFF* was primarily designed for $SE(3)$ and lower-dimensional environments given its dense space filling nature while Mutli-T-RRT only tries to find *a solution* and never improves on it. IST* overcomes all these limitations and is scalable to higher dimensions as we show below.

Extensive Comparison with Baseline

Given the similar final solution cost of the Baseline with IST*, we now try to investigate in detail the difference in their performance, especially in higher dimensions. We let both the planners output the cost of the Steiner tree found *over time*. We ran each planner 50 times to obtain a distribution of solution costs. The mean solution cost with a 99% confidence interval is calculated from 50 trials and shown in Table 2.

Real-vector space problems

IST* and the Baseline were tested on two simulated problems in a unit hypercube with distinct obstacle configurations in $\mathbb{R}^4$ and $\mathbb{R}^8$ (Fig. 4). The collision detection resolution was set to 10^{-4} to make evaluating edge costs computationally expensive. The time given to each planner is available in Table 5 (in Appendix).



(a) Center Obstacle (CO) (b) Uniform Hypercubes (UH)

Figure 4: A 2-dimensional illustration of the problems in real-vector spaces. The configuration space for both is a unit hypercube, $i.e.$, each state space was bounded to the interval [0, 1], illustrated by the black bounding box. Solid regions represent invalid states. In Fig.4(b), the hypercube obstacles are axis-aligned (specifically, 10 on each axis) and uniformly spread with $w = 0.075, d = 0.025$.

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

Center Obstacle (CO): This problem consisted of a big hypercube at the center (with volume $(0.9)^n$ in $\mathbb{R}^n$) (Fig. 4a). In this environment, IST* started on par with the Baseline but significantly improved while the Baseline remained stuck in a sub-optimal solution.

Uniform Hypercubes (UH): This domain was filled with a regular pattern of axis-aligned hypercubes (identically distributed with uniform gaps) (Fig. 4b). For a real vector space of dimension n, this environment contains 10^n obstacles with a total volume of 0.75^n. In this environment, we see a dominant performance of IST* over the Baseline with up to 30% better solution cost.

SE(3) problems

Among the OMPL environments, $Home$ offers the highest scope for optimization among all the environments considered, with the final solution often being 50% better than the initial solution. In $Home$, Baseline performs competitively to IST*, and sometimes also finds better solutions initially. However, IST* converges to a better solution asymptotically in all the problem instances of $Home$. In $Abstract$, both Baseline and IST* find a good solution quickly, and as a result, the scope for optimization is not much. Still, IST* is able to improve very quickly, dominates throughout, and reaches the final solution cost of Baseline ≈ 6 times faster.

Impact of Ripple

In each iteration of IST*, a batch of samples[j] is added to the roadmap G. As Ripple processes each sample incrementally and keeps rewiring a rooted shortest-path tree until convergence, it may seem better to simply add all the samples at once and then run S* on the updated roadmap to get S_T. However, the roadmap grows significantly over time, making S* computationally expensive for repeated executions. Whereas, Ripple only has to process a small fraction of the roadmap for a fixed number of samples per batch. Note that Ripple uses the same amount of memory as S* given that the underlying graph is same for both. However, the time taken by each may vary, significantly.

We benchmarked Ripple with S* to investigate its effect on the performance of IST*. A pseudorandom generator was used for generating the *same* sample points for both. The problem instances considered for this comparison were same as before (Table 5). However, due to space constraints, we only show the results (in Table 3) for just 2 environments depicting extreme scenarios[k]. In UH $\mathbb{R}^8$, both S* and Ripple perform similarly, with Ripple being marginally better in the instance with 50 terminals. In $Home$, Ripple not only finds a better final solution, but is also $2-3$x faster than S* for the same solution cost consistently. In all other problem instances as well, these 2 scenarios were observed throughout: either Ripple's performance was very similar to S* or it was significantly faster while converging to a better solution. In conclusion, while both S* and Ripple can

[j]Note that the batch size is a hyperparameter and remains fixed throughout the execution.

[k]The plot on rest of the problem instances with further discussion is available in the Appendix.

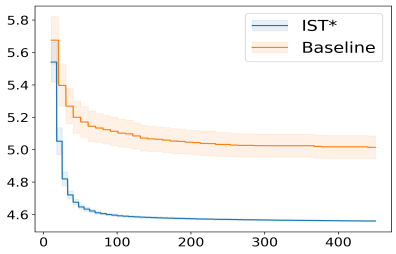
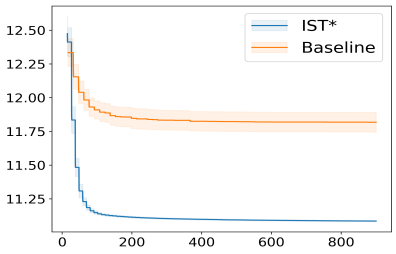
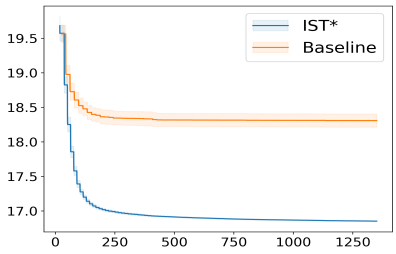
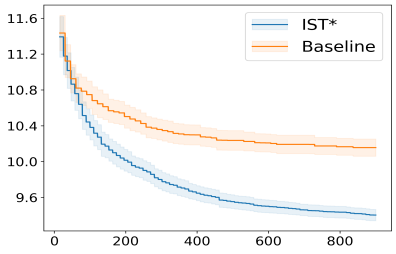
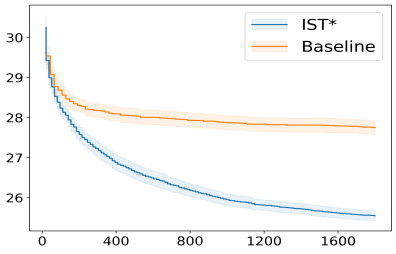
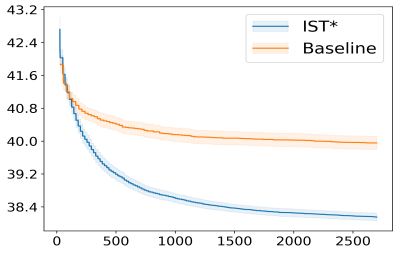
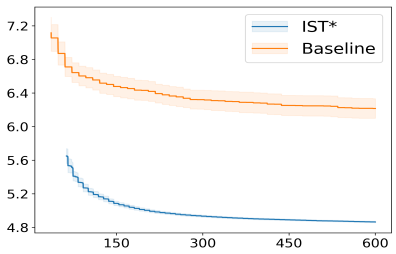
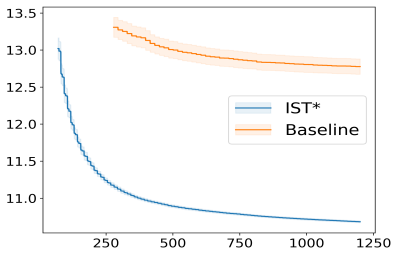
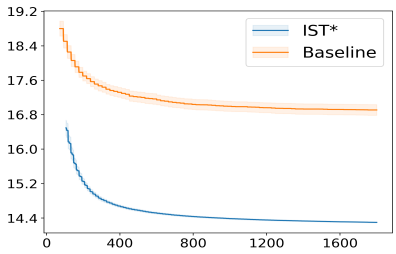
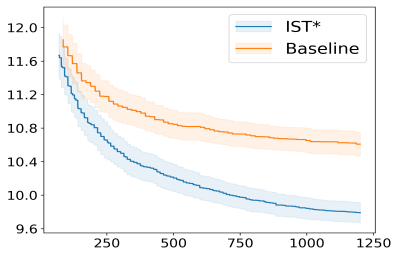
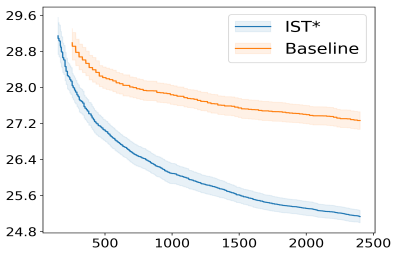
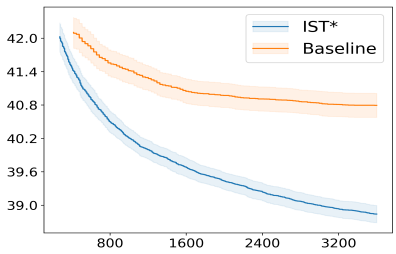
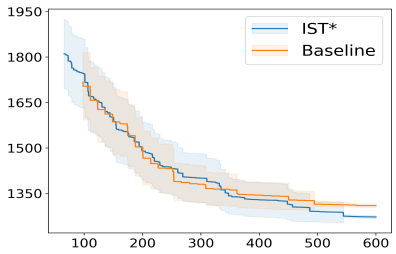
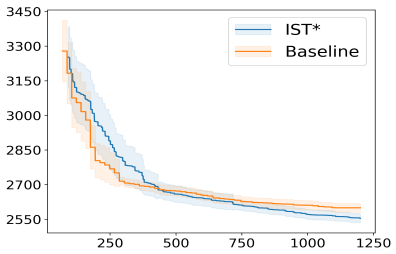
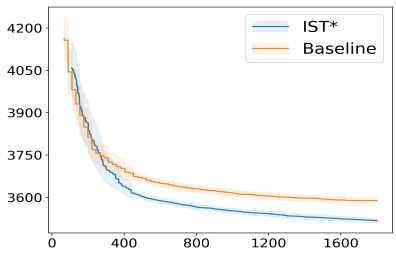
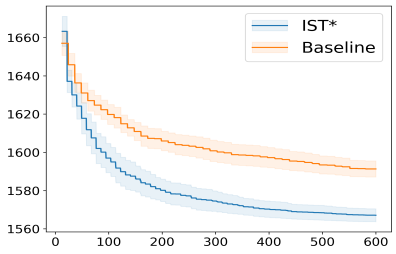
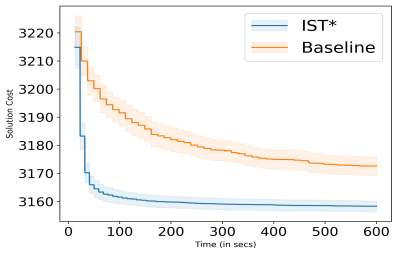
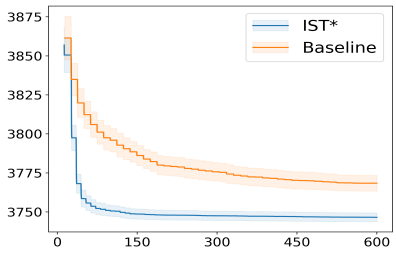
Env.	10 Terminals	30 Terminals	50 Terminals
CO $\mathbb{R}^4$			
CO $\mathbb{R}^8$			
UH $\mathbb{R}^4$			
UH $\mathbb{R}^8$			
HOME			
ABSTRACT			

Table 2: Comparison of IST* against the Baseline across 6 different environments – real vector spaces over uniform hyper rectangles (UH) and center obstacles (CO), and $SE(3)$ environments HOME and ABSTRACT – over 10/30/50 terminals. The y-axis represents the solution cost (i.e., Steiner tree cost) while the x-axis is the time horizon (in seconds). The thin solid line represents the mean solution cost with the shaded region being the 99% confidence interval about this mean.

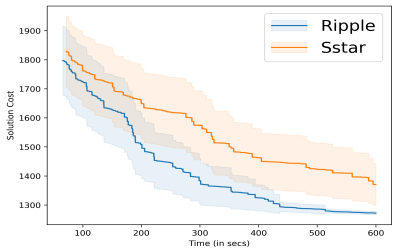
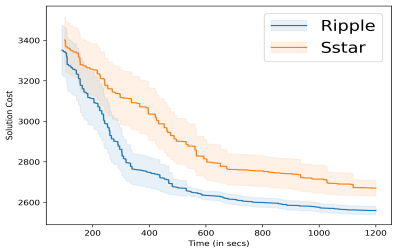
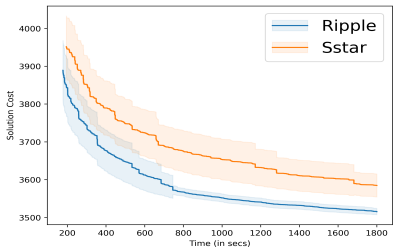
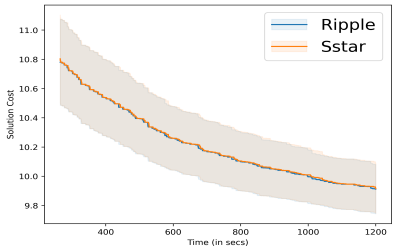
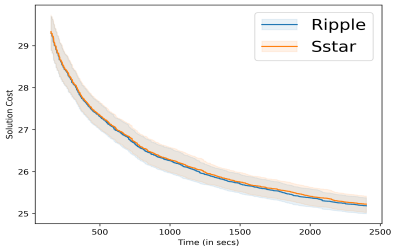
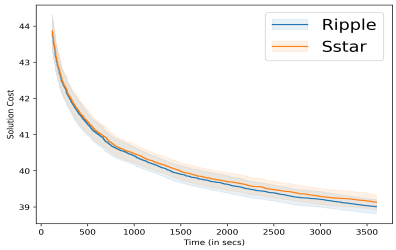
Env.	10 Terminals	30 Terminals	50 Terminals
HOME			
UH $\mathbb{R}^8$			

Table 3: Comparison of the performance of IST* when Ripple is used against S* on an $SE(3)$ instance ($Home$) and a high-dimensional environment (UH $\mathbb{R}^8$). The dark line represents the mean solution cost with the thick region being the 99% confidence interval about the mean.

lead to the optimal MST over G_T, Ripple dominates S* across all instances considered.

Comparing Path Costs

While IST*'s primary output is a Steiner tree, we also evaluated the cost of the feasible path obtainable from IST* for MGPF (details provided in Appendix). Overall, it was seen to perform better than the Baseline in path computations too, an example of which is shown on two challenging problem instances in Fig. 5.

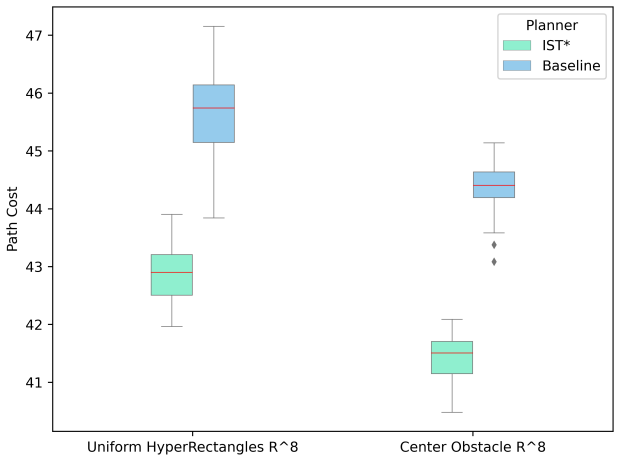


Figure 5: Comparisons of the costs of the feasible MGPF solution found by the planners in the high-dimensional real environments with 50 terminals (over 50 runs).

Conclusion

Inspired by the recent advancements in sampling-based methods for shortest path problems for single source and destination in continuous space and multi-goal path finding in discrete space, we provide a unifying framework to promote a novel line of research in MGPF. For the first time, to the best of the authors' knowledge, the ideas of informed sampling have been fruitfully extended to the setting of planning for multiple goals. Our approach is decoupled in the sense that any further advancements in PRM* or informed sampling can be directly used to update the respective components of our proposed framework.

In the future, we plan to explore obtaining effective lower bounds on the shortest path cost between two terminals instead of using the Euclidean distance. Including heuristics in Ripple also seems to be a promising line of research.

References

Best, G.; Faigl, J.; and Fitch, R. 2016. Multi-robot path planning for budgeted active perception with self-organising maps. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 3164–3171. IEEE.

Canny, J.; and Reif, J. 1987. New lower bound techniques for robot motion planning problems. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, 49–60. IEEE.

Chour, K.; Rathinam, S.; and Ravi, R. 2021. S*: A Heuristic Information-Based Approximation Framework for Multi-Goal Path Finding. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, 85–93.

Decroos, T.; De Causmaecker, P.; and Demoen, B. 2015. Solving Euclidean Steiner tree problems with multi swarm optimization. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, 1379–1380.

Devaurs, D.; Siméon, T.; and Cortés, J. 2014. A multi-tree extension of the Transition-based RRT: Application to ordering-and-pathfinding problems in continuous cost spaces. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2991–2996. IEEE.

Edelkamp, S.; and Plaku, E. 2014. Multi-goal motion planning with physics-based game engines. In *2014 IEEE Conference on Computational Intelligence and Games*, 1–8. IEEE.

Edelkamp, S.; Secim, B. C.; and Plaku, E. 2017. Surface inspection via hitting sets and multi-goal motion planning. In *Annual Conference Towards Autonomous Robotic Systems*, 134–149. Springer.

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

Faigl, J. 2016. On self-organizing map and rapidly-exploring random graph in multi-goal planning. In *Advances in self-organizing maps and learning vector quantization*, 143–153. Springer.

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

Faigl, J.; Kulich, M.; Vonásek, V.; and Přeučil, L. 2011. An application of the self-organizing map in the non-Euclidean Traveling Salesman Problem. *Neurocomputing*, 74(5): 671–679.

Faigl, J.; Váňa, P.; and Deckerová, J. 2019. Fast heuristics for the 3-D multi-goal path planning based on the generalized traveling salesman problem with neighborhoods. *IEEE Robotics and Automation Letters*, 4(3): 2439–2446.

Faigl, J.; Váňa, P.; and Drchal, J. 2020. Fast sequence rejection for multi-goal planning with dubins vehicle. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 6773–6780. IEEE.

Faigl, J.; Vonásek, V.; and Přeučil, L. 2013. Visiting convex regions in a polygonal map. *Robotics and Autonomous Systems*, 61(10): 1070–1083.

Fort, J. 1988. Solving a combinatorial problem via self-organizing process: An application of the Kohonen algorithm to the traveling salesman problem. *Biological cybernetics*, 59(1): 33–40.

Friedrich, C.; Csiszar, A.; Lechler, A.; and Verl, A. 2018. Efficient Task and Path Planning for Maintenance Automation Using a Robot System. *IEEE Transactions on Automation Science and Engineering*, 15(3): 1205–1215.

Gammell, J. D.; Barfoot, T. D.; and Srinivasa, S. S. 2018. Informed sampling for asymptotically optimal path planning. *IEEE Transactions on Robotics*, 34(4): 966–984.

Gammell, J. D.; Srinivasa, S. S.; and Barfoot, T. D. 2014. Informed RRT*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2997–3004. IEEE.

Gammell, J. D.; Srinivasa, S. S.; and Barfoot, T. D. 2015. Batch informed trees (BIT*): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs. In *2015 IEEE international conference on robotics and automation (ICRA)*, 3067–3074. IEEE.

Garrett, C. R.; Chitnis, R.; Holladay, R.; Kim, B.; Silver, T.; Kaelbling, L. P.; and Lozano-Pérez, T. 2021. Integrated task and motion planning. *Annual review of control, robotics, and autonomous systems*, 4: 265–293.

Hauser, K. 2015. Lazy collision checking in asymptotically-optimal motion planning. In *2015 IEEE international conference on robotics and automation (ICRA)*, 2951–2957. IEEE.

Helsgaun, K. 2000. An effective implementation of the Lin–Kernighan traveling salesman heuristic. *European journal of operational research*, 126(1): 106–130.

Janoš, J.; Vonásek, V.; and Pěnička, R. 2021. Multi-goal path planning using multiple random trees. *IEEE Robotics and Automation Letters*, 6(2): 4201–4208.

Jünger, M.; Reinelt, G.; and Rinaldi, G. 1995. The traveling salesman problem. *Handbooks in operations research and management science*, 7: 225–330.

Karaman, S.; and Frazzoli, E. 2011. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7): 846–894.

Kavraki, L.; Svestka, P.; Latombe, J.-C.; and Overmars, M. 1996. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4): 566–580.

Kou, L.; Markowsky, G.; and Berman, L. 1981. A fast algorithm for Steiner trees. *Acta Informatica*, 15(2): 141–145.

Kruskal, J. B. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1): 48–50.

Kuffner, J.; and LaValle, S. 2000. RRT-connect: An efficient approach to single-query path planning. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, volume 2, 995–1001 vol.2.

Lin, C.-W.; Chen, S.-Y.; Li, C.-F.; Chang, Y.-W.; and Yang, C.-L. 2008. Obstacle-avoiding rectilinear Steiner tree construction based on spanning graphs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(4): 643–653.

Lin, S.; and Kernighan, B. W. 1973. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2): 498–516.

Liu, Z.; Wei, H.; Wang, H.; Li, H.; and Wang, H. 2022. Integrated Task Allocation and Path Coordination for Large-Scale Robot Networks With Uncertainties. *IEEE Transac-*

tions on Automation Science and Engineering, 19(4): 2750–2761.

Lumelsky, V. J.; and Stepanov, A. A. 1987. Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 2(1): 403–430.

Macharet, D. G.; and Campos, M. F. 2018. A survey on routing problems and robotic systems. *Robotica*, 36(12): 1781–1803.

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

McMahon, J.; and Plaku, E. 2021. Dynamic Multi-Goal Motion Planning with Range Constraints for Autonomous Underwater Vehicles Following Surface Vehicles. In *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*, 704–711. IEEE.

Mehlhorn, K. 1988. A faster approximation algorithm for the Steiner problem in graphs. *Information Processing Letters*, 27(3): 125–128.

Oberlin, P.; Rathinam, S.; and Darbha, S. 2011. Today's Traveling Salesman Problem. *Robotics & Automation Magazine, IEEE*, 17: 70 – 77.

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

Rathinam, S.; Sengupta, R.; and Darbha, S. 2007. A Resource Allocation Algorithm for Multivehicle Systems With Nonholonomic Constraints. *IEEE Transactions on Automation Science and Engineering*, 4(1): 98–104.

Saha, M.; Roughgarden, T.; Latombe, J.-C.; and Sánchez-Ante, G. 2006. Planning tours of robotic arms among partitioned goals. *The International Journal of Robotics Research*, 25(3): 207–223.

Salzman, O.; and Halperin, D. 2015. Asymptotically-optimal motion planning using lower bounds on cost. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 4167–4172. IEEE.

Schwartz, J. T.; and Sharir, M. 1983. On the "piano movers'" problem I. The case of a two-dimensional rigid polygonal body moving amidst polygonal barriers. *Communications on pure and applied mathematics*, 36(3): 345–398.

Şucan, I. A.; Moll, M.; and Kavraki, L. E. 2012. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4): 72–82. https://ompl.kavrakilab.org.

Vaněk, P.; Faigl, J.; and Masri, D. 2014. Multi-goal trajectory planning with motion primitives for hexapod walking robot. In *2014 11th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, volume 2, 599–604. IEEE.

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

Warsame, Y.; Edelkamp, S.; and Plaku, E. 2020. Energy-aware multi-goal motion planning guided by monte carlo search. In *2020 IEEE 16th International Conference on Automation Science and Engineering (CASE)*, 335–342. IEEE.

Zăvoianu, A.-C.; Saminger-Platz, S.; Entner, D.; Prante, T.; Hellwig, M.; Schwarz, M.; and Fink, K. 2018. Multi-objective optimal design of obstacle-avoiding two-dimensional Steiner trees with application to ascent assembly engineering. *Journal of mechanical design*, 140(6).

Appendix

Detailed Literature Review

The MGPF problem belongs to a general class of integrated task and motion planning problems which have been studied in the literature (Garrett et al. 2021; Liu et al. 2022; Friedrich et al. 2018; Rathinam, Sengupta, and Darbha 2007). Here, we provide an extensive review of the work relevant to the MGPF in continuous domains.

In (Devaurs, Siméon, and Cortés 2014), a meta-heuristic similar to *simulated annealing* is combined with multiple Rapidly-exploring Random Tree (RRT) expansions to solve MGPF. In (Englot and Hover 2013), LazyTSP was introduced for efficient path planning that combines RRT-based planning with TSP (Jünger, Reinelt, and Rinaldi 1995). It starts with TSP to compute an initial tour and then verifies the connections using RRT-based planning, iteratively refining the tour until a valid trajectory is found. In (Vonásek and Pěnička 2019), a method called *Space Filling Forests* (SFF) is proposed to solve MGPF. Multiple trees are grown from the nodes in T and unlike the approach in (Devaurs, Siméon, and Cortés 2014) where any two close enough (or neighboring) trees are merged into a single tree, in SFF, multiple virtual connections are allowed between two neighboring trees leading to multiple paths between any two nodes in T. Recently, in (Janoš, Vonásek, and Pěnička 2021), a generalization of SFF (called SFF*) has been proposed to also include rewiring of the edges in the trees similar to RRT*. Computational results show that SFF* performed the best among several previously known solvers for the simulation environments considered in (Janoš, Vonásek, and Pěnička 2021).

A generalization of MGPF was considered in (Saha et al. 2006) where all the goals were partitioned into groups and the aim is to also visit one goal from each partition. They work under the assumption that a path between two nodes is computed at most once which may, in reality, be far from the optimal path. Their planner stops the instant it has found a feasible path connecting all the goals unlike our approach which keeps refining the paths.

Un-supervised learning approaches (Faigl et al. 2011; Faigl 2016; Fort 1988) using Self Organizing Maps (SOMs) have been used to solve MGPF. SOM is a two-layer neural network and is a iterative procedure that provides a non-linear mapping between a higher dimensional space (set of goals) to a lower dimensional one (goal ordering). SOM also relies on an underlying distance metric, which can be Euclidean (Fort 1988), visibility graph-based (Faigl et al. 2011; Faigl, Vonásek, and Přeučil 2013), or sampling-based graph approximation (Faigl 2016; Vaněk, Faigl, and Masri 2014). In (Faigl 2016), SOM is combined with a rapidly exploring random graph algorithm to find feasible solutions for the MGPF. However, in comparison to our asymptotic 2-approximation guarantee, there is no guarantee on the quality of solutions obtained using SOMs.

In (Faigl et al. 2011; Faigl, Vonásek, and Přeučil 2013), a point robot moves in an environment represented as a polygonal domain, and a convex polygon partition is used for workspace decomposition. Such assumptions may not be straightforward to generalize, for example, such a decomposition is not possible even in common robotic environments like SE(3); thus, making them incomparable to IST* in the environments considered.

Multi-goal motion planning has also been explored in application to data collection. (Faigl and Hollinger 2014) consider this problem in planar environment with neighborhood and prize on the goals. The vehicle is required to visit only the neighborhood of each goal and all the goal (regions) are not given the same priority. This makes their problem significantly different from the one we have addressed. Similarly, (McMahon and Plaku 2021) have considered rewards on goals but also assume the fact that both goals and rewards are unknown to the operating vehicle in the beginning and become known only when a goal comes under its sensing radius.

(Faigl, Váňa, and Deckerová 2019) consider the generalized traveling salesman problem with neighborhoods (GTSPN) *specifically for 3D environments*, where an individual neighborhood may consist of multiple regions (thus, forming a neighborhood set), and the problem is to determine a shortest multi-goal path to visit at least one region of each neighborhood. They propose two heuristics for solving the GTSPN with neighborhoods defined as polyhedra and ellipsoids to quickly find a feasible solution. The heuristics exploit properties of the 3D instances with convex regions making their approach inapplicable for comparison with ours. Further, they mention that an extension of their proposed approach for high-dimensional problems is a subject of their future work, which precisely forms the motivation for our work.

Other papers have addressed variations of multi-goal motion planning taking into account additional constraints like Dubins vehicle (Faigl, Váňa, and Drchal 2020), energy-aware planning with recharging stations (Warsame, Edelkamp, and Plaku 2020), availability of physics-based game engine to model dynamics (Edelkamp and Plaku 2014) which are outside our scope.

There is also an extensive literature on efficiently constructing minimal-cost Euclidean Steiner Tree spanning a set of vertices on a plane avoiding obstacles, having applications in VLSI designs and ascent assembly engineering (Zăvoianu et al. 2018; Lin et al. 2008; Decroos, De Causmaecker, and Demoen 2015). However, all such works assume the underlying roadmap graph (whose nodes act as Steiner points) to be given up-front as input rather than constructing and refining it over time to reduce the path cost.

Finally, we would like to emphasize that the major difference in our proposed approach with respect to prior work is *to adapt informed sampling* (with a pruning method based on the current bounds on the edge lengths) *to the setting of multiple goals enabling faster convergence to an asymptotically 2-approximate solution.*

Key Lemma on Ripple

We will prove Lemma 2 here by induction. At the beginning of the algorithm, G consists only of the terminals $t' \in T$ for which $r_{t'} = t'$ so the claim is trivially true. When a new sample point s is processed, it is assigned to its neighbor n with the smallest value of $g(n) + cost(s, n)$ (line 6 of Al-

gorithm 3) via which it traces a shortest path to its closest terminal (by the definition of the g-values of its neighbors). The subsequent ripple of updates via the priority queue update the value of $g(n)$ and the closest terminals for neighbors n who have just discovered their shortest path to a terminal via the currently processed u^*, and they are entered into a priority queue for further expansion. In this way, if a new shorter path via the newly added sample node s arises in G, the `Ripple` update discovers and updates this information correctly hence maintaining the inductive invariant of the lemma that r_u is (one of) the closest terminals to u among all terminals T for every node $u \in G$.

Connection Radius

The connection radius ρ (used in determining the neighbors of a node) controls the sparseness of the roadmap graph (G). High values of ρ will lead to a dense roadmap making graph-search algorithms (like `Ripple`) computationally expensive while low values will make the roadmap sparse but may also cause it to be disconnected. As the module PRM* from OMPL was used as the underlying roadmap for both `IST*` and the Baseline, it adaptively limits the connections in G as the number of samples increase by decreasing the radius ρ (generally done to make the minimal number of connections required to ensure asymptotic optimality), as proposed in (Karaman and Frazzoli 2011)

$$\rho(q) := \eta\left(2\left(1+\frac{1}{d}\right)\left(\frac{\lambda(X_f)}{\zeta_d}\right)\left(\frac{log(q)}{q}\right)\right)^{\frac{1}{d}}$$

where q is the number of points sampled in X_f (ie., $|V|$), d is the dimension of X, $\eta > 1$ is a tuning parameter, and $\lambda(X_f)$ is Lebesgue measure of the obstacle-free space and ζ_d is the volume of the unit ball in the d-dimensional Euclidean space.

Pruning

Lower Bound on Edges A critical phase in `IST*` is pruning edges of G_T which cannot be part of the MST in future. Success of this phase relies heavily on how close are the lower bounds of these edges to their optimal cost. Past research has been focused mostly on finding lower bounds for special instances of the general motion planning problem (Canny and Reif 1987; Lumelsky and Stepanov 1987). A separate line of research recently has been focused on calculating the lower bound on **the past cost obtainable from the current set of samples** (Salzman and Halperin 2015). However, these are not lower bound on the optimal path cost but only representative of the lower bound derived from the discrete approximation of the state space. Thus, due to lack of effective worst-case lower bound on the optimal path cost for the general motion planning problem, we use the Euclidean distance between two points in X_f as the heuristic estimate.

Numerical Results In Table 4, we show the impact of the pruning condition we have developed. Even though we have a very simple heuristic estimate as the lower bound, we are able to prune a significant number of edges. With more terminals, it is likely that edges in the MST of G_T will be a straight-segments in X_f (hence, the optimal path is quite close to Euclidean distance). This is turn should lead to more pruning which we observe as a trend in Table 4.

Table 4: Average percentage of edges pruned from G_T by `IST*` in each problem instance across 50 runs.

Environment \ Terminals	10	30	50
`CO` $\mathbb{R}^4$	91%	98%	98%
`CO` $\mathbb{R}^8$	64%	84%	90%
`UH` $\mathbb{R}^4$	92%	97%	98%
`UH` $\mathbb{R}^8$	37%	79%	81%
HOME	87%	91%	98%
ABSTRACT	97%	98%	99%

While the pruning statistics may look appealing, it is possible to generate instances where no edges would be pruned.

Worst Case Consider a problem instance with a star-shaped obstacle in the center with terminals at the concave openings between two sharp ends of the star obstacle. Further, the placement of terminals and the concave openings should be such that all terminals are near to each other while the optimal path is long and convoluted. For example, in Figure 6, lower bound between every edge is less than the optimal cost between any two distinct terminals which means no edge will be pruned in G_T.

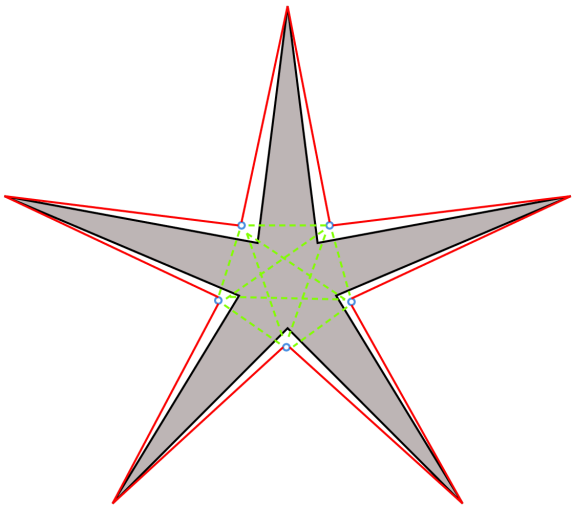


Figure 6: A worst-case problem instance for *pruning*: Light brown star denotes the obstacle region. Small blue hollow circles represent the terminals in T to be connected. The dashed green line is the Euclidean distance (lower bound) between the terminals while the red lines are the optimal paths.

Lower Bound Computation

A lower bound in the results can provide a more comprehensive and informative view when comparing the performance of different methods. Recall that a trivial lower bound for any edge between two nodes is the Euclidean distance between them. Thus, we can create a lower bound graph, where each edge between two nodes is the Euclidean distance and calculate the Minimum Spanning Tree on this graph to obtain a valid lower bound on the original problem. While this may seem to be a loose bound, many planners actually come fairly close to it (Fig. 3) establishing that lower bound was not far from the optimal solution.

Table 5: Time (in seconds) given to IST* and the Baseline as input in different problem instances.

Environment \ Terminals	10	30	50
CO $\mathbb{R}^4$	$450s$	$900s$	$1350s$
CO $\mathbb{R}^8$	$900s$	$1800s$	$2700s$
UH $\mathbb{R}^4$	$600s$	$1200s$	$1800s$
UH $\mathbb{R}^8$	$1200s$	$2400s$	$3600s$
$HOME$	$600s$	$1200s$	$1800s$
$ABSTRACT$	$600s$	$600s$	$600s$

Implementation Details

While we mention that a fixed number of samples is added to the roadmap in the pseudocode of IST*, the implementation could not exactly follow this as such an operation is not allowed in the available module for PRM*. However, OMPL does support growing the roadmap for a *fixed amount of time*. Hence, we had a time-based implementation where in each iteration of IST*, the roadmap was grown for the same amount of time. Consequently, the number of samples added per iteration was not fixed and varied depending upon the time spent in collision-checking for each new sample point. We let IST* and the Baseline run for all of the provided time.

Both the planners used the implementation of PRM* from OMPL (with default parameters). We also tried LazyPRM* (Hauser 2015) from OMPL but it performed worse than PRM*, contrary to our expectations. However, our approach (subsequently, the codebase of IST*) is modular such that LazyPRM* or any other implementation of the underlying roadmap can be used to interface with our planner.

Path Cost Computations To evaluate the cost of the feasible path obtainable from IST* for MGPF (results shown in Fig. 3), Baseline and IST* were given the same time as earlier (mentioned in Table 3) to compute the Steiner Tree. This was followed by computing shortest path in the roadmap graph G for every pair of goal nodes to obtain a distance matrix which was passed to the Lin-Kernighan heuristic (LKH) (Lin and Kernighan 1973; Helsgaun 2000). LKH was run for 100 iterations for both the Baseline and IST* to get the final path cost for MGPF.

Ripple : More Results and Discussion

We show the comparison of Ripple with S* on more environments in Table 6. In $Abstract$ with 10 terminals, we see Ripple performing better throughout while in all other instances its performance is almost same as Baseline. In real-vector space instances, it was observed that size of the roadmap in the end was much smaller compared to instances like $Home$ or $Abstract$. As the size of roadmap was tiny, so we couldn't witness the benefits of Ripple. We believe this was a consequence of our implementation in Python.

Real environments like CO and HR (Fig. 2) were custom defined in Python so when the roadmap was grown for both IST* and the Baseline, OMPL's planners in C++ had to interface with the collision checker of these environments in Python for each point sampled in the configuration space. This made the growth of PRM in these instances extremely slow due to the constant back and forth call between Python and C++. Thus, the size of the roadmap G in these environments was pretty small compared to $Home$ or $Abstract$ which are defined in OMPL App itself.

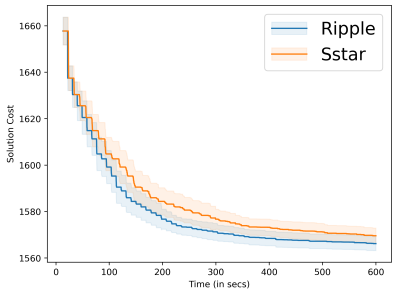
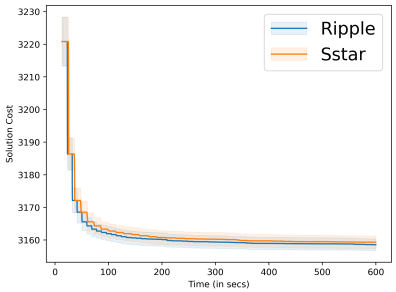
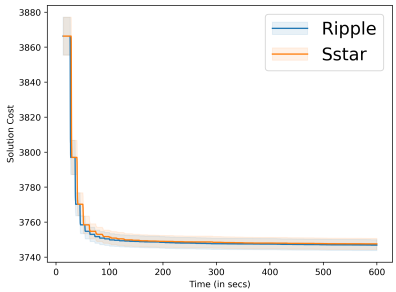
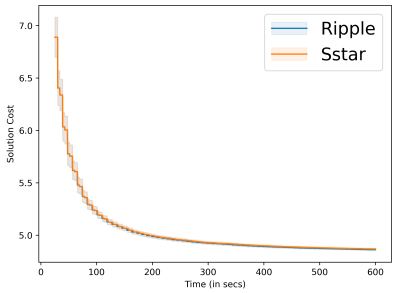
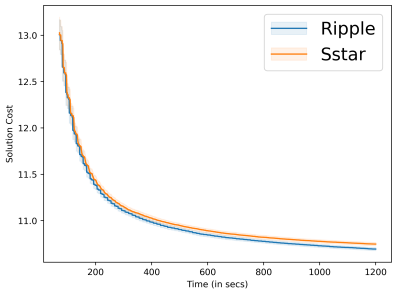
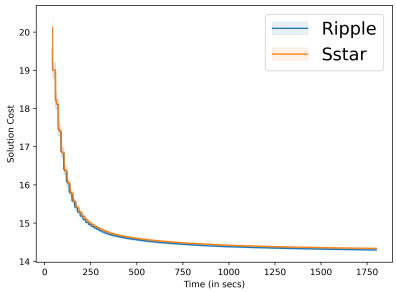
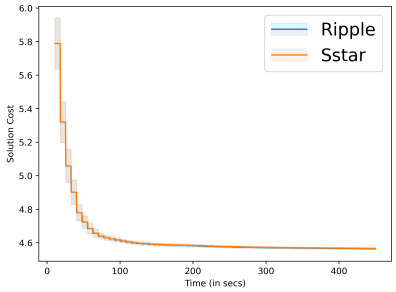
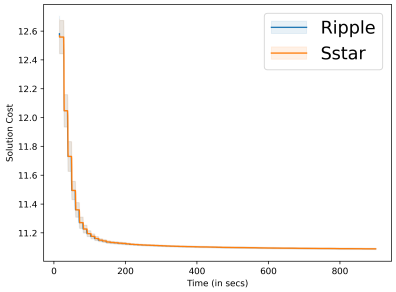
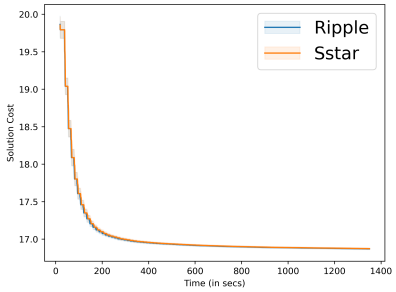
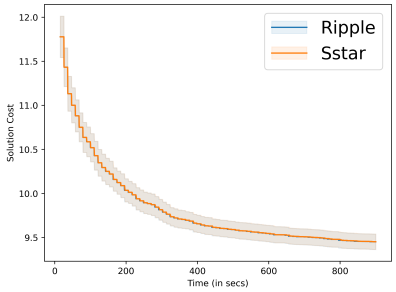
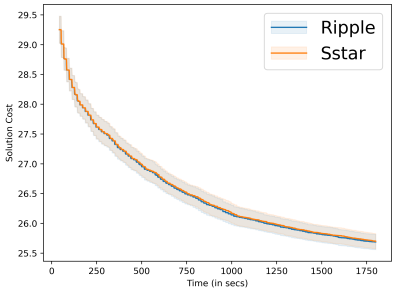
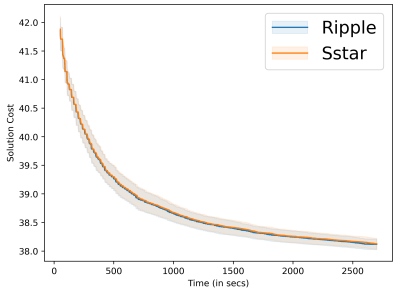
Env.	10 Terminals	30 Terminals	50 Terminals
ABSTRACT			
UH $\mathbb{R}^4$			
CO $\mathbb{R}^4$			
CO $\mathbb{R}^8$			

Table 6: Comparison of the performance of IST* when `Ripple` is used against S* on environments not shown in the main paper. The dark line represents the mean solution cost with the thick region being the 99% confidence interval about the mean.