COMP3702/COMP7702 Artificial Intelligence (Semester 2, 2020)
Assignment 2: Continuous motion planning in CANADARM

Name: Bosheng Zhang

Student ID: 45004830

Student email: bosheng.zhang@uqconnect.edu.au

Note: Please edit the name, student ID number and student email to reflect your identity and **do not modify the design or the layout in the assignment template**, including changing the paging.

Question 1 (Complete your full answer to Question 1 on the remainder page 1)

The robot's ***configuration space C*** is the set of all allowed (possible) robot configurations. The dimension is the number of parameter necessary to uniquely specify configuration which depends on the ***number of segments n*** because it tells how many rotational joints in the robot. For the robotic arm, we require ***parameters {eex, eey; -180 < a_1 < 180, -165 < a_2, …, a_n < 165, min_{length} < l_1, l_2, …, l_n < max_{length}}*** to completely specify the position of the arm in the world.
The ***free space F*** ⊆ C is the portion of the free space which is collision-free. The **occupied space O** ⊆ C is the set of all configurations in which the robot collides either with an obstacle or with itself (self-collision) which made up of obstacle list. The goal of motion planning then, is to find a path in F that connects the initial configuration q_{start} to the goal configuration q_{goal}.

The method I used for searching is ***Probabilistic Road Maps (PRM)***. The idea is to take random samples from C, declare them as vertices if in F, try to connect nearby vertices with local planner. We start with uniform sampling to generate sample points in C, then try to make into a robot configuration to determine whether they are valid. If valid and not collides either with an obstacle or with itself, it will be added to the node list. Then the distance between the sample point and other nodes in the list will be checked. When the distance is less than a certain limit, these two nodes will add to each other to become neighbours, indicating that they are connected in the search tree. Repeat this until N samples are created, and then apply a breadth-first search algorithm to find a path from the initial state to the target state. In order to check whether the path between two sample points is valid, the path will be discretised. It will be divided into several smaller segments, and for each segment, verification checks will be applied to ensure that there are no invalid configurations in the path. Verification inspection includes self-collision inspection, obstacle collision inspection and boundary inspection.

Once the neighbours are connected, we perform a Breath First Search (BFS) algorithm to find a list of configurations that form a path between the initial and the target. In order to meet the requirement that the primitive step is less than 0.001 rad, we need to interpolate between the configurations in the returned list.

Question 2 (Complete your full answer to Question 2 on pages 2 and 3, and keep page 3 blank if you do not need it)

The main strategy can be summarized into sampling, connecting and searching.
Strategy structure can be summarized in following pseudocode:

```
for i := range[1, N] do
    configs = uniformly sampling();
    for config in all configs do
      if distance between c1 and c2 is below certain amount and path is valid do
        add a neighbor relationship between nodes;
search graph
if goal reached:
    break
```

Sampling

In order to generate a random configuration, use *random.uniform()* to uniformly randomly sample the angle and length. For uniformly randomly generated angles and lengths, the generated angles are between *-165° and 165°*, and the generated length are between *minimum and maximum length* of arm. After all angles and lengths are generated, verification inspection is applied to ensure that the configuration does not collide with itself, obstacles, and boundary. verification inspection is performing by using test_obstacle_collision, test_self_collision, and test_environment_bounds.
As for efficiency, I generate 1000 samples for cases with less than 3 grapple points, otherwise, 2000 is generated.

for multiple grapple points

For test cases with multiple grabble points, we need to find the bridge configurations between the two grabble points so that the arm can follow the path. In order to extend the single-grapple problem, we decompose the entire search problem into several steps, each step has its own initial state and target state. Since the status of the bridge is both ee1-grappled and ee2-grappled, the status of the bridge is obtained and used as a partial-goal for each step.
Since there are infinitely many possible bridge states between two points, we make a list of bridge configs so that the program has more chances to find a path. In my approach, 10 bridge configs are created (obtained through experiments). We can get the status of the bridge configs through the following steps:
1) Generate the same uniformly randomly sampled sample as above but with one less angle and length. This give us the first n-1 links of the arm.
2) Because we know where the other end of the arm is, then we can draw a line from the last joint generated to the grapple point targeting. We can find the angle of this line relative to link n-1 (using tan(delta_y/delta_x) and the length of this line (using sqrt(delta_x^2 + delta_y^2)), and use them as the last angle and length of RobotConfig. If the last angle and length are valid (and not colliding with anything), then you have a valid bridge configuration; otherwise, discard the configuration and restart.
3) repeat the process N times to obtain a list of bridge configs and make the elements as the goal state of the current search problem.
The pseudocode:

```
start point (x1, y1) is the current grapple point
goal point (x2, y2) is the next grapple point
create an empty sub goal list to keep the goal we find
```

```
for i := range[1, 10] do
    for j := range[1, num of segments - 1] do
        sample some lengths and angles
    if grapple point is even order do
        partial config = generate configuration (start point, length, and angle)
        if the distance between last sample point and goal point is within the length limit do
            calculate the angle and length of the last segment
            use the parameters to generate a completed configuration q
        if q is collision free do
            add the configuration into partial-goal list
    else do the former steps but with ee2 grappled at the grapple point
```

Note: In some cases. (e.g. Figure 5.4g4_m2), we don't need to connect all the grab points. The strategy I use to reach the target state is to first check whether the target state also grabs ee1. If so, we can decide on an intermediate grab point to connect and avoid the other.

Connecting

Add the neighbour

Each *valid node q* needs to be connected to the node list that contains all the previously generated sample points and other configurations of the initial state and the target state. Perform *distance and interpolation checks* to see if q can be added to neighbours of existing nodes.

Distance check

Check the distance so that the two configurations are under a certain limit in C. The distance check function is used to verify whether the distance between q and other configurations meets the requirements. Calculate the angle and length difference between two identical sequences. If the differences are lower than 0.85 (rad for angle), the two nodes will be connected as neighbours. If the limit is higher than 0.85, the path is more likely to have path problems, and more checks are needed to ensure that the path does not collide. If the limit is less than 0.85, more sampling points are needed, otherwise the path may not be found. The threshold of 0.85 is obtained through experiments.

Interpolation check

In order to test whether the path between two sample points is valid, the path is discretised into 20 segments, and verification inspection will be applied to each smaller segment to ensure that there are no invalid configurations in the path.

Pseudocode of path check:

```
q <- config
list x <- distance between angles
list y <- distance between lengths of segment
for j := range[0, 20] do
    for n := 0 to number of segments do
        new angles[n] = q.angles[n] + x[n]*0.05*j
        new lengths[n] = q.lengths[n] + y[n]*0.05*j
    using new angles and lengths to construct new configuration c
    if c is invalid -> return false
return true
```

Question 3 (Complete your full answer to Question 3 on page 4)

For this question, you should discuss what types of environment characteristics (e.g. single or multiple grapple points, small or large number of arm segments, number of obstacles, narrow passageways, passageways with corners, grapple points that must be visited out of order, etc) make a testcase easy or difficult to solve with your solver (and relate this to the design of your solver - e.g. do you have a sampling strategy which is specialised for addressing a particular environment feature, is there a case where your connection strategy won't work correctly, etc).

testcases	3g1_m0	3g1_m1	3g1_m2	3g2_m1	3g2_m2	3g3_m1	4g1_m1	4g1_m2	4g3_m1	4g3_m2	4g4_m1	4g4_m2	4g4_m3	5g3_m1	5g3_m2	5g3_m3
pass	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	2/3	1/3	1/3	1/3	3/3	3/3	3/3
Time	15.3	15.6	14.0	45.9	58.4	50.1	25.2	27.2	85.4	101	117	108	121	52.4	63.2	69.4
steps	2553	5957	5957	5957	7659	9361	5957	6808	11063	17020	15318	22126	23020	18722	22126	21275

My solver can solve the environment with single and multi-grapple points (up to 2) with small and high arm segments (up to 5 - highest from testcases) safely. It sometimes fails in cases with 3 or 4 grapple points, but still have a large chance to find a valid path from initial to goal configuration within 3 attempts.

As the table shown that with the grapple point number, arm segments number, and map complexity like number of obstacles increasing, the time for solver to find a valid path and failure probability for finding valid path increases proportionally. The strategy I use when solving a larger number of arm segments problems is to generate more samples to help me find an effective path, but it will lead to an increase in time. In order to solve the strategy of multiple grabble points, I first define the first grab point as the starting state, and then define the next grab point as the partial target state to find the bridge configuration that both ee1 and ee2 have grappled. Repeat this process on each successive neighbour to form our valid search path.

For environments with equal or greater than 3 grapple points, the failed attempt is mainly caused by three errors:

1. Path cannot be found either the bridge config in each partial step or the path config from start state to goal state.
2. Collision occurred in some primitive steps
3. Exceed the time limit of 2 mins

Here are a few issues that may lead to the failure:

- Testcase with narrow passageway (e.g. figure 2.3g2_m2) or passageways with corners (e.g. figure 6.4g1_m2 or figure 7.5g3_m3) with complicated actions need to perform, if insufficient samples are generated, which may result in path not being found. Since we are performing random sampling, the point is uniformly distributed. This situation needs the sample points are aggregate in the passageway region in order to find a valid path.
- Testcase with small obstacles, it may fall to detect. As for efficiency, the interpolation check discretised the path into 20 segments. If the environment has smaller obstacle than my interpolated part, the collision will not be detected. To solve this problem, we can increase the number of segments in each path to check for collisions, however the running time will increase proportionally, causing the running time to exceed the time limit. The same result applies to the sample size, a larger sample size increases the probability of finding a path, but also significantly increases the running time.

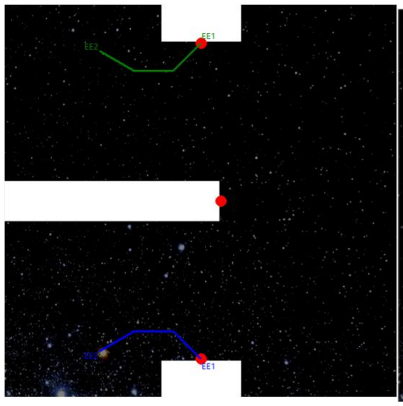

Figure 1. 3g3_m1

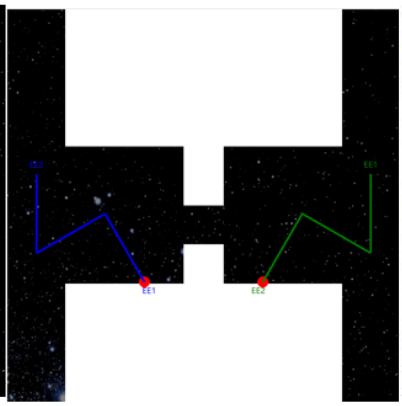

Figure 2. 3g2_m2

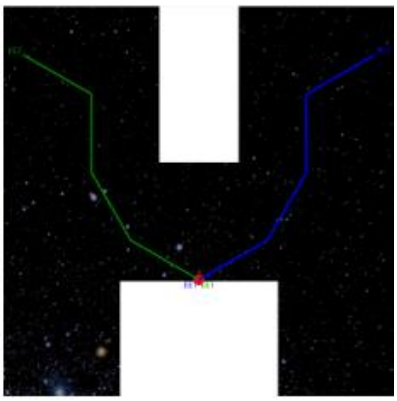

Figure 3. 4g1_m1

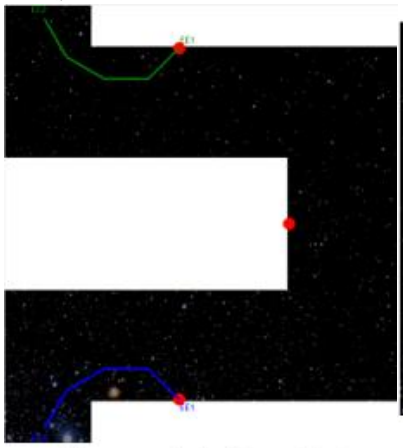

Figure 4. 4g3_m2

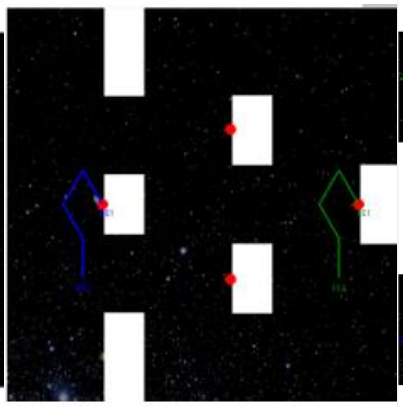

Figure 5. 4g4_m2

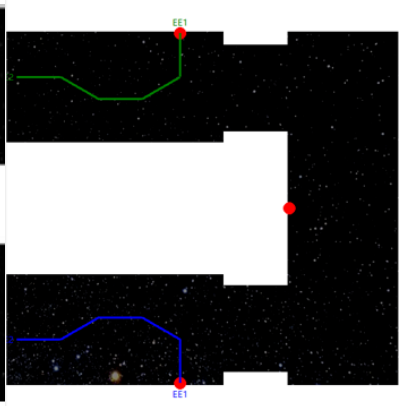

Figure 6. 5g3_m3

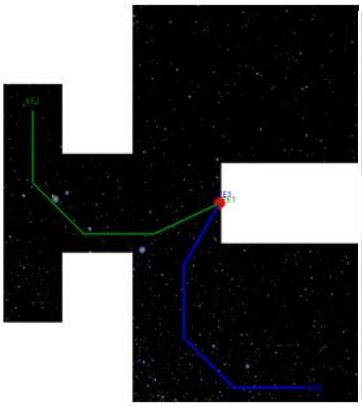

Figure 7. 4g1_m2

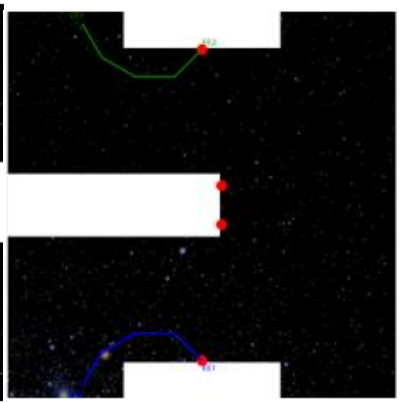

Figure 8. 4g4_m1

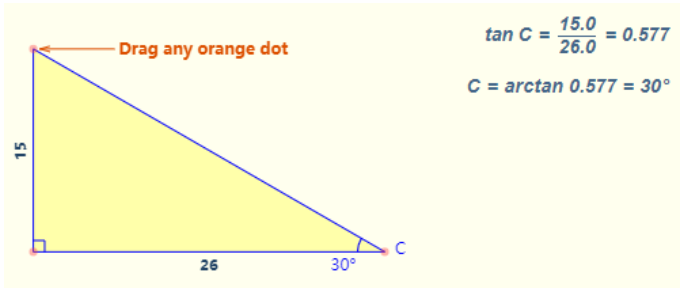

Figure 9. arctan