mjana

Author	Meghdeep Jana (mjana)
Created	@October 25, 2021 8:32 PM
Last Edit	@October 25, 2021 9:55 PM
Tags	Academics
URL	

16782 HW2

Meghdeep Jana (mjana) 10.03.2021

1 OBJECTIVE

In this project, you will implement different sampling-based planners for the arm to move from its start joint angles to the goal joint angles.

For this homework implement four algorithms:

1. RRT
2. RRT-Connect
3. RRT*
4. PRM

To compile the code:

'graphs.h' and 'planner.cpp' should exist in the same folder (I will submit the .zip file confirming this)

The command:

```
mex planner.cpp
```

Also, the script `tests.m` can be run to perform the 20 experiments on the 4 planners in order.

2 Planners

The main loops of planners are defined as functions in planner.cpp, they have the following names:

- `RRTPlan()`
- `RRTConnectPlan()`
- `RRTStarPlan()`
- `PRMPlan()`

The RRT Tree class (`RRTree`), PRM class (`PRoadmap`) and other helper functions(`KNN`, Euclidian distances, `new_config`, `connect`) are defined in `graphs.h` this header file is `#included` in `planner.cppnode_reject()` and `local_bias()` heuristics are implemented as member functions of `RRTree()` class

A `struct vertex` is defined to store the vertex id and its joint configuration

The adjacency list is used to build the tree and roadmaps

1. <u>RRT:</u>
    a. The `RRTree` class is used to grow the tree(G). In which V = array of struct vertex and E = hash_map(parent,child)
    b. goal biasing is done with a probability of 0.1
    c. if new vertex is in epsilon range of goal and the line between new vertex and goal is obstacle free then we declare as RRT path found.
    d. we use `getPathRRT()` function to trace the hash_map and find the plan
2. <u>RRT-Connect:</u>
    a. Two objects of `RRTree` class are used to grow the trees Ta and Tb
    b. for Ta: `qa_nearest` is grown by epsilon towards `qrand` to give the `qnew` vertex. For Tb, `qb_nearest` is grown towards `qnew` to give the `qbnew` vertex.
    c. at the end of each iteration, `std::swap` is used to switch Ta and Tb objects.

d. We declare RRTConnect path found if `qbnew` and `qnew` have the same joint configuration. using the function `same_config()`

e. we use `getPathRRTConnect()` function to trace the hash_map and find the plan. As the edges are reversed for one of the trees, I had to utilize a stack (LIFO) to rearrange the vertices in the correct sequence to execute the plan

3. RRT*:

   a. The `RRTree` class is used to grow the tree(G). A hash_map is defined to keep track of cost at each vertex

   b. goal biasing is done with a probability of 0.1

   c. Extend function includes rewire steps. If cost of the respective vertex is updated (in the hash-map) if rewiring is done.

   d. The plan cost v/s time are kept track of as an array. Time and cost for initial plan are also captured. `getcost()` is used to calculate the cost, give a plan.

   e. we use `getPathRRT()` function to trace the hash_map and find the plan.

   f. `node_reject()` and `local_bias()` heuristics are implemented as member functions of `RRTree()` class

4. PRM:

   a. The `PRoadmap()` class is used to grow the roadmap(G). This object is defined as a global variable for the purpose of reuse between calls to the planner. If the map is changed, the preprocessing phase is invoked. an adjacency list (2D array) is used to keep track of the roadmap

   b. Preprocessing step is defined as a member function in the class (`build_roadmap()`) . The `connect_ends()` connects the given start and goal vertex in query step if they are not present in the roadmap

   c. A hashmap is used to keep track of different components in the roadmap

   d. If start and goal pose are in the same component then we proceed to use Djikstra to search in the roadmap. We get the successors of a vertex by looking up in the adjacency list.

   e. We use `getPath()` function to get the final sequence of joint configurations in the plan.

2 Experiments

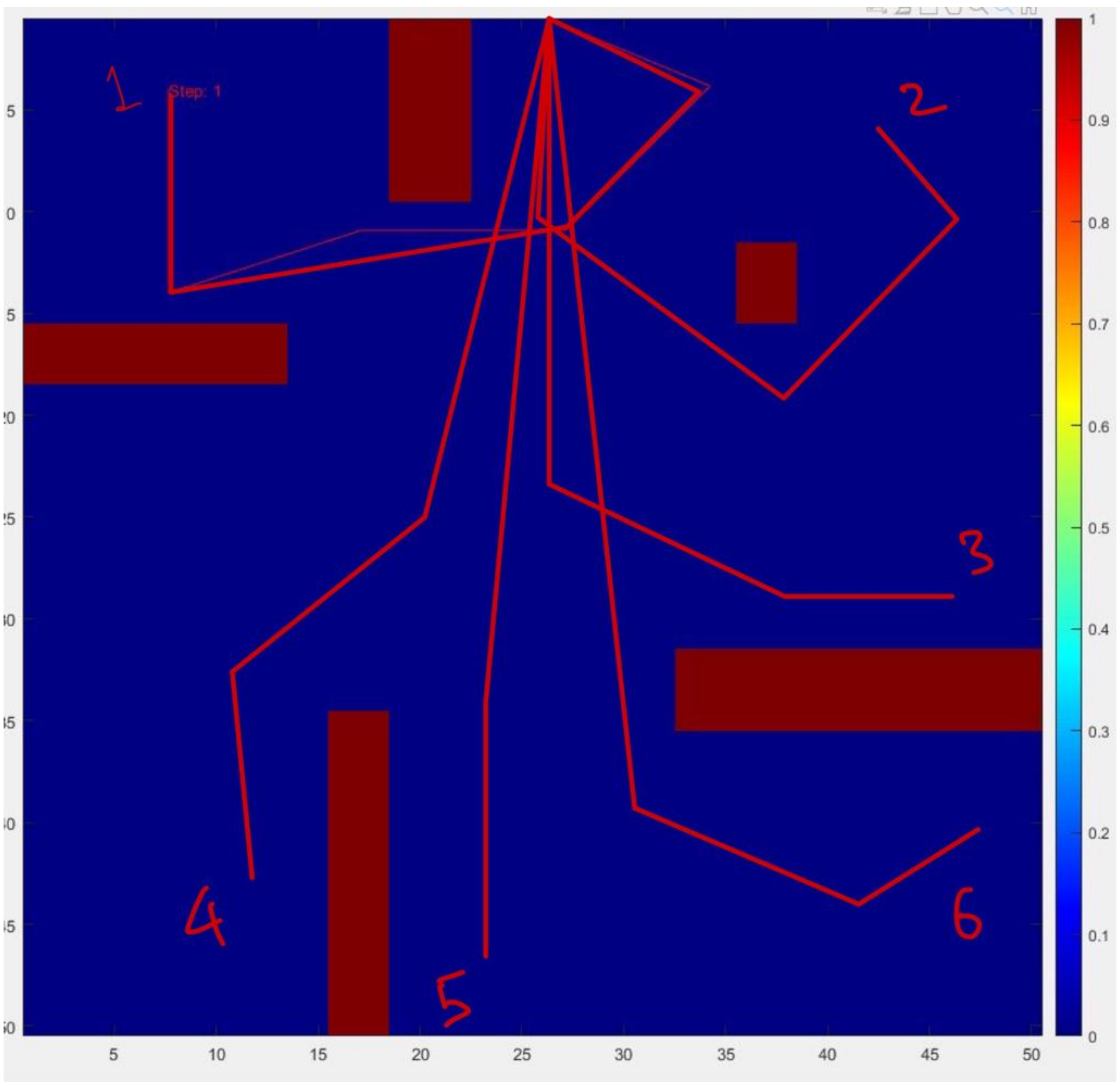
Path cost chosen is the Euclidean distance is joint space for the sequence of joints in the plan. This cost is utilized as the 'Path Quality' metric and metric for RRT* cost.

The threshold time for RRT* is set as 60 seconds and the max number of samples as 150,000.

PRM generates 200,000 samples in its preprocessing step.

2.1 Tests for 4 planners

To generate the result the following start-goal cases are chosen. 20 start-goal pairs are generated by picking two configurations from the 6 configurations shown below:



4 RESULTS

For RRT* the data is arranged as for the initial plan and final plan respectively. Where the final plan is when RRT* has exhausted the planning time or number of samples

4.1 Sampling-based planners on 20 tests

Property	Avg. Plan time (s)	Success Rate(%)	Avg. Vertices	Avg. Path Quality
RRT	1.663	85	3785	12.95

Property	Avg. Plan time (s)	Success Rate(%)	Avg. Vertices	Avg. Path Quality
RRT-Connect	**0.0047**	**100**	**175**	12.76
RRT*	1.834/60.0	75	2053/13698	**5.65/9.38**
PRM	1.878	95	13028	27.045
Untitled				

RRT: Incorporating goal-biased sampling was important for RRT. Without goal-biasing, it was tough for RRT to grow within the goal radius under the time or sampling threshold. Apart from 3 runs, RRT was able to generate a plan in under 5 seconds with goal biasing.

RRT Connect: was the fastest of the 4. Growing the tree from both ends and relaxing the epsilon helps in rapid connection from start to goal. This also results in less number of vertices generated.

RRT*'s extend function is more computationally expensive than RRT, hence having more average planning time for the initial solution than RRT. However, the final plan by RRT* gave the best path quality because of the rewiring in the tree.

PRM: The first plan generated by PRM took about 35 seconds (200,000 samples). But as PRM reuses its roadmap, the subsequent runs were very quick (~0.005 seconds). If the number of samples was kept lower, there were some instances when start and goal configurations were not part of the same component in the PRM, taking a hit to its success rate.

4.2 Conclusion

1. I think RRT-connect was the most suitable for the environment. It generates the plan extremely fast and utilizes the least amount of nodes in the graph.
2. Issues:
    a. One issue would be that the cost is still on the higher side compared to RRT*

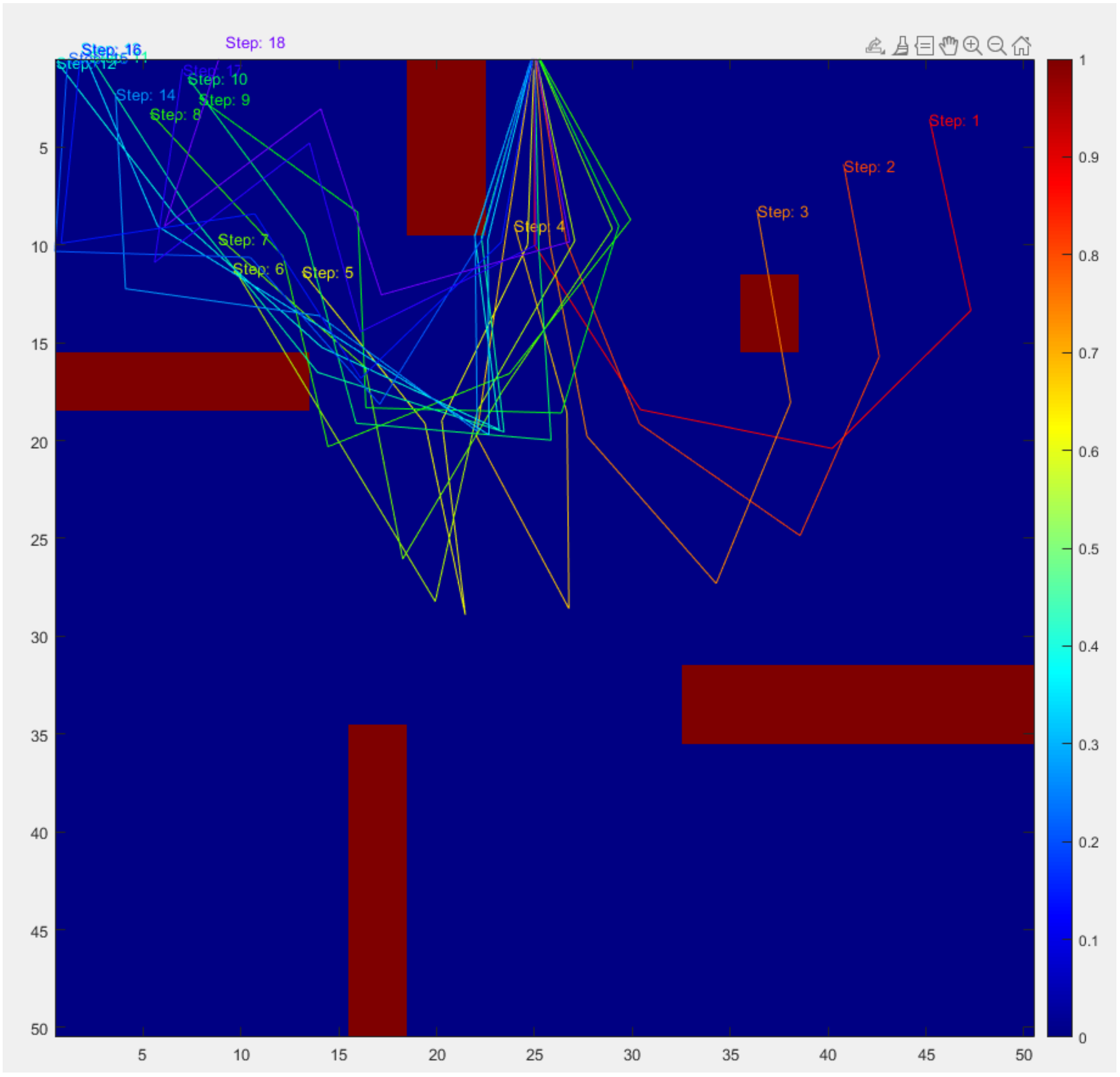
b. the number of nearest neighbor calculations is higher

3. Improvements:

    a. One way the plan can be improved is by performing some kind of post-processing, such as 'path short-cutting'.

    b. Maybe include some sampling heuristic so that the two trees can connect in less iterations, thus reducing the number of nearest neighbor calculations

Example Run: (RRT from 2 to 1)



5 Extra Credit

Test for extra credit objectives

Start configuration - 1

Goal configuration - 4

Repeated tests (~10) were done in this configuration to get the extra metrics and data for RRT* sampling heuristics

5.1 Additional Statistics

Here, the data is shown as "Average(Standard Deviation)" for the repeated tests.

The first comparison done is the path quality where the path quality for RRT* is for the final plan.

The second comparison is done for time to plan where the time for RRT* is for the initial plan

Cost and Time for 4 planners

Planner	final plan cost	initial plan time
RRT	12.26(3.98)	2.26(6.17)
RRTC	11.5358(3.17)	**0.0119(0.0144)**
RRT*	**5.7232(0.25)**	0.042(0.095)
PRM	8.749(0)	3.68(11.6)

5.2 Sampling Heuristics

The Local Bias sampling heuristics and Node Rejection was implemented in the RRT class.

In planner.cpp

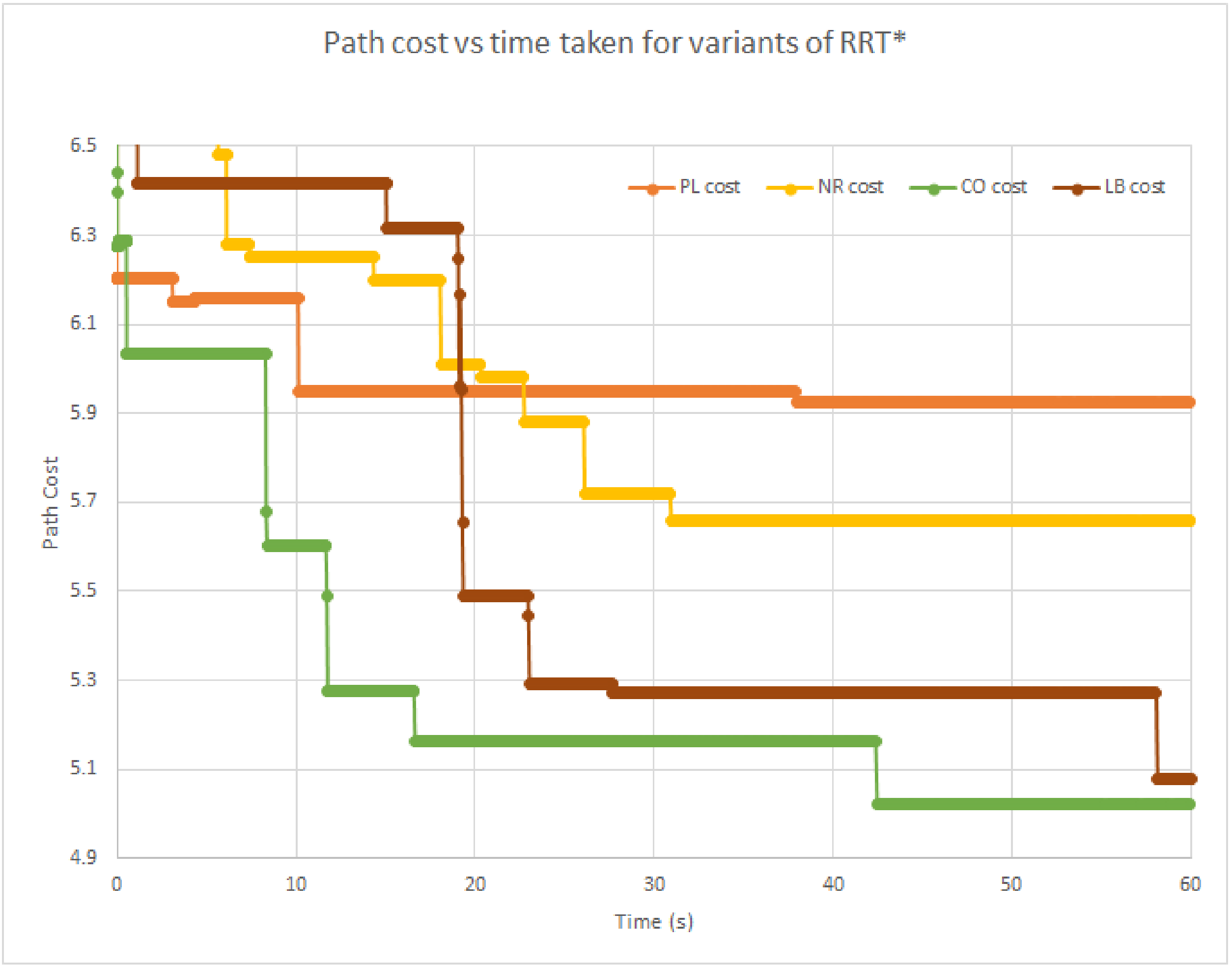
- To enable local bias comment line #341 and uncomment #342 (reverse of this to enable simple goal biasing)
- To enable node rejection uncomment lines #344,#345,#404 (comment to disable)

Results:

Here, c1 is the cost of the initial path. c_end is the cost of the final path and Iterations is the number of times a node was added to the tree. The data is represented as 'Average(Standard Deviation)'

Sampling Heuristics

Aa EC2	c1	c_end	Iterations
Vanilla RRT*	8.31(1.51)	5.7232(0.25)	7471(97)
NR	7.54(1.04)	5.4276(0.26)	7447(1460)
LB	7.56(1.64)	5.305(0.157)	7046(880)
CO	**7.13(1.31)**	**5.2522(0.14)**	**6548(1323)**



Plot for one of the runs