3D Planning Algorithm Visualization

Huakun Shen

December 16, 2022

Abstract: Searching algorithm like BFS and DFS are the fundamentals of many topic, including planning and navigation. We live in a 3-dimensional world. In real life, when we need to apply these planning algorithms, they usually work in 3D. It's beneficial to have a visualization of these algorithms in 3D to give us a better understanding of how they work in 3D space. In this project, I built a flexible and easy to use framework for building interactive 3D environment visualization, implemented a few planning algorithms in 3D and visualize the results of them in 3D using my framework.

Keywords: Robots, Planning, RRT, BFS, DFS, A*

1 Introduction

When learning graph algorithms, we usually start from 2D as it's easier to understand. However, we live in a 3D world and the practice of these algorithms is usually in 3D space. I bought a DJI drone (Quadcopter) recently and enjoyed flying it. It brings me views I would never have a chance to see in normal life. Unmanned aerial vehicles have been evolving quickly in recently years, in both civil and military fields. Planning algorithm is an essential for UAVs to navigate autonomously. There are many scenarios an UAV may want to drive autonomously. For example, FPV is one category of drones that allows user to fly through narrow spaces with a first person view. I've seen people flying FPV smoothly in buildings. This requires lots of practice and crashing. Flying within a buildings may be a useful use case when people need to explore something (potentially dangerous) that human cannot reach personally, such as search and rescue in some natural disaster, exploration in caves, etc. In these scenarios, it's helpful if a drone can navigate itself and even drive itself, especially when remote control isn't possible due to harsh environment. It would be helpful an UAV and explore by itself and bring back the data.

3D visualization and simulation is crucial for developing these planning algorithms as visualization gives us better intuition of the environment, and simulation gives us cheap cost of failure. I've built a 2D visualiztion and animation of BFS, DFS and A* algorithm previously https://huakunshen.github.io/GraphSearchVisualizer/ when I was learning these algorithm for the first time, because I found imagining these algorithm non-intuitive, and having such visualization and animation is beneficial for understanding. This visualizer supports animation and even debugger-like feature allowing user to step through the program step by step, giving students a even better understanding. See figure 1 for how it looks like.

In this project, I took the idea further to elevate to the 3rd dimension. I built a framework for random 3D environment/obstacle generation, interative 3D visualization, and a flexible OOP planning algorithm implementation framework supporting both search-based and sample-based planning algorithms.

[1]

[1] Code: https://github.com/HuakunShen/3D-Planning-Algorithm-Visualization

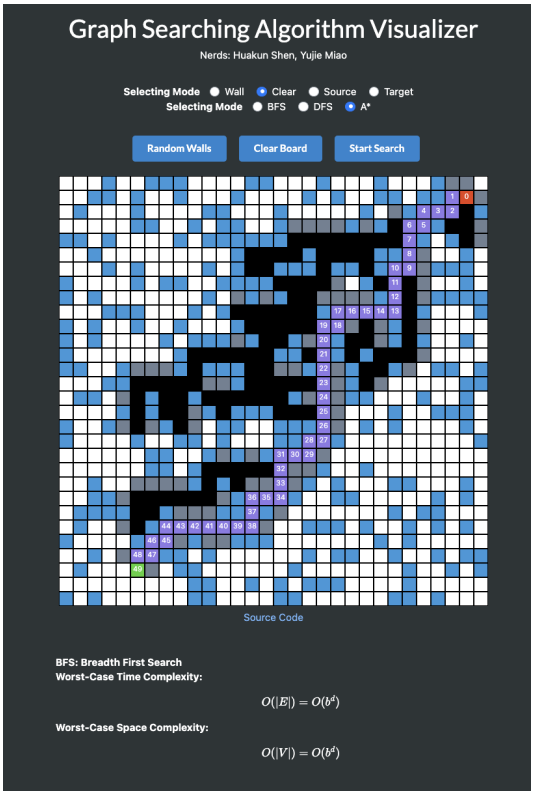


Figure 1: 2D Interactive Visualizer of Graph Searching Algorithm

2 Method

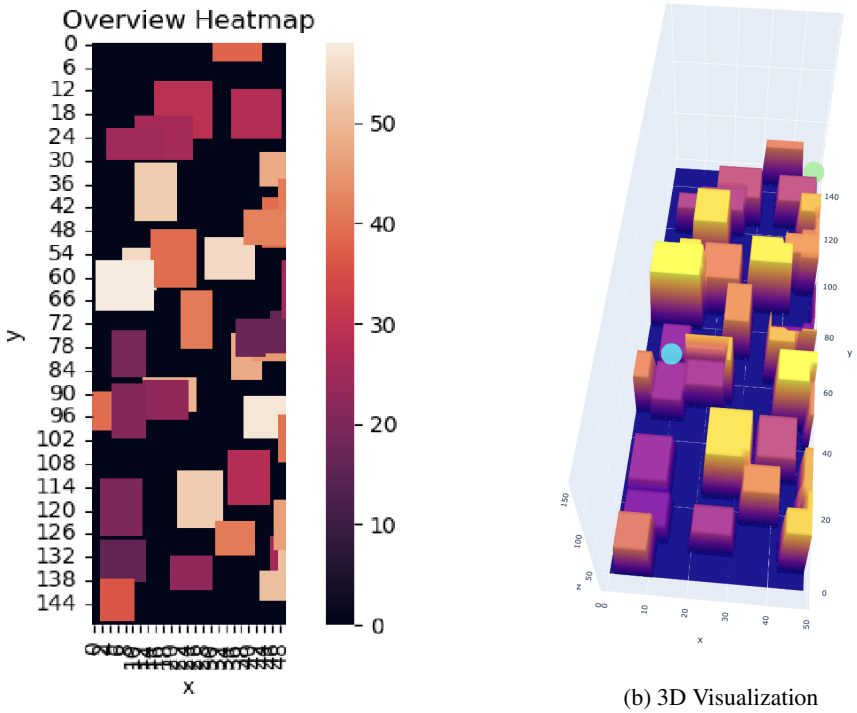
I decided to implement Dijkstra, DFS, A* [1], RRT [2], and RRT*[3] in 3D, and built a visualization of the result. Everything is implemented using Python [4].

First of all, we need an environment to plan with. The environment would be an occupancy grid in 3D, with width, length and height correspondsing to y, x, z axis. Each cell in the occupancy grid indicates whether there is an obstacle. I suppose the environment would be city-like, with buildings and doesn't contain any floating object for simplicity. Although the environment is in 3D, I used a 2D representation to save space. Figure 2a is an example or how the map look like in 2D representation using heatmap. Brighter color means the building is taller. Each value in the 2D matrix correspond to building/obstacles' height. To determine whether a position $\mathbf{A}$ has an obstacle, we can simply compare the z value of $\mathbf{A}$ and the value in the 2D matrix (which is the height of obstacle). By using these height values as surface values (in z-axis), we can plot a 3D visualization in Figure 2b.

A source and target can be drawn uniformly from the entire environment where a position is free. Source and target are plotted as green and blue points in the environment. Figure 3 has an example of how each algorithm is visualized. Resulting path is in red dots. For search based based algorithm like Dijkstra, visited nodes are marked as white dots to show the amount of space it has searched for. For sampling based algorithm like RRT, yellow points and edges are used to illustrate the expanding tree structure.

The 3D visualization is achieved with open source library Plotly [5]. It provides nice-looking visualization as well as interactive interface in browser that allows user to rotate the environment. This is an crucial feature for 3D visualization. Unlike in 2D where we can see everything, objects in 3D can block our views in certain angles. Freedom to rotate the environment gives us better intuition.

Seeds are used to keep environment generation and sampling consistent and reproducible, making it possible to compare multiple algorithm under the same randomly generated environment. This framework has a scenario generator for generating environment automatically based on a given size. Supported size (width) ranges from 5 to 150. Planning package of the framework uses OOP inheritence and template design pattern to make extending more planning algorithms easier. Template abstract class handles environment and visualization. One just need to implement the algorithm and return resulting path, and visualization is automatically generated. Customization to visualization is also possible with dependency injection. The tree-like structure in figure 7d is achieved by injecting a custom edge ploting algorithm to the original visialization algorithm without having to rewrite the entire function.



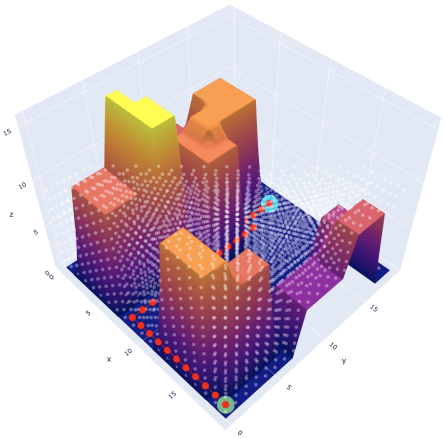
(a) 2D Representation

(b) 3D Visualization

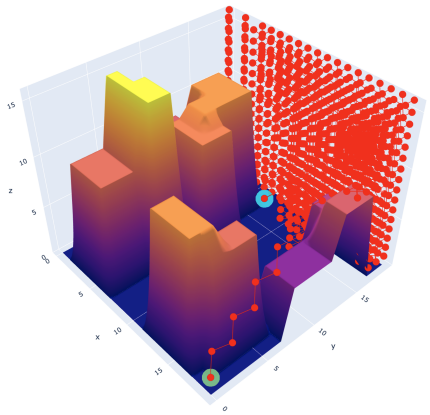
Figure 2: Visualization of Environment

3 Results and Evaluation

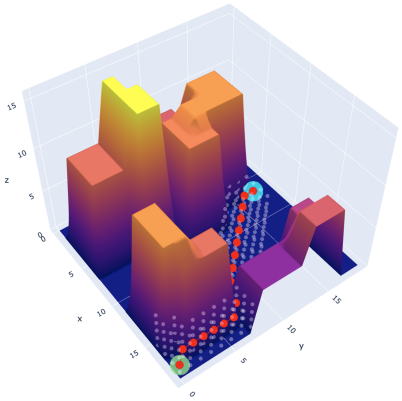
Experiments has been conducted to evaluate the implemented algorithms on runtime and path length based on different map sizes. Success rate is not compared as they all will eventually find a path if there is any given enough time RRT and RRT* will eventually find a path given enough number of iteration.



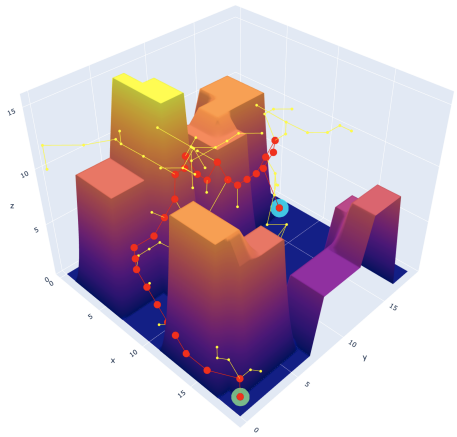
(a) Dijkstra



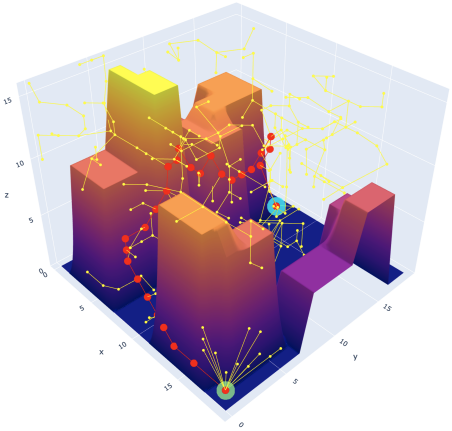
(b) DFS



(c) A*



(d) RRT



(e) RRT*

Figure 3: Planning Result Visualization (Small)

Figure 6 compares time taken to plan and path length against map size. Map used in this experiment all have equal width and length, so I used width as the x-axis. Map width size was taken from this range $\{10, 20, 40, 60, 80, 100\}$. 20 seeds were used for each width to take average.

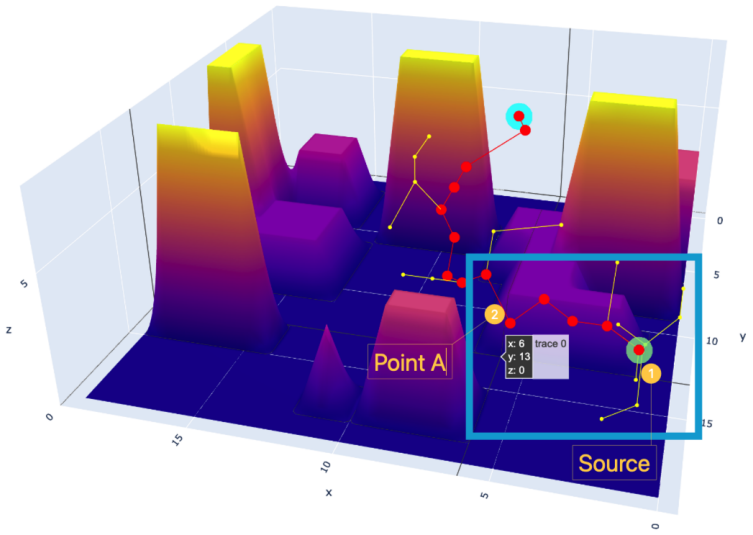
The path length of DFS algorithm is so large that I have to take log of runtime to make it comparable in the same plot. Figure 3b, 8b and 7b has 3 examples of DFS's path in small, medium and large maps. Its path pretty much fill the entire map. Comparing to DFS, other algorithm's path length is on another level.

Figure 6b removes DFS. Dijkstra and A* should theoretically always have optimal path, but due to implementation issue, Dijkstra only doesn't consider corner heighbors as neighborhood, and A* does. So A* can take some shortcuts comparing to Dijkstra.

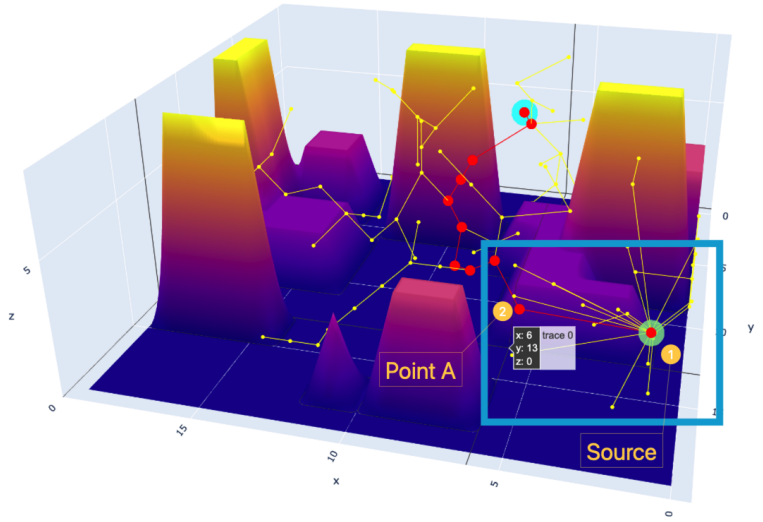
In figure 3a and 3c we can see the explored area in the environment marked as white dots. A* explores significantly less environment and is thus much faster. Figure 8 and 7 in appendix has visualization for medium and large maps, comparing the explored area of A* and Dijkstra. Depending on the environment, Dijkstra can be faster or slower, but it usually cover almost the entire environment. DFS's explored area is covered by its path (which is all red). It explores the entire space in figure 8b.

Figure 6c shows the explored region ratio vs map size. Figure 6d shows the number of explored nodes vs map size. We can see that RRT and RRT* has much smaller explored area comparing to other algorithms.

In figure 4 and figure 5, I compared the path generated by RRT and RRT*. In figure 4, if we pay attention to the blue box, we can see RRT* 4b has emanative edges around the Source node. One of them connects straight to Point A. However in RRT 4a, the path takes 4 steps to reach the same point. Figure 5 is another example in a large map. This is the fundamental difference between RRT and RRT*. RRT* rewires existing graph by connecting node to neighbor that results in shortest path. The more sample points we take, the closer the final path will be to the optimal path. However, since it takes extra time to rewire the neighborhood, RRT* has a much higher runtime than RRT. Without enough iterations, RRT* can have a longer path than RRT.



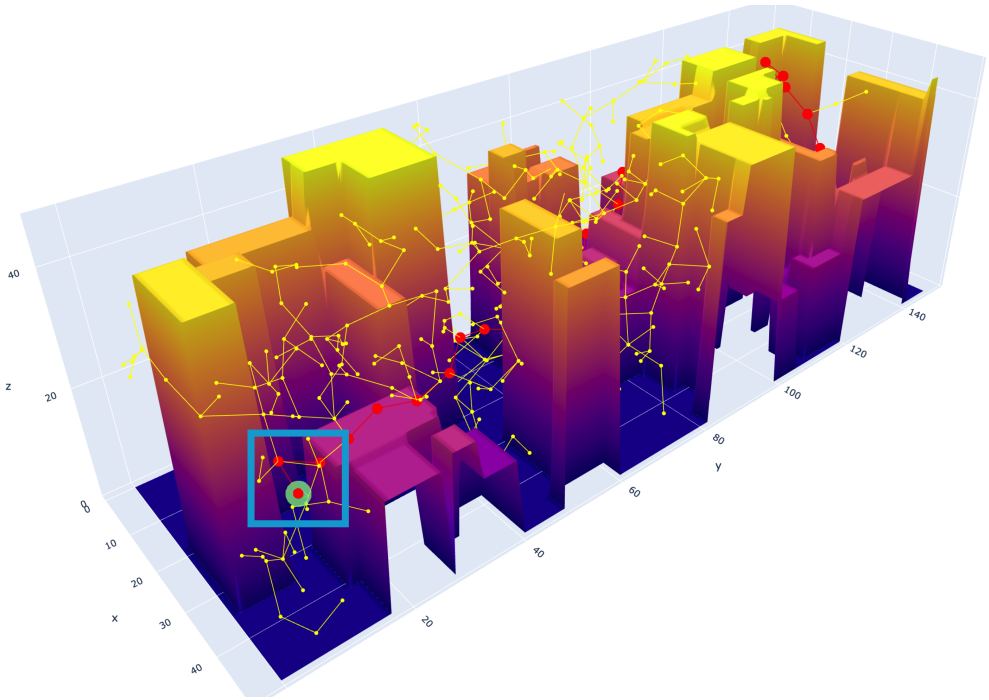
(a) RRT



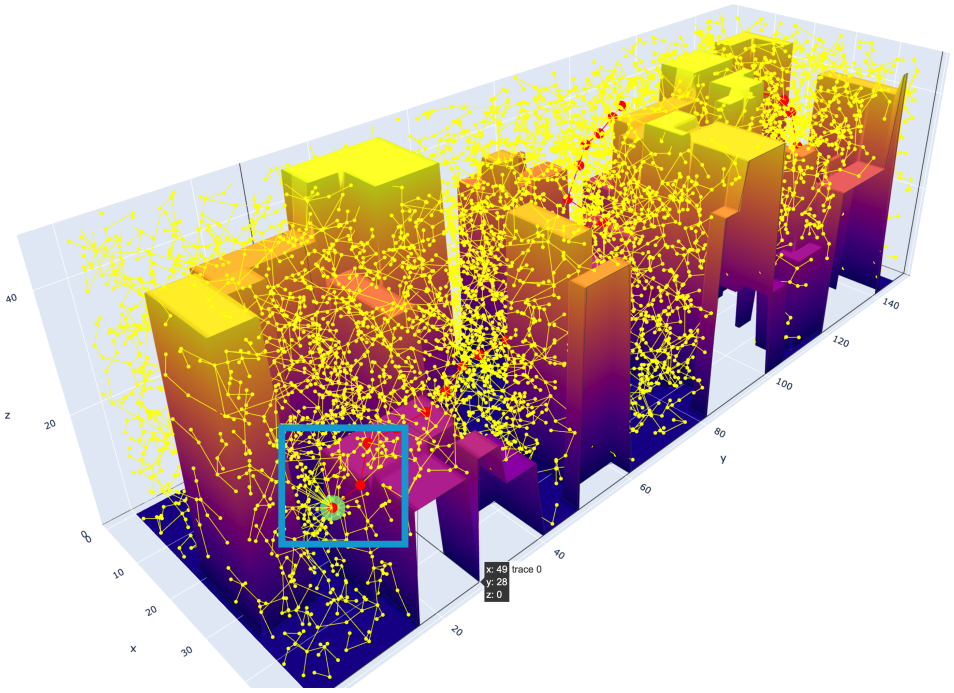
(b) RRT*

Figure 4: RRT vs RRT*

RRT Star should be theoretically asymptotically optimal (i.e. as more samples are taken, the resulting path will be close to the optimal path). In figure 6b, we can observe that RRT Star has lower path length than RRT when map width is 20, and becomes longer afterwards. This is because RRT star takes a lot more time and sampling to achieve optimal path. Given enough iterations, RRT star can indeed shorten the path comparing to RRT as I discussed previously with figure 4 and 5. RRT Star also has runtime bottleneck as environment size grows, especially when environment is in higher dimensions.

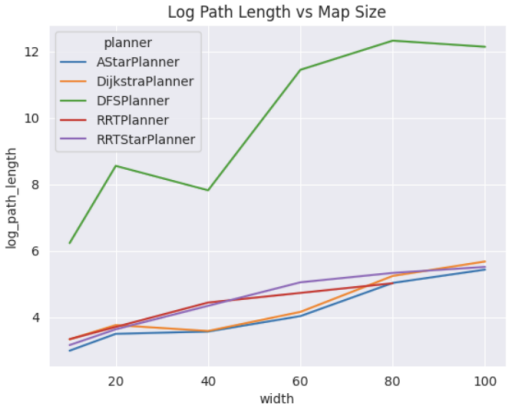


(a) RRT

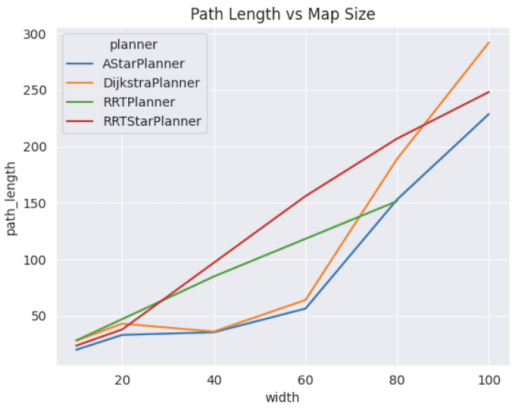


(b) RRT*

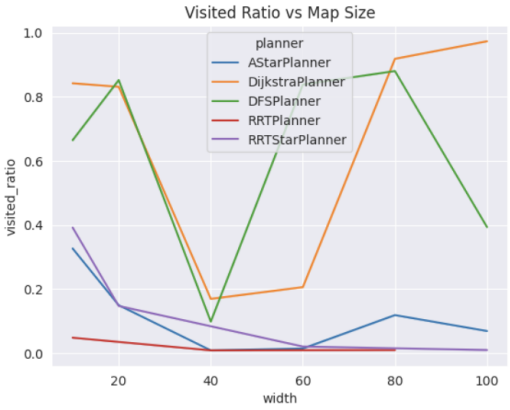
Figure 5: RRT vs RRT*



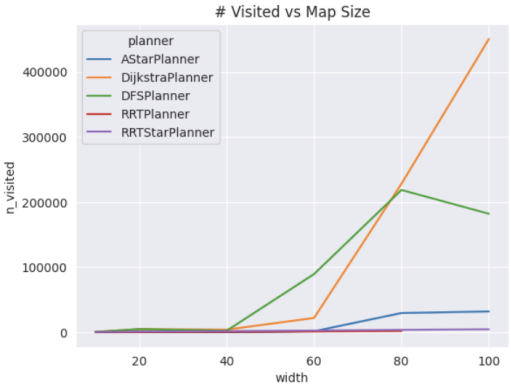
(a) Log Path Length vs Width



(b) Path Length vs Width



(c) Visited Ratio vs Width



(d) Visited Ratio vs Width

(e) Time vs Width

Figure 6: Performance Statistics

4 Limitation

The problem with RRT and RRT* is that the sampling process is performed uniformly from the entire space thus not effective. Most of the time the randomly sampled point is not related to the goal. There has mean many variants of RRT* and other sampling based algorithms trying to improve the performance. Such as Informed RRT* [6], fast marching trees [7], RRT Connect [8], RRT* Smart [9]. Informed RRT* by Gammell et al tries to narrow the sampling space given existing knowledge of the environment, instead of sampling from the entire environment. In each iteration, the sample

space becomes smaller, resulting a lower runtime. RRT Connect tries to expand RRT from both src and target to achieve a faster speed. I've tried to implemented Informed RRT* and RRT Connect in 3D in this project but didn't succeed for now.

Comparing to my previous project (2D visualization and animation) 1, this project lacks animation and debugger-like feature to step algorithm step by step. Having this feature will be amazing and good fore understanding. In the future, I plan to extend this work to support animation with a debugger-like feature allowing user to step through the algorithm with animation instead of directly getting the result.

References

[1] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. doi:10.1109/TSSC.1968.300136.

[2] S. M. LaValle. Rapidly-exploring random trees : a new tool for path planning. *The annual research report*, 1998.

[3] S. Karaman and E. Frazzoli. Incremental sampling-based algorithms for optimal motion planning, 2010. URL `https://arxiv.org/abs/1005.0416`.

[4] G. Van Rossum and F. L. Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[5] P. T. Inc. Collaborative data science, 2015. URL `https://plot.ly`.

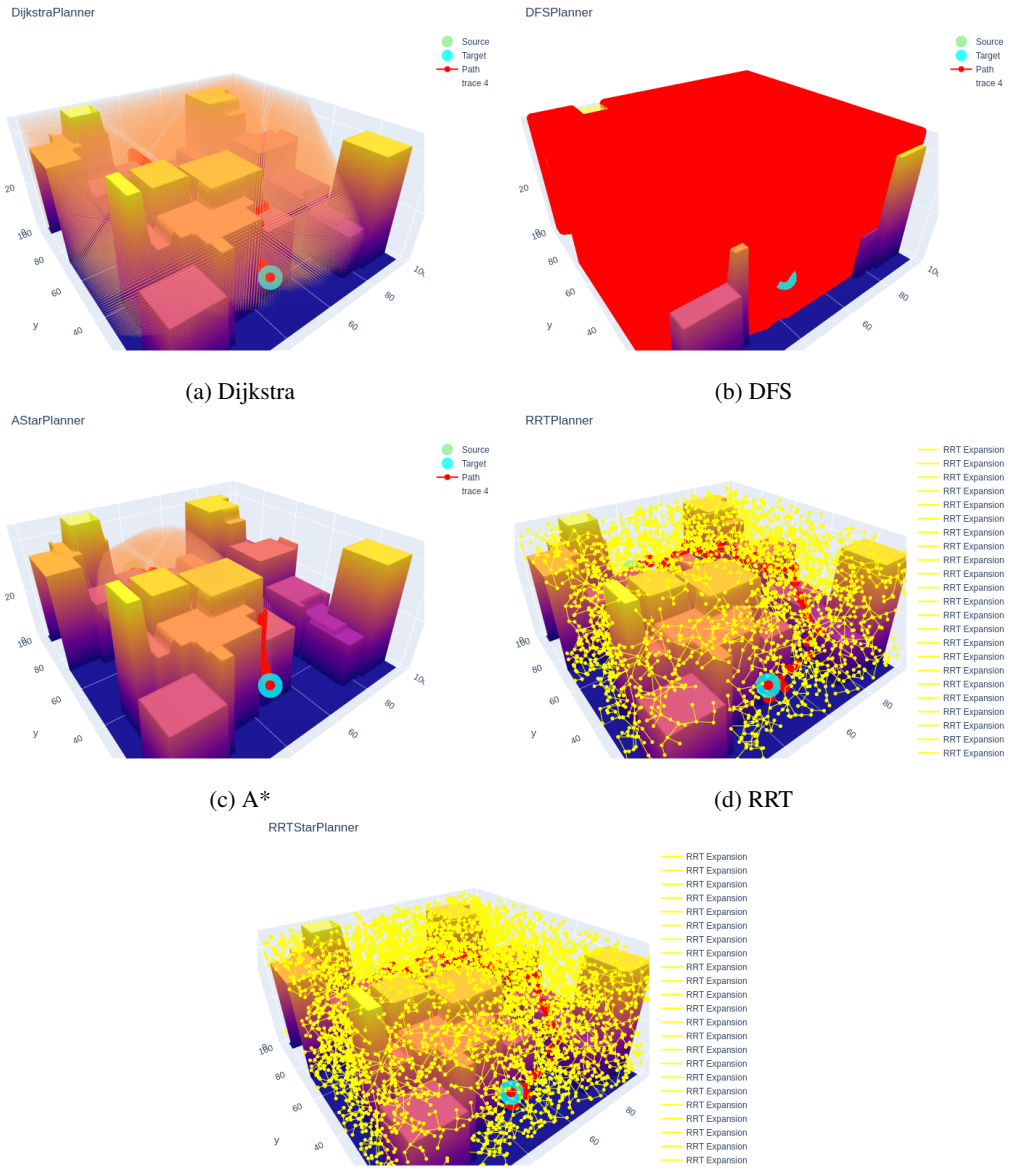
[6] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot. Informed rrt: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2997–3004, 2014. doi:10.1109/IROS.2014.6942976.

[7] 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, 2013. URL `https://arxiv.org/abs/1306.3532`.

[8] J. Kuffner and S. LaValle. 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, pages 995–1001 vol.2, 2000. doi:10.1109/ROBOT.2000.844730.

[9] J. Nasir, F. Islam, U. Malik, Y. Ayaz, O. Hasan, M. Khan, and M. S. Muhammad. Rrt*-smart: A rapid convergence implementation of rrt*. *International Journal of Advanced Robotic Systems*, 10(7):299, 2013. doi:10.5772/56718. URL `https://doi.org/10.5772/56718`.

5 Appendix



(a) Dijkstra

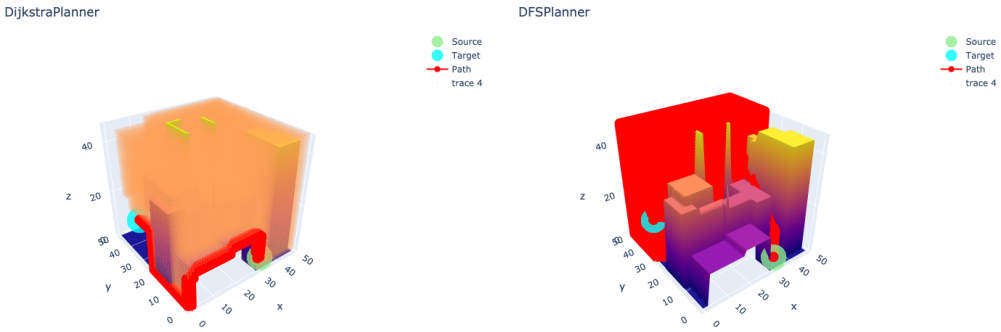
(b) DFS

(c) A*

(d) RRT

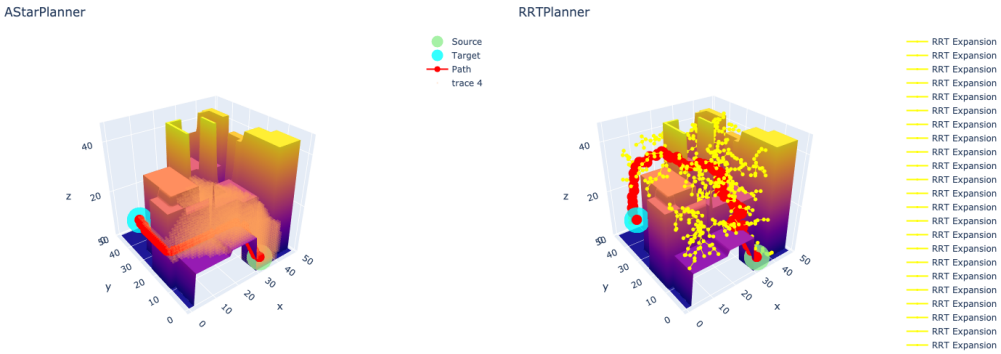
(e) RRT*

Figure 7: Planning Result Visualization (Large)



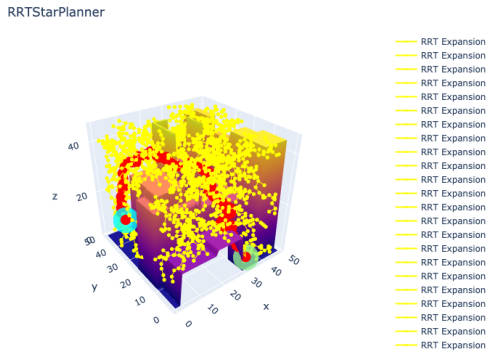
(a) Dijkstra

(b) DFS



(c) A*

(d) RRT



(e) RRT*

Figure 8: Planning Result Visualization (Medium)

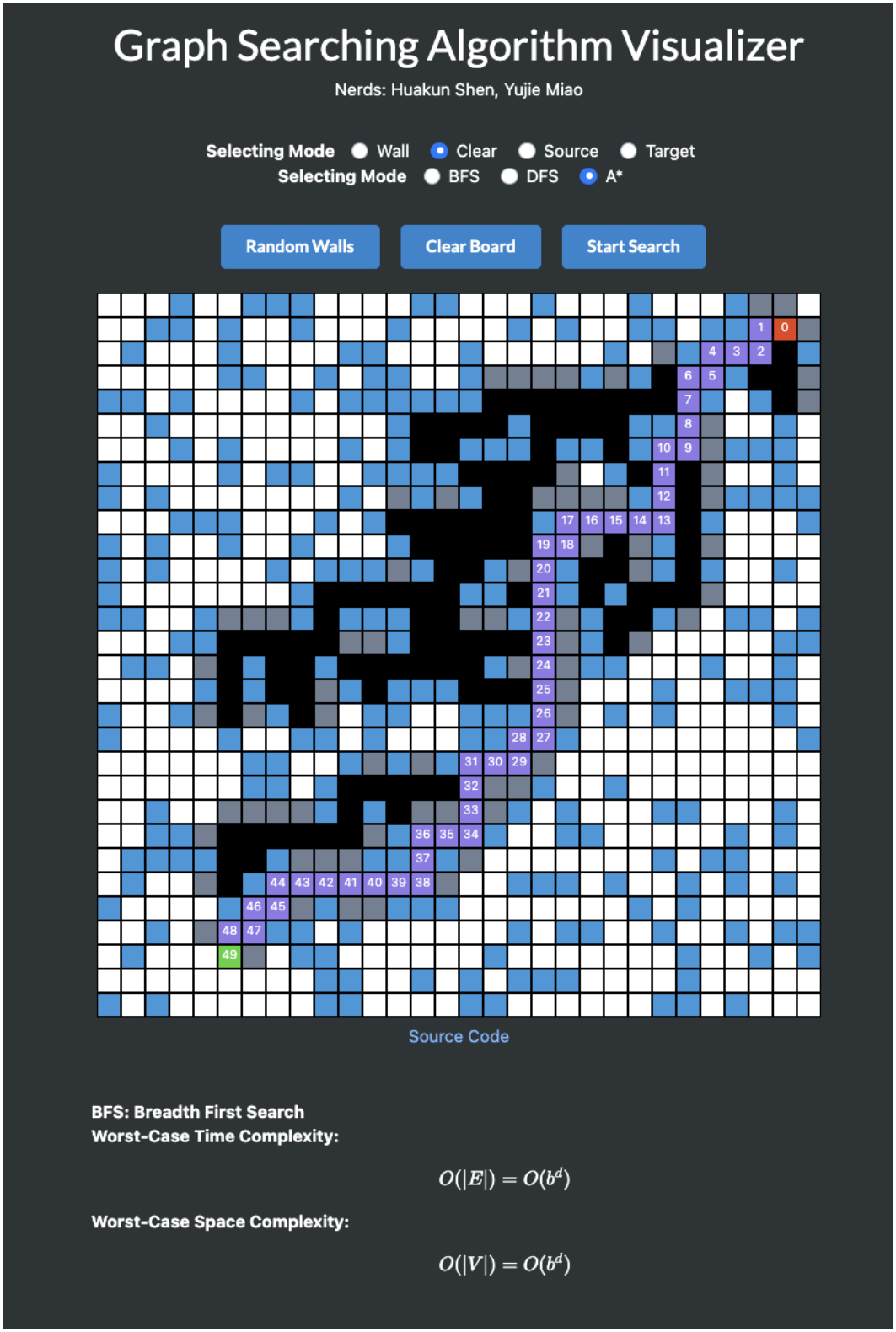


Figure 9: 2D Representation