UNIVERSITY OF TORONTO

Institute for Aerospace Studies (UTIAS)

AER 1516 Project Report
FMT* with Bi-directional Searching Ability

Team 7 Members:

Rongze Ma:1007546425

Jingwei Zhang: 1003326616

Xinlin (Cynric) Li: 1003151705

1. Introduction

1.1 Sampling-Based Algorithms

Motion planning refers to the computation of paths that lead the robots or systems to reach a set of goal configurations around obstacles while optimizing the objective function [1]. The probabilistic sampling-based algorithm has become a successful method in the robotics field in terms of the motion planning problem of the robots in the high-dimensional and expensive collision-checking environment. The main concept behind the sampling-based algorithm is to construct the configuration space explicitly while completing a search that can sample the whole configuration space with a sampling plan [1]. Technically speaking, the sampling-based algorithms can be sorted into two categories. The first category is called multi-query. It essentially constructs a roadmap and allows users to solve problems regarding multiple initial-state or goal-state queries. The second category is called single-query, where one initial or goal state is given and eventually the algorithm has to either find a solution or report failure [1]. The sampling-based algorithm usually provides probabilistic completeness guarantees. In other words, the probability that the planner fails to return a solution, if one exists, decays to zero as the number of samples approaches infinitely [1].

1.2 FMT* & Project Objective

The Fast Marching Tree* algorithm, a sampling-based method, is designed based on some existing state-of-the-art asymptotically-optimal algorithms to reduce the number of collision checks of obstacles in complex configuration spaces, and it is a useful technique to target the appropriate sample points to update and form the path step by step in an outward direction so that the FMT* algorithm will never backtrack those sample points that have been evaluated before. The FMT* performs efficiently in high-dimensional environments occupied with obstacles and is reminiscent of the FMT. The FMT* is capable of performing a forward dynamic programming recursion on a predetermined number of probabilistically-drawn samples in configuration spaces [1]. According to Lucas Janson [1], the FMT* has three unique features. First, the FMT* is tailored to disk-connected graphs, where two neighbors are considered and connected if the distance between two neighbors is below a self-defined connection radius value. The second feature is that the FMT* constructs graphs and searches graphs simultaneously. Lastly, it lazily ignores the presence of obstacles for the purpose of evaluating the immediate cost in the dynamic programming recursion [1]. As known from its name, the algorithm forms a tree of trajectories in the configuration space. The objective of this project is to extend the FMT* to BFMT*, namely the fast marching tree* algorithm with bi-directional searching ability.

2. Background

2.1 FMT*

There are numerous sampling-based powerful planners available in robotics applications, including but not limited to Probabilistic Roadmap algorithm (PRM), Rapidly Exploring Random Tree (RRT), Expansive Space Tree algorithm (EST), etc. However, the Fast Marching Tree (FMT*) algorithm presented by Lucas Jason and fellows from Stanford University [1] uses an innovative method that is conceptually different from all of the above. They have proved that FMT* can converge much faster than other state-of-the-art like PRM* and RRT*, especially in high-dimensional configuration spaces including a high-dimensional maze and alpha puzzle. Besides, FMT* only requires collision checks for locally optimal connections, which makes it computationally cheaper than others.

2.2 Bi-directional Search Algorithm

In 1971, Ira Pohl first designed and implemented a bi-directional heuristic search algorithm [2] which later splits into three categories normally: Front-to-Front, Front-to-Back, and Perimeter Search. These three categories are different from each other by the function used for calculating the heuristic. Front-to-back type of bi-directional search is the most actively researched and utilized among the three categories. However, compared to Front-to-Back, the Front-to-Front approach is more computationally expensive. As stated by Michael Luby and Prabhakar [3], bi-directional search aids boost the convergence rate of planning algorithms. With this benefit of performance improvement, many have merged the bi-directional search strategy with some outstanding planners. RRT-connect is a good example that combines the bi-directional method and RRT planner.

3. Literature Review

3.1 Literature Review of FMT*

3.1.1 Abstract of the FMT Paper*

The paper 'Fast Marching Tree: a Fast Marching Sampling-Based Method for Optimal Motion Planning in Many Dimensions*' [1] gives a novel and complex sampling-based algorithm regarding the motion planning field, called Fast Marching Tree* algorithm, namely FMT*. In this paper, authors have proved the FMT* algorithm to be asymptotically optimal and it is able to converge to an optimal solution faster than some other state-of-the-art counterparts, namely probabilistic roadmap algorithm (PRM*) and rapidly exploring random trees algorithm (RRT*). The paper stated that the FMT* algorithm is able to perform a dynamic programming recursion on a predetermined number of probabilistic-drawn samples to grow a tree of paths, which moves outward in cost-to-arrive space. In other words, the FMT* algorithm is the combination of both RRT* and PRM*. In addition, it is also reminiscent of the Fast Marching Method for the solution of Eikonal equations. As FMT* is analyzed under the notion of convergence in probability, this extra mathematical flexibility allows the authors to work out the convergence rate bound of order $O(n^{-1/d+\rho})$, where n represents sampled points, d represents the dimension of the configuration space, and ρ is a small arbitrary number.

3.1.2 Results Presented in the FMT Paper*

Asymptotic optimality for a number of variations of FMT* is demonstrated under the conditions when the configuration space is sampled non-uniformly, when the cost is not arc length, and when the connections are made based on the number of nearest neighbors instead of a fixed connection radius. Based on the numerical experiments presented in this paper, it is confirmed that the theoretical and heuristic arguments of FMT* are true by returning substantially better solutions than either PRM* or RRT*, especially in high-dimensional configuration spaces and in the environment where collision-checking is expensive.

As mentioned above, the FMT* algorithm expands in a tree-shape path as shown in Figure 1, where a total of 2500 sample points are implemented in a 2D environment.

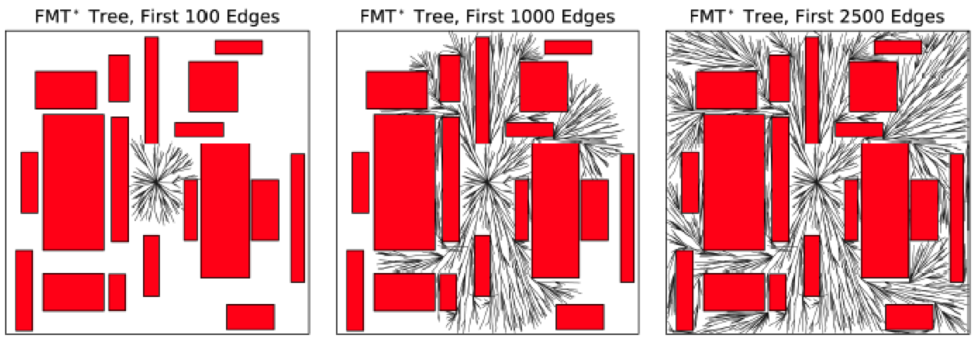

FMT* Tree, First 100 Edges
FMT* Tree, First 1000 Edges
FMT* Tree, First 2500 Edges

Figure 1 - The FMT algorithm implementation with 2500 samples in 2D environment [1].*

Followed by the explanation of the FMT* algorithm features and its properties, authors compared the performance of the FMT*, RRT*, and PRM* in several environmental setups and configuration spaces including $SE(2)$ and $SE(3)$. Figure 2 indicates those configuration spaces.

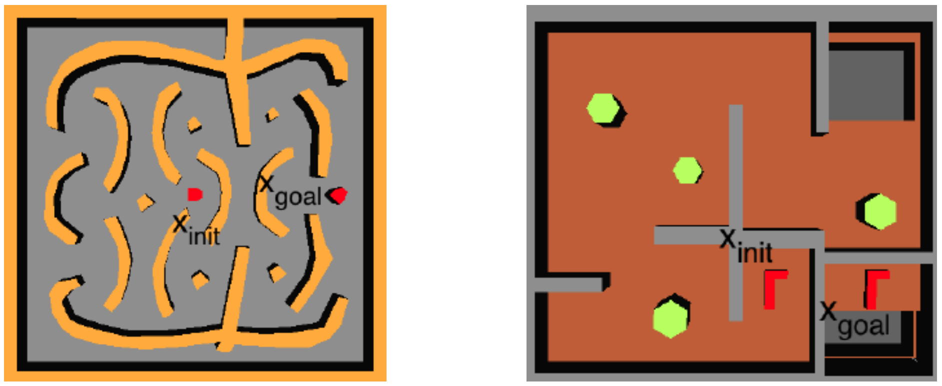


Figure 2 - $SE(2)$ and $SE(3)$ rigid body configuration spaces [1].

As shown in Figure 3, in a $SE(2)$ configuration space, FMT* algorithm reaches given solution qualities faster than PRM* algorithm and RRT* algorithm by factors of about 10 and 2 respectively. All the algorithms achieves 100% success rate in a short period of time, however, FMT* algorithm takes the least time.

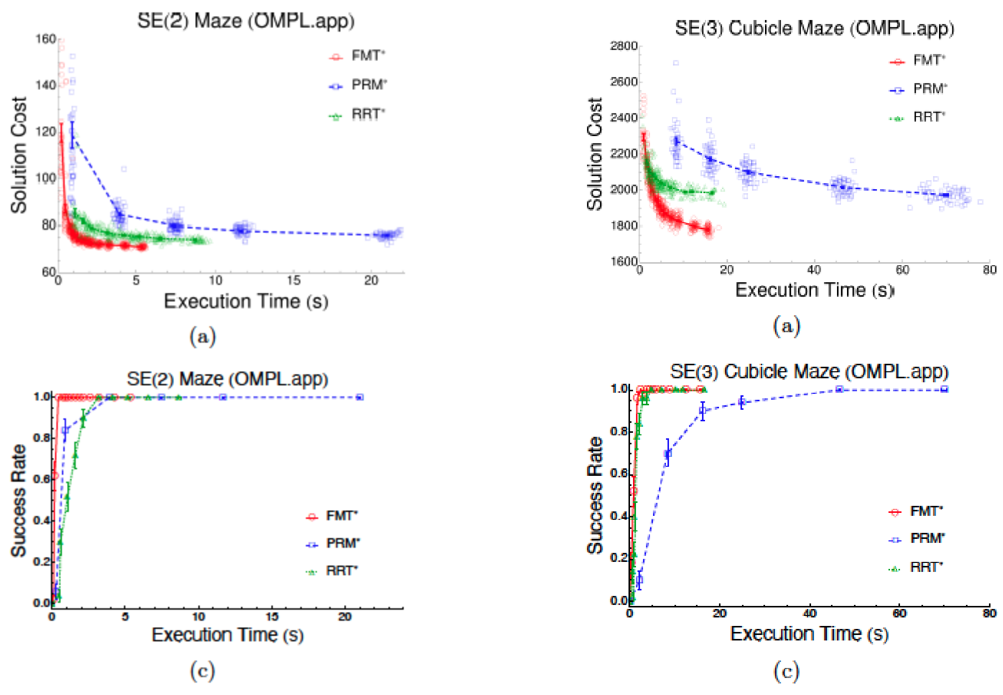


Figure 3 - Results of three algorithms in $SE(2)$ and $SE(3)$ [1].

Figure 3 also includes the simulation results for a three-dimensional maze. The solution cost vs. execution time plot illustrates the FMT* algorithm performs better than PRM* and RRT* since the FMT* achieves an average solution quality in less than five seconds, whereas it costs around seventy and twenty seconds for PRM* and RRT* respectively. In addition to the 2D and 3D mazes, authors also performed experiments in higher dimensional environments such as 5D and 7D recursive mazes. All results have proven that the FMT* is significantly faster and outperforms its state-of-the-art algorithms in terms of solving a wide range of complex and challenging motion planning problems. This paper also leaves some important extensions for further research.

Overall, this paper is well-structured and easy to follow. It contains solid mathematical proof to back up all the concepts behind the FMT* algorithm. Besides, the FMT* algorithm has been proved to outperform those state-of-the-art sampling-based algorithms, namely PRM* and RRT*, in a variety of complicated environments. Last but not least, this paper has opened up many extensions for further research.

3.2 Literature Review of Bi-directional FMT*

3.2.1 Abstract of the Bi-directional FMT Paper*

Since the team aims to implement the bi-directional search algorithm into the FMT* algorithm, apart from the FMT* paper reviewed above, another paper named 'An Asymptotically-Optimal Sampling-Based Algorithm for Bi-directional Motion Planning' [4] is also reviewed as a reference for the team to complete the bi-directional FMT*. According to the paper, few results are available that combine both asymptotic optimality and bidirectional-search features and merge into existing optimal planners. Filling the gap then becomes the objective of this paper.

Bi-directional FMT* refers to the condition where the FMT* essentially performs the bi-directional dynamic programming recursion over a set of probabilistically-drawn samples in the free configuration space [4]. The convergence rate of a planning algorithm is a parameter that represents how quickly the algorithm approaches the limit. According to the written paper, the bi-directional FMT* can increase the rate of convergence significantly to accelerate the motion-planning-related query. The bi-directional FMT* propagates two wavefronts in the configuration space at the same time as the algorithm starts to function. Same as the FMT*, the bi-directional FMT* algorithm is characterized by those features mentioned in the FMT* paper. However, instead of having only one-source dynamic programming recursion, the bi-directional FMT* generates a pair of search graphs in both cost-to-come spaces from the initial configuration and cost-to-go spaces from the goal configuration, and they are called forward tree and backward tree respectively [4]. Since the BFMT* expands in two directions with a larger rate of convergence, the time it takes to finish BFMT* will be smaller than the for FMT*.

3.2.2 Results Presented in the Bi-directional FMT Paper*

Even though this paper has implemented the bi-directional search algorithm into FMT* and obtained desired results, the code for all experiments was written in C++ and can only be implemented on a Linux-operated PC, which makes this implementation limited. Some results from the paper will be presented in this section and our implementations will be discussed later in the Implementation section. According to Figure 4, numerical experiments in $SE(2)$ and $SE(3)$ present the fact that BFMT* tends to be an optimal solution at least as fast as some state-of-the-art algorithms, and even significantly faster in some cases.

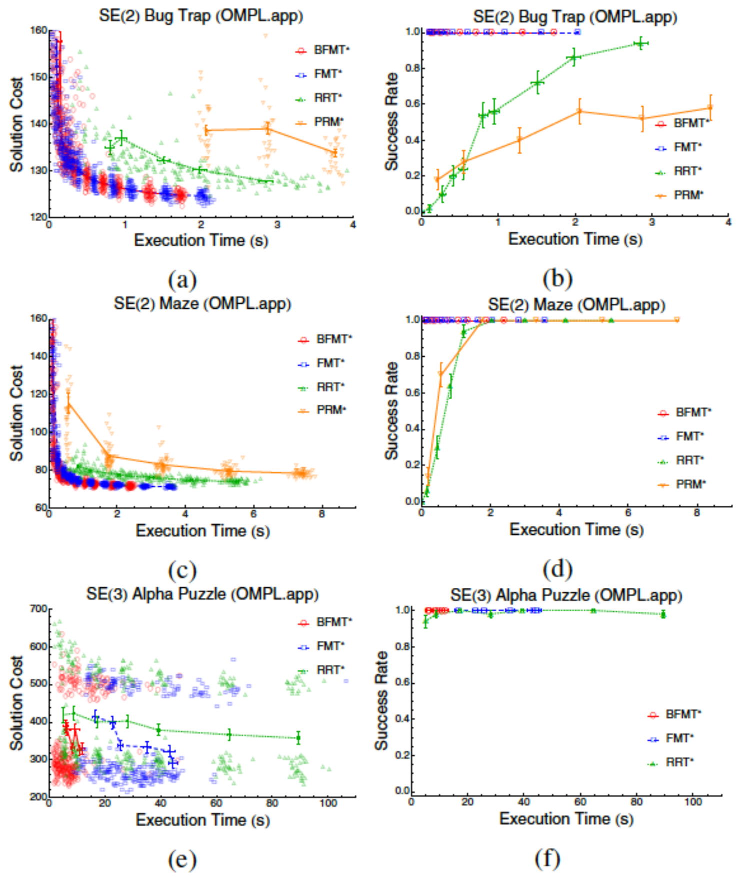


Figure 4 - Simulation results for the three OMPL scenarios [4].

4. Implementation

4.1 BFMT* Algorithm

Generally, the BFMT* is adapted based on FMT*, a pair of trees is generated where one forms in cost-to-come space from the start node, the forward tree, and the other forms in cost-to-go space from the goal node, the backward tree.

Below we have provided the pseudocode for our BFMT* algorithm. This pseudocode is formed based on the pseudocode in [1] and [5] with a few modifications which will be explained later. To better understand the pseudocode for the BFMT*, we will start explaining the algorithm by introducing a series of notations. Let V be a dictionary in which set $V[0]$ contains the initial state x_{init} and a set of n randomly sampled nodes in the obstacle-free region x_{free} and set $V[1]$ contains the goal node x_{goal} and the same set of n randomly sampled nodes. V_{open}, $V_{unvisited}$ and V_{close} are exclusive subsets of V (note that in the actual code, V_{open}, $V_{unvisited}$ and V_{close} are dictionaries just like V that store nodes for forward and backward trees separately). V_{open} contains samples that have already been added to the tree and are still candidates for further connections. $V_{unvisited}$, however, contains the samples that have not yet been considered for the growth of tree path. V_{close} is the set for samples added to the tree and no longer considered for tree expansion. The search radius r_n is calculated using $r_n = 40 * \sqrt{\frac{log(n)}{n}}$, this equation is adapted from the FMT* paper [1].

Let's now look at the details of Algorithm 1, during the initialization stage (line 1 - line 3), a new node called x_{meet} is initialized as an empty set, this variable will contain the lowest-cost candidate node for the two trees to join. x_{init} is then added to the set $V_{open}[0]$ and all other randomly sampled nodes are added to $V_{unvisited}[0]$ for the forward tree. Likewise, x_{goal} is added to the set $V_{open}[1]$ and all other nodes are added to $V_{unvisited}[1]$ for the backward tree. A global variable tree is initialized to 0 which represents the forward tree (if tree equals to 1, that represents backward tree) as we will be expanding the forward tree first.

Before we start expanding the forward tree, node z, the node to expand, is first initialized to x_{init} as shown on line 4. The while loop at line 5 indicates when the expansion can stop, that is when both the forward and backward trees have propagated sufficiently far through each other that no better solution can be discovered, in other words, this occurs when both trees have selected the same lowest-cost node to expand next. This is the first change we have made from the pseudocode shown in [5], in [5], the stopping criteria is when x_{meet} is not an empty set anymore, in other words the algorithm will return the first available path discovered which is normally not the lowest cost path.

Inside the while loop, the function ExpandTreeFromNode(z, r_n) is called. This function expands the current tree the same way as FMT* does except the additional steps to update x_{meet}. The pseudocode for this function is shown in Algorithm 2. First, all the nodes within search radius r_n of node z from $V_{unvisited}$ are found and denoted as nodes x (line 3). For each of node x, it's neighbours from V_{open} are found and denoted as nodes y (line 4). Among nodes y, the one that has the lowest cost (shortest distances from y to x) will be selected (line 5). If this locally-optimal connection is collision-free, then this connection is added to the current tree of paths (line 6 - line 7). Node x is now considered as successfully connected and should be removed from $V_{unvisited}$ to prevent revisiting (line 8). This node x will also be added to V_{open} to be considered as a candidate for further expansion (line 9). x_{meet} is then updated if the total cost (the total cost is the sum of the forward tree cost and the backward tree cost from node x) of the current node x is smaller than the total cost of x_{meet} (line 10 - line 11), this indicates we have found a more optimal node to connect the two trees and x_{meet} will then be updated. Lastly, node z will be removed from V_{open} and added to V_{closed} (line 12).

Now going back to Algorithm 1 after expanding the node z, if V_{open} of both trees are empty, that means, the algorithm has failed to find a path and there are no other suitable nodes to expand, the algorithm will stop and report failure (line 7 - line 8), if the companion tree has an empty V_{open}, then we will continue expanding the current tree by selecting the lowest cost node from V_{open} as the next expanding node (line 9 - line 10), if the companion tree's V_{open} is not empty, the lowest cost node from V_{open} of the companion tree will be selected as the next expanding node. The tree variable will be toggled and the companion tree will be expanded in the next loop (line 11 - line 13). In [5] however, the BFMT* algorithm sample a new node in the obstacle-free region if the companion tree has an empty V_{open} and trying to balance the two trees by expanding the smaller tree. We think this approach is not effective for the maps we will experiment with, in which the start node will be placed inside a narrow corridor and the only situation when V_{open} is empty before a solution is found is when there are not enough sampled nodes inside the narrow corridor for the tree to expand. The probability of the new sampled node to be inside the narrow corridor is extremely low, therefore we decide to omit this part of the algorithm. Also instead of expanding the smaller tree, we have modified ours to expand the tree regardless of the tree size, this could result in a situation where BFMT* becomes FMT* when one of the trees has an empty V_{open}, we will briefly discuss such a situation in the results section.

Both FMT* and BFMT* algorithms are implemented in python, we have used the implementation of FMT* in [6] and the BFMT* implementation is built upon the python code for FMT*. Our code can be found here.

```
Algorithm 1 The Bi-direction Fast Marching Tree Algorithm (BFMT*)
Require: sample two sets {V[0], V[1]}, V[0] for forward tree comprising $x_{init}$ and n samples in
$\chi_{free}$, V[1] for backward tree comprising $x_{goal}$ and n samples in $\chi_{free}$; Search radius $r_n$
1.  Initialize $x_{meet}$ as an empty set
2.  Place $x_{init}$ in $V_{open}[0]$ and all other samples in $V_{unvisited}[0]$;  Place $x_{goal}$ in $V_{open}[1]$ and all
    other samples in $V_{unvisited}[1]$
3.  Initialize tree as 0 (forward tree)
4.  initialize z with root node $x_{init}$
5.  While z is not in companion tree's $V_{closed}$
6.      Call ExpandTreeFromNode(z, $r_n$)
7.      If both $V_{open}[0]$ and $V_{open}[1]$ are empty
8.          Return Failure
9.      If the companion tree has an empty $V_{open}$
10.         Find lowest-cost node z in $V_{open}$ of the current tree
11.     Else
12.         Find lowest-cost node z in $V_{open}$ of the companion tree
13.     Swap tree
14. Return success and optimal path
```

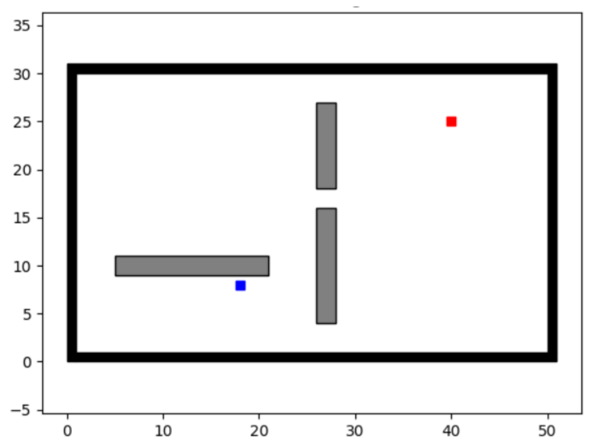
```
Algorithm 2 Fast Marching Tree Expansion Step
1.  Function ExpandTreeFromNode(z, $r_n$)
2.      Initialize new $V_{open}$ set
3.      For each of z's neighbours x in $V_{unvisited}$ of the current tree
4.          Find neighbours nodes y in $V_{open}$ of the current tree
5.          Find locally-optimal one-step connection to x from among nodes y
6.          If that connection is collision-free
7.              Add edge to current tree of paths
8.              Remove successfully connected nodes x from $V_{unvisited}$ of current tree
9.              Add those nodes x to $V_{open}$
10.             If sum of x cost in both tree is smaller than sum of $x_{meet}$ cost in both tree
11.                 Update $x_{meet}$ with x
12.     Remove z from $V_{open}$ and add it to $V_{closed}$
```

Figure 5 - Bi-directional Fast Marching Tree Algorithm Details.*

4.2 Simulation Results

4.2.1 Environment Setup

Two different maps are designed to test the BFMT* algorithm, the first map is simpler with just one narrow passage whereas, in the second map, the start node has been placed inside a narrow corridor. Previously, we have seen that FMT* struggles to find a path when the start/goal node is placed inside a narrow corridor at low sample counts, therefore we would like to see if BFMT* will have a higher success rate under the same condition.


(a)

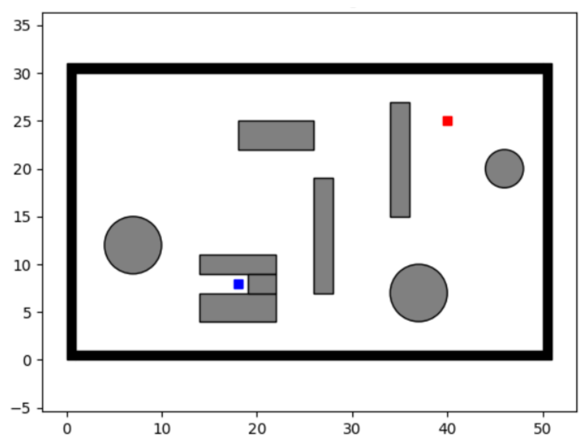

(b)

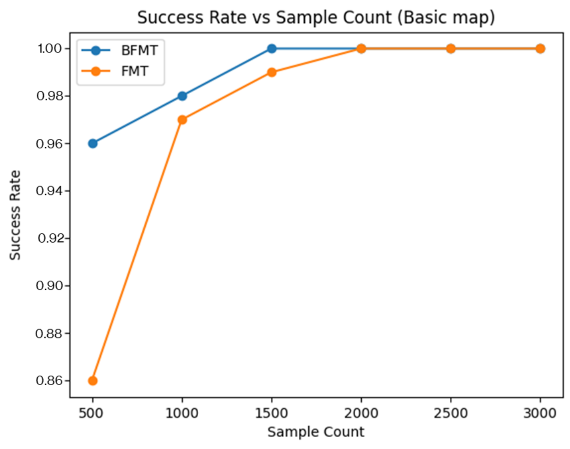
Figure 6 - Two testing configuration spaces with different obstacles.

For the experiment setups, we have set the sample counts to 500, 1000, 1500, 2000, 2500, and 3000, and both FMT* and BFMT* algorithms are performed 100 times under each sample count for each map. During each run, the number of collision checks, the execution time, the solution cost of the final returned path, and the success rate at successfully returning a path are recorded. The mean value of the 100 runs is later used for performance analysis and algorithm comparison.

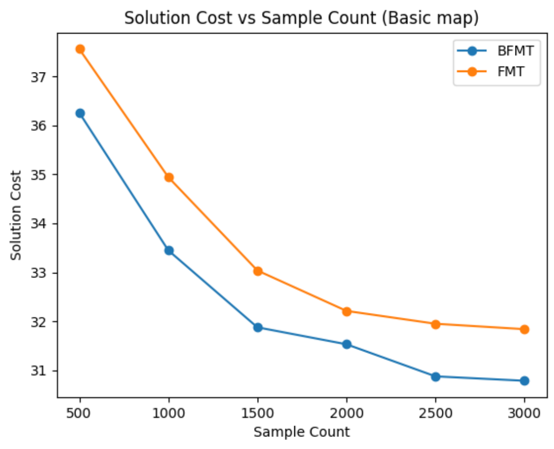
4.2.2 Performance Comparison

Because Map 1 is rather simple, we can see the success rate in Figure 7 (a) of both FMT* and BFMT* are high. BFMT* has a slightly higher success rate at low sample counts than FMT*, 96% success rate compares to an 86% success rate at 500 sample counts. BFMT* achieved a 100% success rate at 1500 sample counts whereas FMT* required 2000 randomly sampled nodes to achieve a 100% success rate.

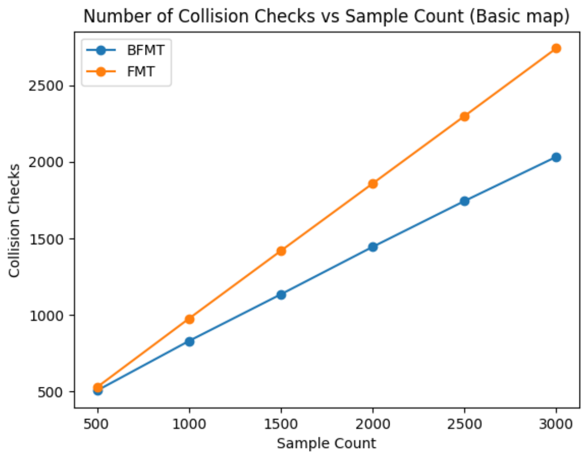
In terms of solution cost of the final returned path Figure 7 (b), it is obvious to see BFMT* can always return a lower-cost solution than FMT* (i.e., a more optimal solution) at all sample counts. Also, we could observe that as sample count increases, the solution cost decreases up to 2500 sample counts. Beyond this point, increasing the sample count won't improve the solution cost by a significant amount.
Figure 7 (c) and Figure 7 (d) shows the number of collision checks and execution time at different sample counts, and have a very similar trend, this is because the number of collision checks the algorithm executes in each run is directly related to the execution time, that is, the more collision checks the algorithms need to execute, the longer the run will take. At low sample counts, FMT* and BFMT* require almost the same number of collision checks and execution time, as the sample count increases, this difference also increases with BFMT* always being the one that executes fewer collisions checks and requires shorter execution time. This indicates that for map 1, BFMT* run more efficiently than FMT* especially when the sample count is high.



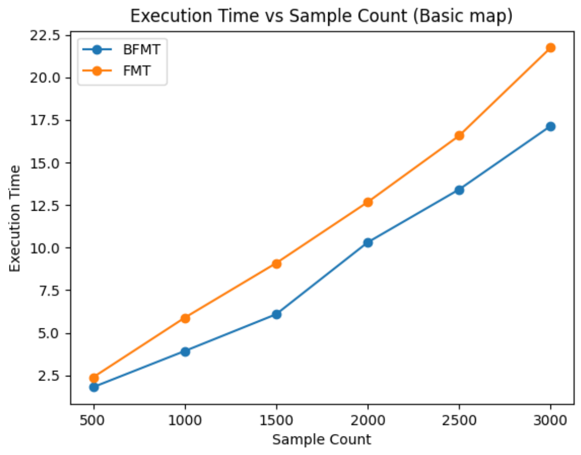
(a)



(b)



(c)

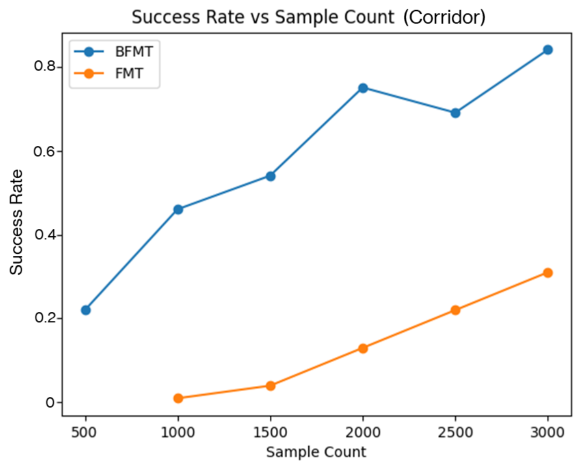


(d)

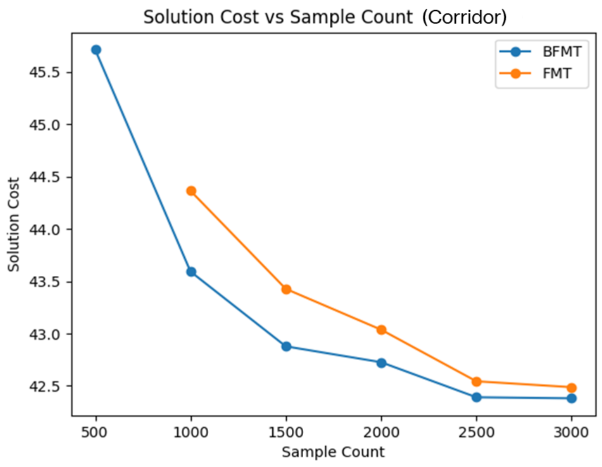
Figure 7 - Simulation results between FMT and BFMT* in terms of the first configuration space.*

For map 2 in which the start node has been placed inside a narrow corridor, both FMT* and BFMT* are struggling to find a path as evident by the low success rate at a low sample count. With 500 randomly sampled nodes, FMT* has a 0% success rate while BFMT* has a 20% success rate. As the sample count increases, the success rate of BFMT* increases to 80% at 3000 sample count but FMT* still have a very low success rate around 30%. Looking at the solution cost plot, similar to the results for map 1, BFMT* returns more optimal solutions at all sample counts than FMT*.

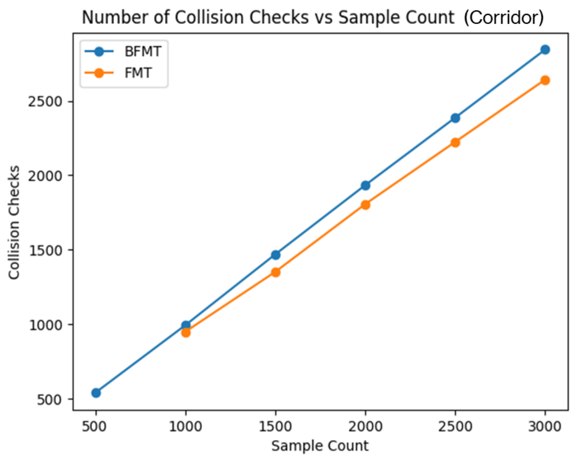
The number of collision checks and execution time, however, are different from what we have seen for map 1, BFMT* now requires more collision checks in each run on average and the execution time is longer than FMT*. We think the reason is that for a simpler map, BFMT* doesn't need to expand too far from the start and goal nodes to find a path, that is, when the trees are still small, therefore, the algorithm requires fewer connections and in turn results in fewer collision checks before finding a path. Conversely, for map 2, both algorithms now need more randomly sampled nodes to expand out the narrow corridor and more sample counts can potentially add more branches to the trees i.e., a wider tree. Because BFMT* has two trees, we suspect it has more branches to expand in total than FMT*, which results in more collision checks and longer execution time.

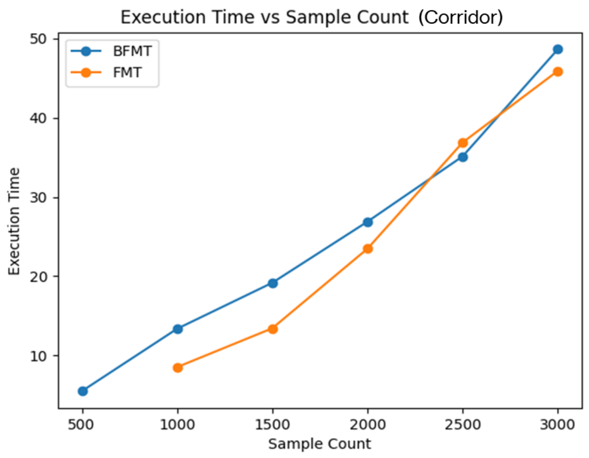


(a)



(b)





(c)

Figure 8 - Simulation results between FMT and BFMT* in terms of the second configuration space.*

The plots below illustrate how the trees of the FMT* and BFMT* grow for both maps at 1000 sample counts for map 1 (Figure 9 (a) and (b)) and 3000 sample counts for map 2 (Figure 9 (c) and (d)).

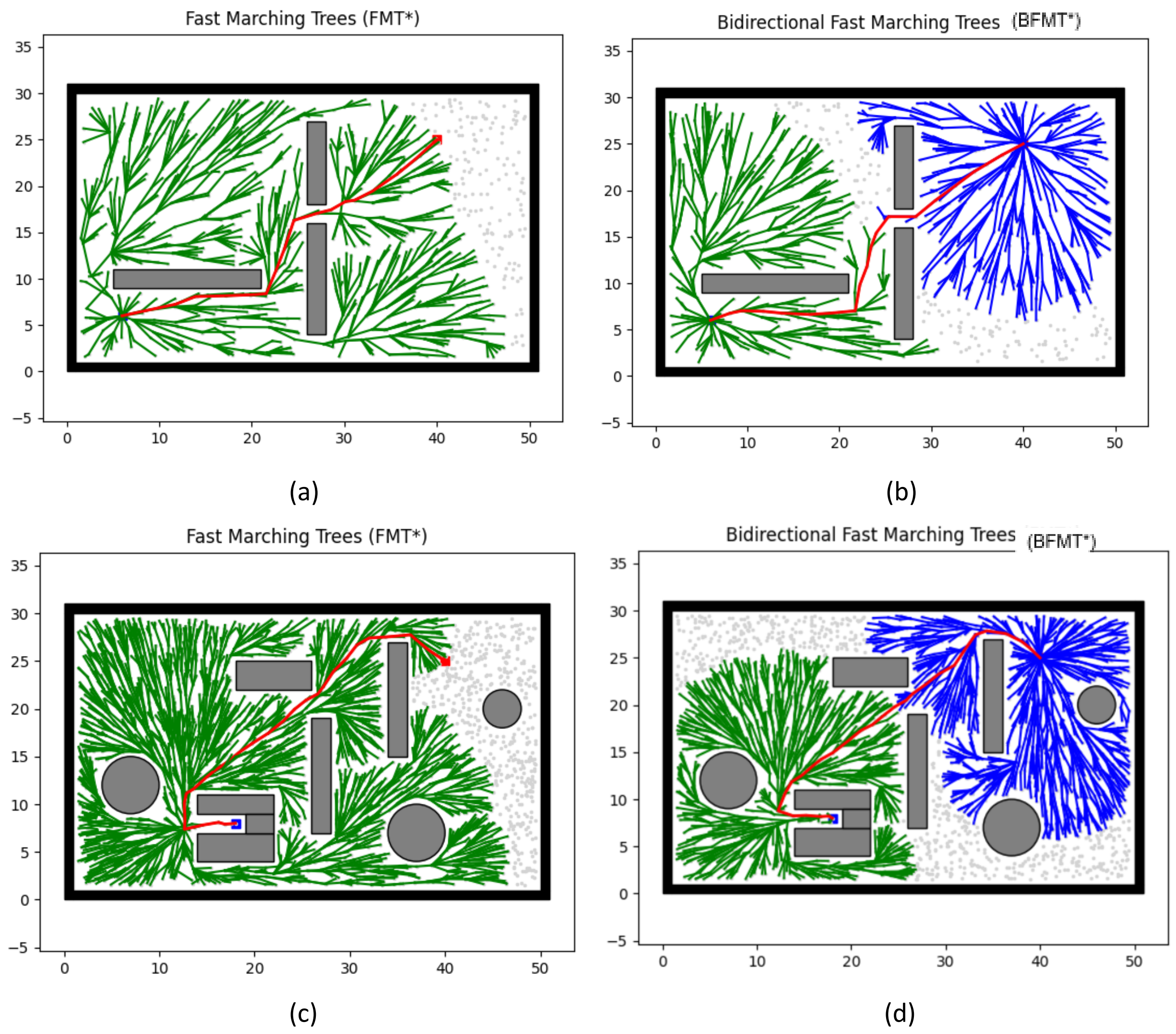


Figure 9 - Plots showing how the trees grow in different configuration spaces.

For map 2, when the sample count is low, BFMT* sometimes turns into a reversed FMT* as shown in the plot below. This is when the start node is inside a narrow corridor and the forward tree cannot find a valid node in the neighbour to expand out which results in an empty V_{open} set, the backward tree however, is able to expand. Our BFMT* implementation encourages the trees to continue to expand unless both trees have an empty V_{open}. As a result, the backward tree continues to expand until it meets the forward tree inside the corridor. This approach increases the success rate of BFMT* at finding a path when either of the start or goal node is stuck inside a narrow corridor.

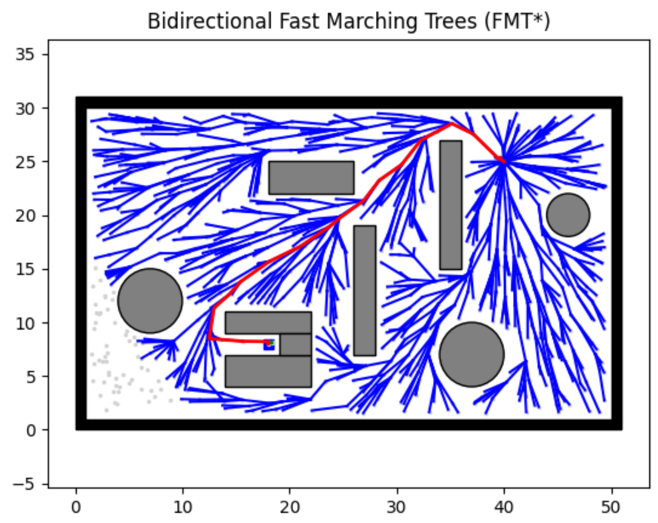


Figure 10 - BFMT turns into a reversed FMT* at 1000 sample counts.*

5. Conclusion

In this report, we have introduced and implemented a sampling-based algorithm named BFMT*. This algorithm uses a combination of the bi-directional search strategy and FMT* algorithm. Experiments were done running 100 times for each set of samples with various sample counts on two SE(2) environments with narrow passages and corridors. Experiment results indicate that the BFMT* algorithm appears to have higher success rate and lower-cost solution at all times when compared with the FMT* algorithm in both self-designed maps. In terms of collision checks and run time which are highly correlated, the BFMT* algorithm generally requires more time than the FMT* algorithm in a less challenging map.

The low success rate at low sample count of our current BFMT* implementation can be potentially improved with an additional non-uniform sampling strategy. A new node can be sampled near the last node of V_{close} when that tree's V_{open} is empty before the algorithm finds a path to help the tree grow outside the narrow corridor. Furthermore, we have only tested SE(2) environments due to limited computational power available. Experiments in higher dimensions will help in understanding BFMT* performance better as well as real world implementations.

Reference

[1] L. Janson, E. Schmerling, A. Clark, and M. Pavone, “Fast marching tree: A fast marching sampling-based method for optimal motion planning in many dimensions”, International Journal of Robotics Research, vol. 34, no. 7, pp. 883-921, 2015.

[2] I. Pohl, “Bi-directional and heuristic search in path problems”. Department of Computer Science, Stanford University., 1969, no. 104.

[3] M. Luby and P. Ragde, “A bidirectional shortest-path algorithm with good average-case behavior”, Algorithmica, vol. 4, no. 1-4, pp. 551-567, 1989.

[4] J.Starek, E.Schmerling, L.Janson and M.Pavone. “Bidirectional Fast Marching Trees: An optimal sampling-based algorithm for bidirectional motion planning.” *Technical report*, 2014

[5] J.A.Starek, J.V.Gomez, E. Schmerling, L. Janson, L. Moreno, and M. Pavone, “An asymptotically-optimal sampling-based algorithm for bi-directional motion planning,” in 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2015, pp. 2072–2078.

[6] Alina Kolesnikova. 2021. Path Planning. https://github.com/zhm-real/PathPlanning#papers