A Comparison for Between Search-based and sampling-based Motion Planning Algorithm

Wenran, Tian

I. Introduction

THis is an era of known. With more and more sensors implemented in robotic system, the environment, or say the state of the robot, will be more clear and ready to preprocess. Once we know the goal of robot, our next step to give a proper motion planning algorithm which helps robot to move. In this paper, we will talk about the possible solutions for motion planning for a point in 3D environment which will contain a series of AABB blocks, and a start and a goal. Firstly, we try to find the solution by a weighted A* algorithm, which is search-based, and a famous sample-based algorithm which is called RRT*. Secondly, we will compare the performance for these two algorithms and try to give some basic concepts about how to choose them in real implementation.

II. Problem for Statement

A. Environment and Collision Checking

In this paper, firstly we will be given a range of a 3D environment, and a series of axis-aligned bounding boxes (AABBs). Our algorithm should can give the path from any pair of possible start point and goal point, which can avoid the blocks, and return the cost of the path.

Any path from start to goal will combined with a series of points, and each consecutive pair of points is called "step". So basically, each step will be a line segment, and any step intersect with any block will be a collision. Clearly, we do not want any collision in our algorithm. Then, for a segment we will have two points which is point (x_1, y_1, z_1) and point (x_2, y_2, z_2). And we will also use two points which is left-lower-front corner (x_1^b, y_1^b, z_1^b) and right-upper-rear corner (x_2^b, y_2^b, z_2^b) to represent the block. Our goal for collision checking algorithm is to show if a segment intersects with any block in the environment. We will use $colli(i_1,\ i_2) \in \{0, 1\}$ to denote this algorithm. If there exists collision between any block and segment $i_1 i_2$, we have $colli(i_1,\ i_2) = 1$

B. DSP Problem in 3D Space

To solve the problem, we should firstly formulate the model into a deterministic shortest path problem in 3D space:

1. Vertices $\mathcal{V}$: there will be limited number of the vertices in the 3D space, but the number should be enough so that we can find the path from the vertices. For start point s and goal point τ we have: $s, \tau \in \mathcal{V}$.
2. Edges ε: for any two vertices $i_a,\ i_b$, if we have $colli(i_a,\ i_b) = 0$, then there can be an edge between two vertices ε_{ab}.
3. Cost C: the cost for an edge ε_{ab} will be the length of the edge (Euclidean distance for two vertices), if i_a denotes the 3D coordinate of vertex i, we have: $c_{ab} = \|i_a - i_b\|_2$. We can assume that there is no negative cycles in the graph, because each weight will be non-negative.
4. Path: a sequence $i_{1:q} = (i_1, i_2, \dots, i_q)$, where $i_k \in \mathcal{V}$. We use $P_{s,\tau} = \{i_{1:q} | i_k \in \mathcal{V},\ i_1 = s,\ i_2 = \tau\}$ to denote all paths from s to τ
5. Path length: sum of the edge weights along the path: $J^{i_{1:q}} = \sum_{k=1}^{q-1} c_{k,k+1}$
6. Objective: find a path that has the minimum length from s to τ: $dist(s,\tau) = \min\limits_{i_{1:q} \in P_{s,\tau}} J^{i_{1:q}}$

Based on the definition above, we can transform the question into a deterministic finite state problem to do deeper analysis.

C. Implementation

To solve the problem, we use the weighted A* algorithm, which is search-based, and a famous sample-based algorithm which called RRT*. In the real implementation, we need to some specific definition about the vertices, path and objective so that we can substantiate them to codes.

III. Technical Approach

A. Collision Checking

First of all, based on geometry analysis we can easily get the following conclusion between a line segment and a cubic AABB block:

1. If a line segment intersects with a AABB block, there will be common part between their coordinate range in each dimension
2. Given a line segment with two endpoints (x_1, y_1, z_1) and (x_2, y_2, z_2), and an AABB block with corner $(x_1^b, y_1^b, z_1^b)\ and\ (x_2^b, y_2^b, z_2^b)$. We can give the following statements:
   a. The intersection must happen in the common range between $[x_1, x_2]$ and $[x_1^b, x_2^b]$. And we use $[x_{\mathrm{l}}, x_{\mathrm{r}}]$ to denote $[x_1, x_2] \cap [x_1^b, x_2^b]$.
   b. For a line in 3D, we can have:
   $$y = ax + b$$
   $$z = cy + d$$
   c. After we insert $[x_{\mathrm{l}}, x_{\mathrm{r}}]$ to the function above, we can

have $[y_{\mathrm{xl}}, y_{\mathrm{xr}}]$. The intersection must happen in the common range between $[y_{\mathrm{xl}}, y_{\mathrm{xr}}]$ and $[y_1^b, y_2^b]$. And we use $[y_{\mathrm{l}}, y_{\mathrm{r}}]$ to denote $[y_{\mathrm{xl}}, y_{\mathrm{xr}}] \cap [y_1^b, y_2^b]$.

d. In the same way, we insert $[y_{\mathrm{l}}, y_{\mathrm{r}}]$ to the function above and get $[z_{\mathrm{yl}}, z_{\mathrm{yr}}]$, and there must exits $[z_{\mathrm{yl}}, z_{\mathrm{yr}}] \cap [z_1^b, z_2^b] = [z_{\mathrm{l}}, z_{\mathrm{r}}]$ if segment intersects with the block.

For any $[x_{\mathrm{l}}, x_{\mathrm{r}}], [y_{\mathrm{l}}, y_{\mathrm{r}}]$ and $[z_{\mathrm{l}}, z_{\mathrm{r}}]$ does not exist, we can give the conclusion that the line does not intersect with the block, or if all three exist the line will intersect with the block.

Based on the above analysis, we can give a program easily for collision checking.

B. Weighted A* algorithm

In our real approach, we used A* algorithm and it is proved to be completeness if it is implemented properly from experience. We build the weighted A* algorithm by the following principles:

1. Grid the environment into voxel-like cells, and each center of the cell will possibly be a vertex. And there is a trade-off between the number of cells and algorithm efficiency. As we know, if there are more cells in each unit length, the algorithm will be more precise, but the growth rate for computation complexity is cubic. After analysis the environment, in the implementation we set each unit length contains 5 cells.
2. Define the control policy: each cell has at most 26 neighbors so that there are at most 26 types of control input, since we want the state just move to its neighbor for each time strap.
3. Define the vertices: we try to avoid the collision during the search phrase, so that we define the vertices at all the centers for the cells which are not occupied by any block. If we move a point by the control policy (to its neighbor), there is certainly no collision.

In fact, because we used the 26-connected control policy, our algorithm cannot get to the optimal goal in every task if the none neighbor nodes can connected with each other.

Here is the pseudocode for a normal weighted A star problem:

```
$OPEN \leftarrow \{s\}, CLOSED \leftarrow \{\,\}, \epsilon \geq 1$
$g_s = 0, g_i = \infty\ for\ all\ i\ \in \mathcal{V} \backslash \{s\}$
$\boldsymbol{while}\ \tau \notin CLOSED\ \boldsymbol{do}$:
    $Remove\ i\ with\ smallest\ f_i = g_i + \epsilon h_i\ from\ OPEN$
    $Insert\ i\ into\ CLOSED$
    $\boldsymbol{for}\ j\ \in Children(i)\ and\ j \notin CLOSED\ \boldsymbol{do}$:
    $\boldsymbol{if}\ g_i > (g_i + c_{ij})\ \boldsymbol{then}$:
        $Parent(j) \leftarrow i$
        $\boldsymbol{if}\ j \in OPEN\ \boldsymbol{then}$:
            $Update\ priority\ of\ j$
        $\boldsymbol{else}$:
            $OPEN \leftarrow OPEN \cup \{j\}$
```

In our code, each of the following list g, $CLOSED$ and $Parent$ will be set as 3D tensor, which will be easy to index. In this paper, we use a classical way to define the heuristic h, which is can be proved to be consistent:

$$h_i = \|i_r - i_\tau\|_\infty + 0.7\|i_r - i_\tau\|_{-\infty}$$

$$h_i \leq dist(i, t)$$

The priority in $OPEN$ list is the cost from start point to the node plus ϵ times heuristic h. Our algorithm will be at least ϵ-suboptimal. Our algorithm will search all vertices which is accessible by the start point, and if it gets the goal, the main loop will break and the path will be saved in $Parent$. And the algorithm will keep in searching if it does not get to the goal vertex. So, our algorithm is completeness.

There is a quite profound point is that it is not easy to build an efficient data structure which can update the priority of j in $OPEN$ list. Normally, we would like use a priority queue to build the OPEN list, but if we want to update the priority of a node in the queue, generally we need to search the entire queue firstly because with only a node j we cannot tell where is it in queue. This will lead to a $O(n)$ computational complexity. This should be avoided because this process is contained in another loop (the main loop). In addition, the heapq package, which can be the most common priority queue API in Python, even does not provided the method to update a node. So, it is better for us to find a way to optimize the original algorithm.

After analysis, we found that for each iteration in main loop, we will drop to the vertex with highest priority to the $CLOSED$. And each node in $CLOSED$ should not be searched again. So if we just insert the j into $OPEN$ list instead of search and update it, and add a $in\ CLOSED$ judgement in the main loop, the result should be the same. The main loop will be like:

```
$\boldsymbol{while}\ \tau \notin CLOSED\ \boldsymbol{do}$:
    $Remove\ i\ with\ smallest\ f_i = g_i + \epsilon h_i\ from\ OPEN$
    $\boldsymbol{if}\ i\ in\ CLOSED$:
        $\boldsymbol{continue}$
    $Insert\ i\ into\ CLOSED$
    $\boldsymbol{for}\ j\ \in Children(i)\ and\ j \notin CLOSED\ \boldsymbol{do}$:
    $\boldsymbol{if}\ g_i > (g_i + c_{ij})\ \boldsymbol{then}$:
        $Parent(j) \leftarrow i$
        $OPEN \leftarrow OPEN \cup \{j\}$
```

As we know, the computational complexity for an insert to a priority queue should be only $O(\log(n))$, which is much faster than the operation in $O(\mathrm{n})$.The testing results can shows that our algorithm can run much faster after this amend.

Here, we use n to denote the number of vertices. If our environment has length, width and height: (l, w, h), and there are r cells per unit length, we can have $n = lwhr^3$. In our code, each of the data structure is define by the gridded space, so our space complexity of the code will be $O(n)$. Besides, in the main loop, each time of iteration will put a vertex to the $CLOSED$, then the worst situation is that our code will iteration for all n vertices, and each vertex requires time at most $26\log(n)$ to process. These will cause computation complexity as $O(n\log(n))$. So, our algorithm will have the space complexity $O(n)$ and computational complexity $O(n\log(n))$, which is quite reasonable. And we can try to set the ϵ equals to 1 in real implementation because our algorithm is such efficient.

C. RRT* Algorithm

Technically, the RRT algorithm is a tree constructed from random samples with start vertex i_s. The tree will keep in random grow until it connected with goal vertex i_τ. The RRT* algorithm is basically the RRT algorithm plus rewiring of the tree, a part which is quite like label correcting, to ensure asymptotic optimality. It contains two steps: extend (similar to RRT) and rewire. Both RRT and RRT* are well-suited for single-shot planning, and requires less computational and space complexity because they are sample-based. A normal implementation for an RRT* algorithm will be like:

```
$V \leftarrow \{x_s\};\ E \leftarrow \emptyset$
for $i = 1 \dots n$ do
    $x_{rand} \leftarrow$ SampleFree()
    $x_{nearest} \leftarrow$ Nearest($(V, E), x_{rand}$)
    $x_{new} \leftarrow$ Steer($x_{nearest}, x_{rand}$)
    if CollisionFree($x_{nearest}, x_{new}$) then
        $X_{near} \leftarrow$ Near($(V, E), x_{new}, \min\{r^*, \epsilon\}$)
        $V \leftarrow V \cup \{x_{new}\}$
        $c_{min} \leftarrow$ Cost($x_{nearest}$) + Cost($Line(x_{nearest}, x_{new})$)
        for $x_{near} \in X_{near}$ do
            if CollisionFree($x_{near}, x_{new}$) then
                if Cost($x_{near}$) + Cost($Line(x_{near}, x_{new})$) $< c_{min}$ then
                    $x_{min} \leftarrow x_{near}$
                    $c_{min} \leftarrow$ Cost($x_{near}$) + Cost($Line(x_{near}, x_{new})$)
        $E \leftarrow E \cup \{(x_{min}, x_{new})\}$
        for $x_{near} \in X_{near}$ do
            if CollisionFree($x_{new}, x_{near}$) then
                if Cost($x_{new}$) + Cost($Line(x_{new}, x_{near})$) $<$ Cost($x_{near}$) then
                    $x_{parent} \leftarrow$ Parent($x_{near}$)
                    $E \leftarrow (E \setminus \{(x_{parent}, x_{near})\}) \cup \{(x_{new}, x_{near})\}$
return $G = (V, E)$
```

In the implementation RRT* Algorithm, we used the package from Python motion planning library at https://github.com/motion-planning/rrt-algorithms. Basically, this will help us to build the model we described. Then, we need to choose the following hyperparameters:

1. Sample numbers: as we know the process for RRT* requires select random point in environment. We cannot sample it unlimitedly or the algorithm will not stop. So, we need to set a threshold, and the threshold also need to be large enough. We set this to 16384 in the code.
2. Collison check resolution: a high resolution will make sure there is no collision in the path, but high resolution also requires more time complexity. We set this to 0.01 in the code, after run and test.
3. Edge length: A large length of edge will be time-efficient, but if the length is too large we cannot find a path in a more complicated environment. We set this to 1 in the code, after run and test.
4. Rewire count: a large value of this parameter will give us a smoother path. We set this to 64 in the code.

The implementation of RRT* from Python motion planning library requires 3 steps:

1. Set up the environment. In the library, there is a special class which is called "SearchSpace". This will build the environment with AABB blocks from what we required
2. Initialize the solver with environment and hyperparameters. In the library the solver class for RRT* is called "RRTStar", we need to initialize the class with previous mentioned environment and parameters.
3. Execute the RRT* search step by rrt_star() method in solver, this will return the path for us.

IV. Results

A. Parameters and Results from Weighted A*

The most important parameter of Weighted A* algorithm might be the weight value ϵ. Here are the results from algorithm which cell density equal to 5 with different ϵ:

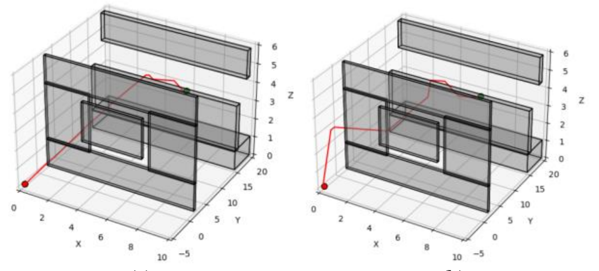

Fig1. The results from different ϵ. (a) is the result where $\epsilon = 1$, (b) is the result where $\epsilon = 1.2$. The results where $\epsilon = 1.2,\ 1.5$ *and* 2 are quite like (b)

Table 1. Results from different ϵ in weight A* algorithm

	Searched points	costs	runtime	Collision
$\epsilon = 1$	48709	26.60	6.60s	False
$\epsilon = 1.2$	2404	26.94	0.46s	False
$\epsilon = 1.5$	873	27.06	0.21s	False
$\epsilon = 2$	628	27.65	0.17s	False

From the Table1, we know that with a larger ϵ value, the algorithm will run faster and faster, but the corresponding cost will increase. This is basically accord with our experience of the weight. With a smaller ϵ, our algorithm will follow the heuristic function more strictly, which will gives us a lower cost but need to search more nodes; with a larger ϵ, our algorithm will follow the heuristic function less strictly so that it will more likely to return other paths with more costs. In our real code, we choose $\epsilon = 1.2$ for it seems give a great trade-off.

B. Parameters and Results from RRT*

Firstly, we talk about how the collision checking resolution will influence the algorithm. Because after test, we found that this may the main cause for elongation for runtime. The results from different resolution r will be like:

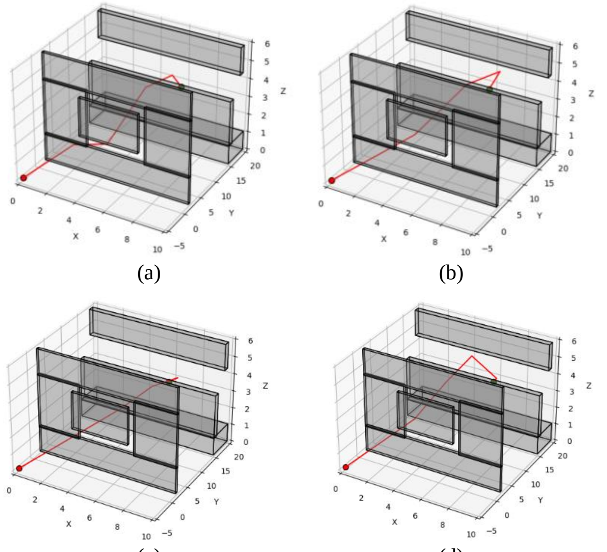

Fig2. The results from different r for RRT* algorithm. (a) is the result where $r = 0.01$; (b) is the result where$r = 0.02$; (c) is the result where$r = 0.05$; (b)is

the result where $r = 0.1$.

Table 2. Results from different r in RRT*

	Searched points	costs	runtime	Collision
$r = 0.01$	277	24.58	16.08s	False
$r = 0.02$	451	25.30	8.10s	False
$r = 0.05$	376	24.81	3.61s	False
$r = 0.1$	517	25.14	3.11s	True

From the table, we can know that with a lower collision checking resolution, the iteration efficiency will be higher， and the algorithm will require less runtime. There is no obviously influence on the cost by changing the resolution. But what's important is that if the resolution is too low, there may exist collision in the return path. Finally, we choose $r = 0.02$ in our code.

Secondly, we talk about how rewire count will influence the algorithm. The results with $r = 0.02$ from different rewire count rc will be like:

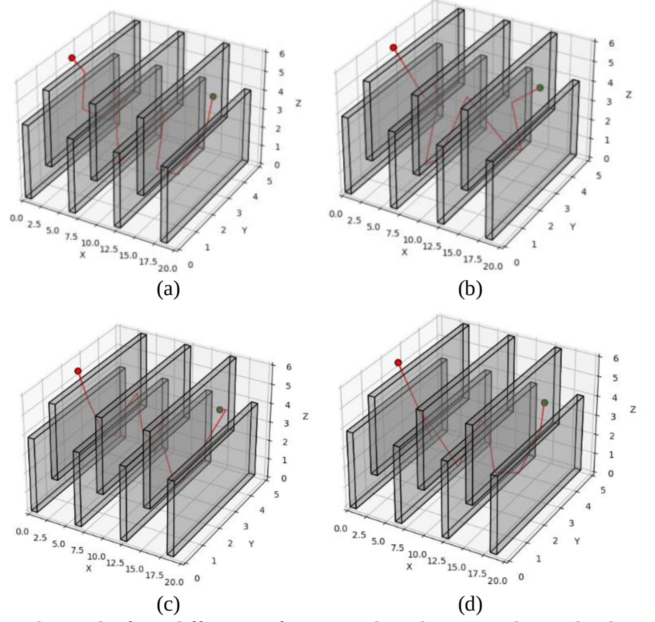

Fig3. The results from different rc for RRT* algorithm. (a) is the result where $rc = 8$; (b) is the result where $rc = 32$; (c) is the result where $rc = 64$; (b)is the result where $rc = 128$.

Table 3. Results from different rc in RRT*

	Searched points	costs	runtime	Collision
$rc = 8$	488	30.47	2s	False
$rc = 32$	560	29.91	9.29s	False
$rc = 64$	840	27.70	30.61s	False
$rc = 128$	390	27.41	35.98s	False

From the Fig.3, we can know that with a larger rewire count, the path will be smoother. This is corresponding to the fact that the higher rc, the lower cost in the table. Besides it is also conform to our experience that a larger rewire count will require more runtime. It seems that $rc = 32$ gives a good trade-off between runtime and cost.

C. *Weighted A* vs RRT**

Finally, we will versus the results from weighted A* and RRT*. For weighted A*, we will choose $\epsilon = 1.2$, and for RRT* we will choose $r = 0.02$ and $rc = 32$. After running all of the environments, we can get the results like Fig4, Table 4 and Table 5.

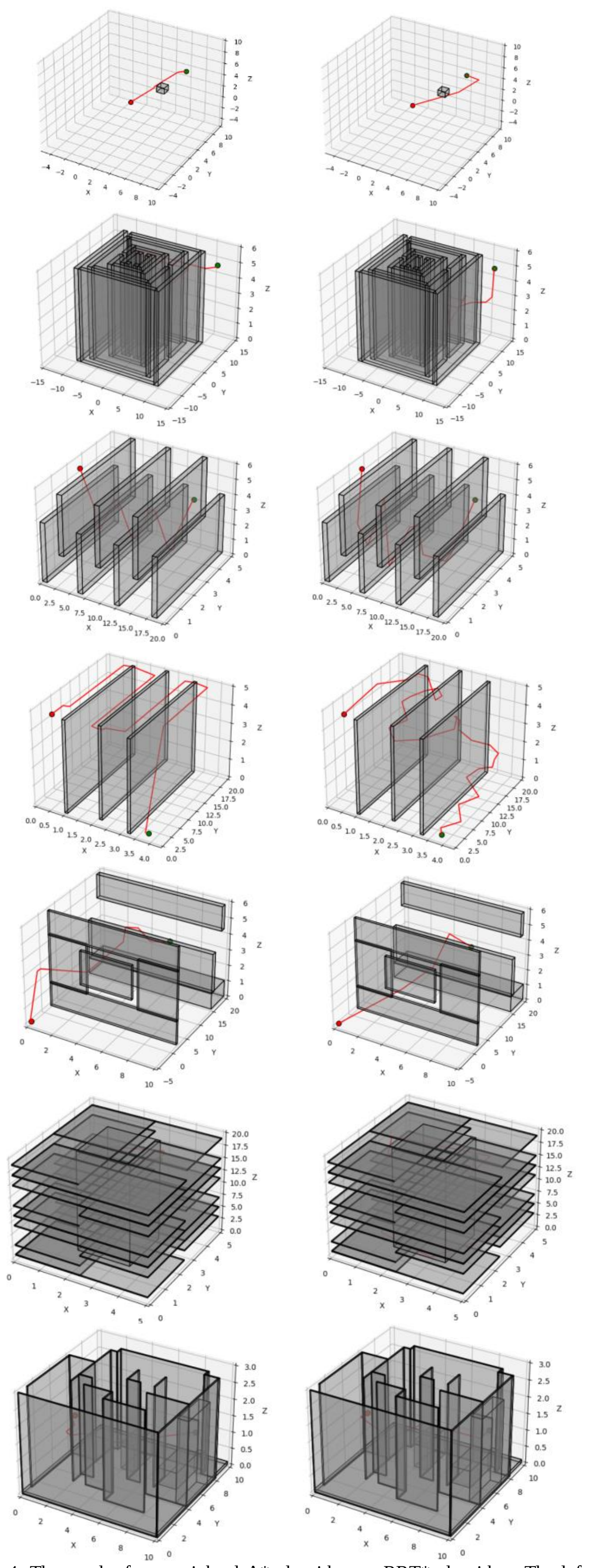

Fig4. The results from weighted A* algorithm vs RRT* algorithm. The left graphs are the results from weighted A*, right graphs are the results from RRT*.

Table 4. Results from different environment in weighted A*

	Searched points	costs	runtime	success
$single\ cube$	27	7.98	0.25s	True
$maze$	129566	75.79	17.72s	True
$flappy\ bird$	23581	25.96	3.65s	True
$monza$	30567	76.75	3.91s	True
$window$	2404	26.95	0.47s	True
$tower$	27479	29.61	3.62s	True
$room$	2157	11.59	0.38s	True

Table 5. Results from different environment in RRT*

	Searched points	costs	runtime	success
$single\ cube$	419	10.66	2.14s	True
$maze$	9580	78.48	117.30s	False
$flappy\ bird$	483	29.46	14.03s	True
$monza$	2197	75.80	55.25s	True
$window$	243	24.49	1.94s	True
$tower$	701	34.15	19.77s	True
$room$	134	11.54	2.48s	True

Though the costs are not quite different between two algorithms, the runtime for RRT* algorithm is much longer than weighted A* algorithm. This seems more surprisingly when you see generally the search points for RRT* in much less than the points in weighted A* algorithm. In addition, from the graph, the path from the results of weighted A* algorithm is smoother.

There are three reasons that the RRT* algorithm gives a worse performance

1. The RRT* requires check the collision for each iteration. This requires plenty of computational resources. We set $r = 0.02$, but still there is a collision in maze environment. To avoid the collision, we need to rerun the algorithm or set a smaller r. But both methods will require even more time to run.
2. For the weighted A* algorithm this will not happen because all the vertices are neighbor-connected and the vertices which might occupied by block are ignored.
3. The dimension is not high enough. As a sample-based algorithm, the RRT* is more suitable in high-dimensional environment like robot with a lot of sensors and motors. But in this case, the dimension is only three, the search-based algorithm is still competitive.
4. Although both of the algorithm are written in Python, the most preferable package for RRT* is from OMPL, which is written in C/C++. That may give a better performance since it's used more widely.

So finally, our conclusion is that in the lower dimension environment like $d \leq 3$, the search-based algorithm weighted A* may give a better performance than sample-based algorithm RRT*.