Truly understanding sampling-based motion planning

Student: Seong Woo AHN 20226042
Answer sections will start with a → sign.
Please contact me via personal mail if certain links don't work: chrisahn99@gmail.com

This project is intended to make you truly understand and appreciate why sampling-based motion planning algorithms have become *the* dominant paradigm in robotics. You will also see its drawbacks, and that it is not *the* solution to motion planning problems. You will implement several sampling-based algorithms and answer a series of questions that test your understanding of motion planning algorithms. Your grades will depend on the depth of your understanding. Good luck!

Part 0: Environment creation
Create a motion planning domain of your choice. It can be a 2D or 3D environment. I recommend that you read the below before you set up your environment to plan ahead the kind of environment and robot you want.

→ For ease of use and sharing, I had initially decided to work in a google colab environment and use Python as the main coding language (as it is the coding language I am most proficient in).

I soon found out however, that there was an issue with rendering the pygame simulations (I chose pygame as the python library for simulating sampling algorithms), as the google colab VMs lack a native video reader. Thus, I decided to work locally using VSCode. My code is uploaded on github, and here is its link:

https://github.com/chrisahn99/Sampling-algorithms

The simulation environment is a 2D environment, rendered using the pygame library. (Below is an example of a sim environment, where the squares are obstacles).

→ The way the 2D environment is generated is that a start point and a goal point are arbitrarily defined on the upper left edge and the lower right edge respectively (in green and red circles). Obstacles are then generated randomly with a number that is once again an arbitrary parameter. The task for the robot is to sample a collision-free trajectory from the start point to the goal point.

Below is an example picture of a simulation environment (notice that the goal point in this image is green and not red as it is on simulation videos).

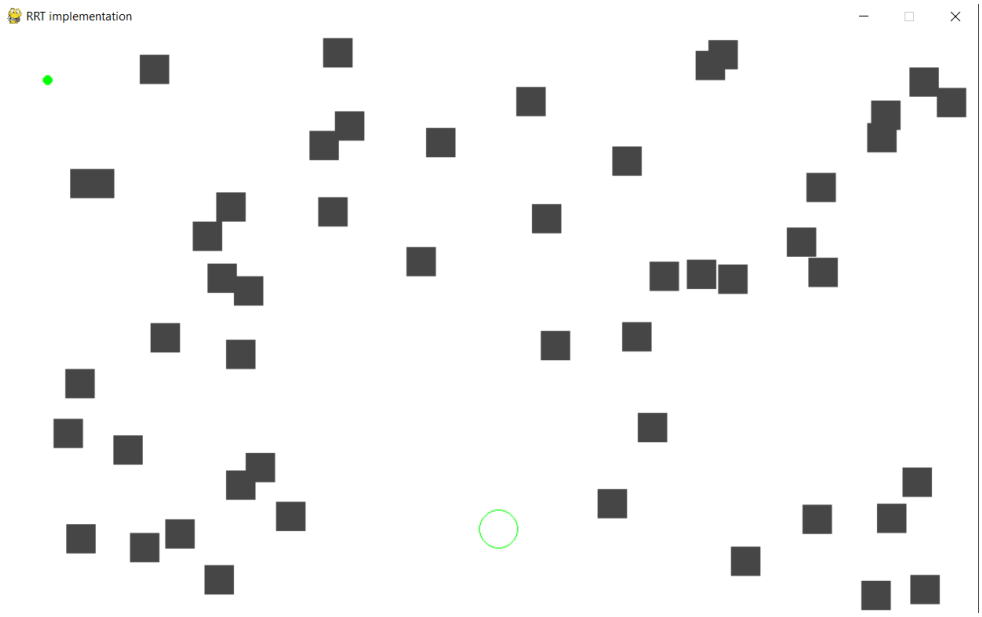

Figure 1: 2D simulation environment rendered using Pygame

1.0 Exploration and exploitation

1.1: An alternative sampling-based algorithm

Implement the following sampling-based algorithm:

1. Initialize a tree with a initial configuration
2. Select a random node from the tree, called x_{tree}
3. Select a random configuration, called x_{rand}, from the collision-free configuration space
4. Extend from x_{tree} to x_{rand}
5. Add the new nodes from the extend operation to the tree
6. If a goal configuration is added to the tree, terminate. Solution found!
7. Otherwise, go to step 2

Visualize its tree-growing procedure, like this video.

→ Here is an implementation of the above algorithm.

Answer the following questions:

1. Is this algorithm probabilistically complete? Why or why not?
2. Comment on the tree-growth procedure. Does it efficiently explore the configuration space? Why or why not?

→ 1. First of all, I am dealing with a **discrete graph** with my simulation environment (pygame environment). The fact that the grid map is discretized **strongly favors for probabilistic completeness** as the algorithm could **exhaustively test all possible points** on the grid until it finds one that connects the tree to the goal (the 2D environment is 800x1200 so around 1 million grid points). Secondly, we have proven in class that for RRTs, in **any C-space**, if **we grow a tree** then we get **arbitrarily close to any point in the C-space** from the **closest point on the tree**. For our case of course, unlike RRT we aren't choosing the closest point in the tree to our randomly generated configuration, but we're taking a random point: this, however doesn't really matter **as the number of iterations reaches infinity**, since all possible positions can be tested and added to the tree (including those that would've been added with RRT). Therefore, the algorithm is **probabilistically complete**.

→ 2. No, it **does not explore the configuration space efficiently**. As the above video shows, as points are chosen randomly from the existing tree, there is **no bias that speeds up the orientation towards the goal**. Instead, the algorithm naively tests points that are concentrated in the same region, and paths are tested in a very inefficient way.

1.2: Exploiting domain knowledge

Implement the following alternative algorithm: instead of choosing a random node from the tree in line 2 of the algorithm given in part 1, choose the node with the least heuristic value defined by:

$$x_{tree} = argmin_{x \in V} \text{straight-line-distance-to-goal}(x)$$

where V is the set of nodes in the current tree, and the straight-line-distance-to-goal function measures the straight-line distance to the goal from the given configuration x. Visualize its tree-growing procedure.

→ Implementation video

Answer the following questions:

1. Is this more efficient than the previous algorithm? Why or why not?
2. How would you improve this algorithm?

→ 1. It is much more efficient than the previous algorithm. Unlike the previous one which randomly selects a node from existing ones, here we have **a bias that constantly pushes the graph closer to the goal**, and which is why it gets to the goal much faster and efficiently.

→ 2. A way this algorithm could be improved would be to **sample random points in a limited area** around the last tree element, since we know that at each iteration, we are connecting sampled points to those that are closest to the goal. The radius of this region can also have a **decaying evolution**, that is it reduces at every iteration so as to have **more efficient convergence towards the goal**.

1.3: RRT

Now implement an RRT. Visualize and compare the tree-growth procedure with the algorithms given in parts 1 and 2.

→ implementation video

Answer the following questions:

1. Is RRT more efficient than the ones given in 1.1 and 1.2? If so, describe what makes RRT more efficient.
2. In what cases will the algorithm in 1.2 be more efficient than RRT?

→ 1. We can clearly see that for 10 obstacles, RRT is quicker than both 1.1 and 1.2. The fact that the **arbitrary choice of a point** in the tree to be extended to the randomly sampled one makes it more efficient than one that does this **task randomly**, therefore the superiority of RRT to 1.1 is trivial. As for 1.2, the comparison is a bit more subtle.

In fact, less increase the number of obstacles (50) to see how both algorithms do:

→ **RRT**
→ **1.2**

As we can see, RRT seems to search the grid **more widely**, whereas 1.2 tries to go **more deeply** towards the goal. It seems like here we are comparing something like **BFS** and **DFS**.

However, due to its sampling nature, these types of sampling algorithms sometimes fail regardless of their bias.

https://www.youtube.com/watch?v=hsOiH92iwBQ

→ 2. Taking inspiration from the previous observation, I'd say that an obstacle terrain which has a direct opening towards the goal configuration would be much quicker for 1.2 than RRT, since here RRT would waste some time sampling out other areas.

Here are examples of an improved version of RRT using a biasing inspired from 1.2.
https://www.youtube.com/watch?v=XIQ5e5tXNec
https://www.youtube.com/watch?v=_ku_3LDntPQ

NOTE: Unfortunately, due to a lack of time and a lack of organization on my behalf, I wasn't able to answer the 2nd part of the project. The fact that too many assignments, projects, exams, and presentations were crammed up during the same period made it difficult for me to properly allocate time for each task. I sincerely apologize for this.

2.0 Limitations of RRT*

2.1 RRT*

Here is a pseudocode for RRT*:

Algorithm 1: $\texttt{RRT}^*((V, E), N)$

```
for i = 1, ..., N do
    x_rand ← Sample;
    X_near ← Near(V, x_rand);
    (x_min, σ_min) ← ChooseParent(X_near, x_rand);
    if CollisionFree(σ) then
        V ← V ∪ {x_rand};
        E ← E ∪ {(x_min, x_rand)};
        (V, E) ← Rewire((V, E), X_near, x_rand);
return G = (V, E);
```

Algorithm 2: $\texttt{ChooseParent}(X_{\text{near}}, x_{\text{rand}})$

```
minCost ← ∞; x_min ← NULL; σ_min ← NULL;
for x_near ∈ X_near do
    σ ← Steer(x_near, x_rand);
    if Cost(x_near) + Cost(σ) < minCost then
        minCost ← Cost(x_near) + Cost(σ);
        x_min ← x_near; σ_min ← σ;
return (x_min, σ_min);
```

Algorithm 3: $\texttt{Rewire}((V, E), X_{\text{near}}, x_{\text{rand}})$

```
for x_near ∈ X_near do
    σ ← Steer(x_rand, x_near);
    if Cost(x_rand) + Cost(σ) < Cost(x_near) then
        if CollisionFree(σ) then
            x_parent ← Parent(x_near);
            E ← E \ {x_parent, x_near};
            E ← E ∪ {x_rand, x_near};
return (V, E);
```

V - nodes in the tree
E - edges in the tree
N - the total number of iterations
Steer - Equivalent to Extend function
Cost(x) - cost from the initial configuration to x
Near(x) - computes a set of configurations near x. It is computed by

$$\texttt{Near}(V, x) := \left\{ x' \in V \ : \ \|x - x'\| \leq \gamma \left(\frac{\log n}{n} \right)^{1/d} \right\},$$

where n is the number of nodes in the tree, d is the dimensionality of the configuration space, and gamma is a user-defined constant

Implement RRT*. Is it faster than RRT? What is the most time consuming procedure?

2.2 (Potentially) Improving RRT*

Suppose we have the following two functions:

- CostLocal(x_1, x_2) - defined as a straight-line distance between configurations x_1 and x_2
- CostGlobal(x) - defined as a straight-line distance from x to x_{goal}

In the *ChooseParent* function above, instead of choosing the one with the least cost, use the following to determine x_{min}:

$$x_{min} = argmin_{x \in X_{near}} \text{Cost}(x) + \text{CostLocal}(x, x_{rand}) + \text{CostGlobal}(x)$$

In the *Rewire* function, use

$$\text{Cost}(x_{rand}) + \text{CostLocal}(x, x_{rand})$$

instead of using

$$\text{Cost}(x_{rand}) + \text{Cost}(\sigma)$$

in Line 3, and skip the Steer operation.

Answer the following questions.

1. Is this RRT* variant faster than standard RRT*? Why or why not?
2. How would you improve this variant?