Path Planning Algorithms

Nansong Yi
Department of Mechenical Engeneering
University of Washinton
nansong@uw.edu

Yao-Chung Liang
Department of Mechenical Engeneering
University of Washinton
yliang2@uw.edu
https://github.com/nansongyi/SearchAlgorithm

1 Introduction

This project is based on *NLinkArm* model in our last homework. We will introduce Path Planning algorithms including graph-based method, A*, and sampling-based method, RRT. Different from the start code given, our A* algorithm utilizes PriorityQueue and HashTable to optimize this algorithm. Since RRT is a sample-based algorithm, the planned path is continuous in configuration space, which makes it hard to deal with edge case. We found an easy, efficient way to handle continuous and periodic configuration when implementing sample-based algorithms.

2 Methods

2.1 Graph-Based Search

2.1.1 A*

A* algorithm is derived from Dijkstra algorithm. The only difference between them is that A* takes the distance between current position and goal position into consideration to guide its search, as known as heuristics.

Algorithm 1 A*

```
frontier ← PriorityQueue()
frontier.put(start, 0)
came_from, cost_so_far ← HashTable()
came_from[start] = None
cost_so_far[start] = 0
while not frontier.empty() do
    current = frontier.get()
    if current == goal then
        break
    for next in graph.neighbors(current) do
        new_cost = cost_so_far[current] + graph.cost(current, next)
        if next not in cost_so_far or new_cost < cost_so_far[next] then
            cost_so_far[next] = new_cost
            priority = new_cost + heuristic(goal, next)
            frontier.put(next, priority)
            came_from[next] = current
```

2.2 Sample-Based Search

2.2.1 RRT

Due to the periodicity of the configuration space, it's hard to do collision check at edge case. We came up with concatenating 9 duplicate configuration spaces to form a big complete and continuous configuration space, which easily handle the edge case.

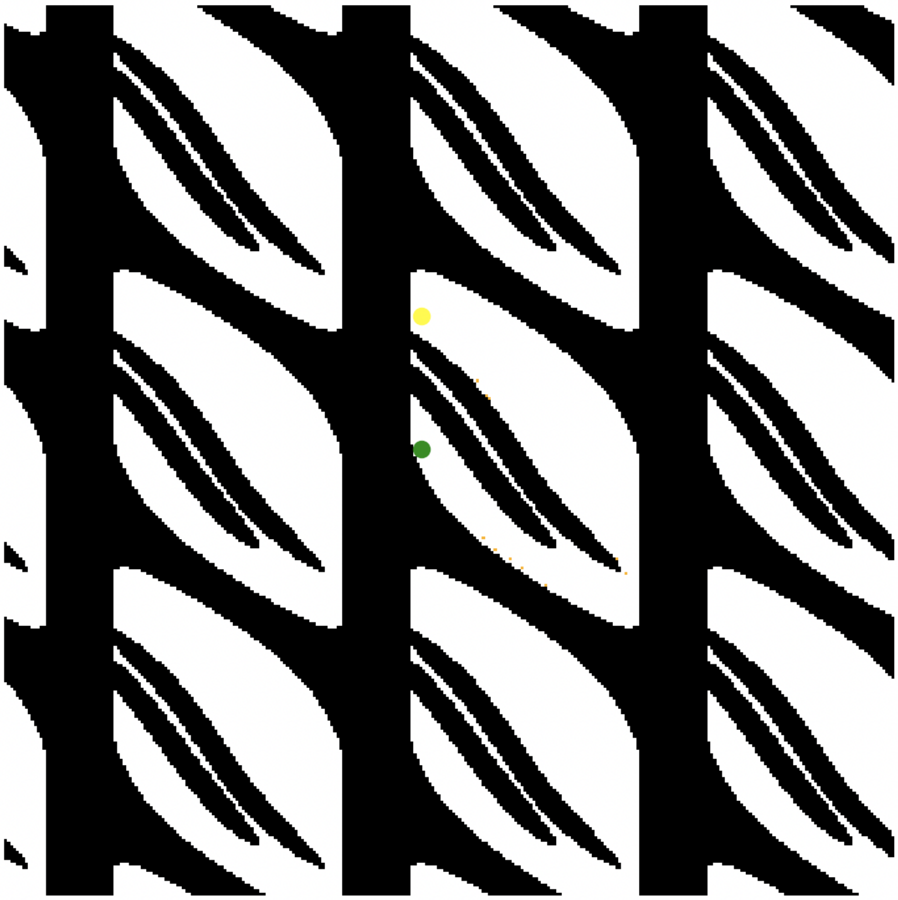

Figure 1: concatenating 9 configuration spaces

Algorithm 2 RRT(q_0)

$\mathcal{T}$.Init(q_0)
for i = 1 to K **do**
  $q_{rand} \leftarrow$ Rand($\mathcal{C}$)
  $q_{near} \leftarrow$ Nearest($q_{rand}, \mathcal{T}$)
  Extend($\mathcal{T}, q_{near}, q_{rand}$)

Algorithm 3 Extend($\mathcal{T}, q_{near}, q_{rand}$)

$q_{new} \leftarrow$ Steer(q_{near}, q)
if ObstacleFree(q_{near}, q_{new}) **then**
  $\mathcal{T}.V.add(q_{new})$
  $\mathcal{T}.E.add(q_{near}, q_{new})$

---

3 result

3.1 A*

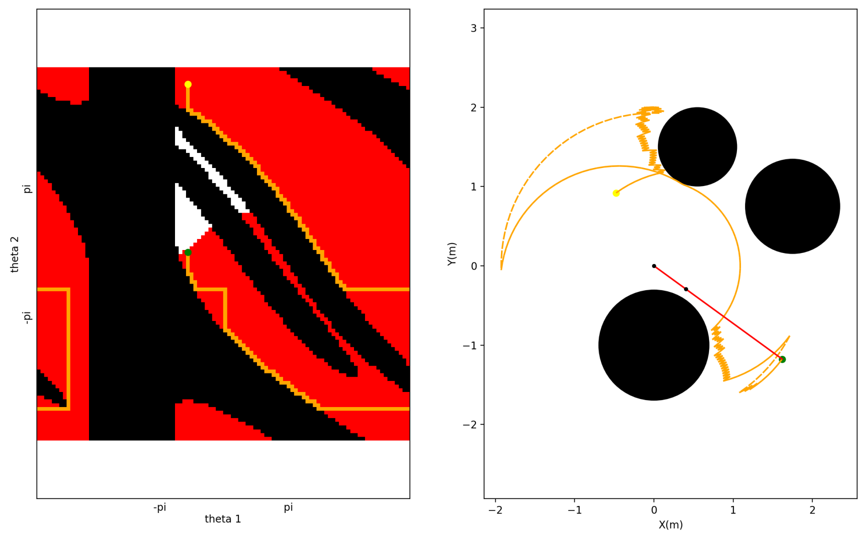

Figure 2: A*

3.2 RRT

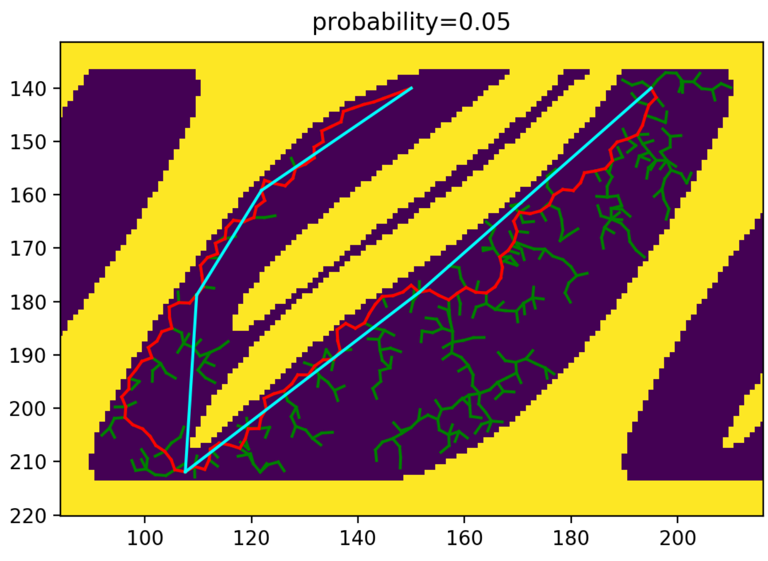

Figure 3: RRT plannar

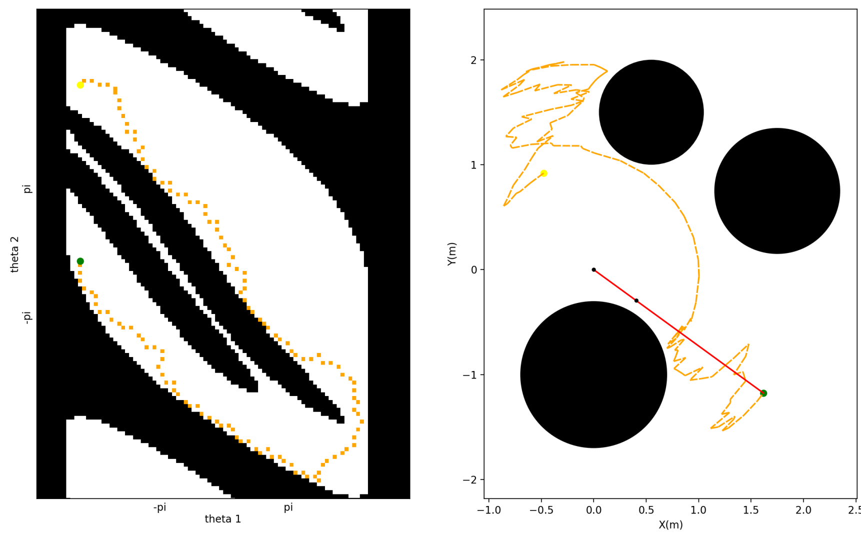

Figure 4: clean path

4 Conclusion

- RRT explore much less states in C-space than A*
- RRT performs worse than A* when the path need to cross some narrow mouth, otherwise better than A*.
- After post-processing, the RRT path becomes closer to global optimal.

References

[1] Hart, P. & Nilson, N. & Raphael (1968) A Formal Basis for the Heuristic Determination of Minimum Cost Paths. In *IEEE Transaction of Systems and Cybernetics*, Vol. ssc-4, No.2.

[2] LaValle, S. (2005) Rapidly-Exploring Random Trees: A New Tool for Path Planning.