**Agent for continuous space**

**Part – 1**

***Configuration Space of the problem***

Configuration space is essentially, the state space for motion planning, or the transformations that can be applied to the robot. The configuration space would be all possible configurations that the robot arm can take.

For our use case, the robot configs are dependent on the angle and the length of the arm. The arms also have variable length, the C-space can be built with the angle and armlength.

According to the environment and the requirements for the given problem, we need to take care of the following things for a valid configuration:

1. The robot should not collide with itself
2. The robot should not collide with an obstacle in the environment
3. Should not move out of the bound
4. Length of the config is lesser than minimum length and greater than the maximum length from the spec
5. The angle is not less than 15° for any arm

The configuration space is essentially, every configuration that satisfies the above requirements.

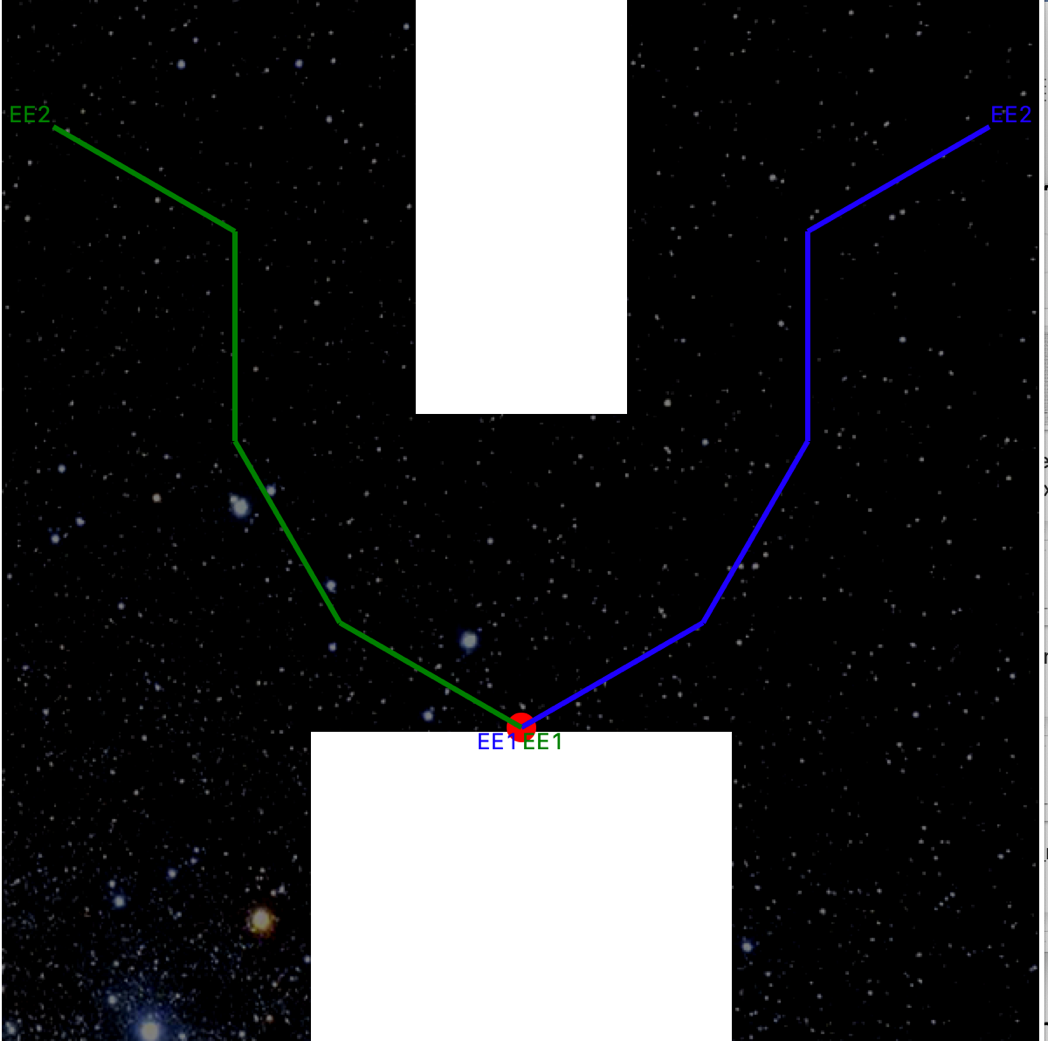
***Searching Continuous space***

To search in the continuous space, since we had length and angles of the arm, we discretised the space with circles.  For instance, the visualisation below can be seen as 4 circles of colour yellow. The centre is the starting point of the arm, radius is the length of the arm.

Similar, the whole space can be discretised as circles (orange) circles. Doing this gives us the following benefits:

1. Random angle changes can be made
2. Length of the rod can be maintained easily as it is radius of the circle
3. Since our discretisation is a circle, we can use various geometric rules for calculation of random samples as will be shown in next part

Figure – 1: Discretization of space



**Part – 2**

***Valid Configuration***

As discussed above, the valid configuration is one with:

* No collision within itself
* No collision to the obstacles
* Length of each arm between minimum and maximum length
* Angle should not be tighter than between any arm

***State Graph Generation***

Generate a state graph, the nodes of these state graph will essentially be configurations. Neighbouring configurations will be added to a given node if they pass the neighbour checking test. For any two configurations, we do not add all the configurations that lead to a collision free path, we just check if two points are neighbours and add them as neighbour. This is done so that the state graph does not explode.

To optimise the solution, for every sample we have, we keep on adding its neighbours to the sample so far and visa-versa

***Discretization Sampling Strategy***

Now, to sample in the continuous space, we followed the following steps:

* Let’s say, we have a configuration of 3 arms
* For this configuration, we have minimum length, maximum length and angles between each point
* From the 3 arms (A,B,C), we choose a random arm, say B. To B, we added a random angle between -20 degrees to 20 degrees and we choose a random length from minimum length to maximum length of B
* We then, checked if the new configuration thus formed is valid, if its valid we added it to our graph. If not, we ignore it
* We sampled 100 such points in one iteration
* To add these samples to the state graph, we check if these samples could be neighbours

***Neighbour checking***

Ones we have two points, to add them as neighbours in the state graph, we need to see if we can get a collision free path from one sample to another. To check that, we follow the following step:

* Check if the two samples have distance less than the tolerance level of . If yes, then these two samples are neighbours
* If not, then, get a mid-point between these two samples,
  + Check if this mid-point has distance less than the tolerance level of
  + If not, sample points between the 3 points (two samples and one mid-point)
  + Repeat till we get all the points that have distance less than the tolerance
  + Terminate, if we after 100 times

***Robot configuration from a random point***

For a random point (blue cross) as shown in the figure, we need to make a configuration such that it is a valid configuration.

For 3 arm configurations, we made use of the geometry between points. For a given grapple point marked in red, the arm of length can move in the red circle, and a given random point marked in blue, the arm of length can move in blue circle. Let’s say, from a point in red circle, we put the third arm (black) this arm can move around the points in black circle and radius is . The intersection between the black and the blue circle is a valid configuration, as blue circle is the path of blue arm and the black circle is the path of black arm, while red is already constant. The final configuration is represented by green line path.

**C:** Intersection point

**D:** Random Point for which we need config

**B:** Random point selection from initial point

**A:** Initial Point

The algorithm for 3 arms configurations, is as follows:

* To get the points in a circle, we are using complex numbers as it generates any number of points, and it’s not only easy to get co-ordinates but also angle from a complex number
* From the initial point (**A**), draw 100 points at a distance (this length can be any length between the maximum and the minimum length)
* To get this random point, we are making use of complex numbers
* From these 100 points, draw a random point (x,y) (**B**)
* From the point that we wish to get a valid configuration (**D**), draw a circle, from that point of length i.e blue circle
* From point (**B**) build the black circle such that it intersects the blue circle
* The intersection points will give the fourth point of the configuration (**C**)
* From these points (A, B ,C,D), build a configuration and check if its valid
* We just need one valid configuration at one time

For 4 arms,

* We can sample a random point from the first red circle, and then subsequently use 3-arm solution

For 5 arms,

* We can sample a random point from the first red circle, and then subsequently use 4-arm solution

***Near obstruction sampling Strategy***

For simple test cases, such as 3g1\_m1.txt or 4g1\_m1.txt, we were able to get to a solution just with Discretisation sampling strategy, however, as soon as the environment became complex, like in 4g1\_m2.txt, our model was failing miserably. This was because one or the other mid points of the samples that were neighbours were falling in the obstacles. We needed something that could sample near the obstruction. With the help of robot configuration above, we sampled a few samples near the obstruction.

The steps that we followed are:

* Between two edges of the obstruction, sample 5 points
* Make a concatenated list
* For each point in this list, see if we can generate a robot configuration
* Do not try more than 50 times to make a robot config

***Between Passage sampling Strategy***

For various configurations like 4g2\_m1.txt, we need a way to go through the passage, for sampling between the passage we followed the following steps:

* Between each two edges of the obstacle, sample 2 points between them
* Many of these points will be inside the obstacle, but we will take care of that while making a valid configuration.
* Additionally, we also sample with the boundary points ([0,0], [0,1], [1,0], [1,1]). This helped us in sampling for configurations like 4g3\_m1.txt

***Remove unwanted points***

Since, we are sampling random points, near obstruction and between passage, we need to check if the points are valid or not, we are doing two quick checks

* Firstly, the distance between of this new sampled point and the grapple point, should not be greater than sum of maximum length of rod
* Secondly, the distance should be greater than the minimum length of configuration, formed by either +15 or -15 angles between the rods.

***Connection Strategy***

Two points that are neighbours can be connected. We took care not to include the neighbours to the state graph as it was increasing the size of graph nodes to manifolds. Ones we have a path till the goal state, we got the neighbours between the points.

***Optimisation during State Graph creation:***

To optimise the neighbour state graph generation, we check for possible neighbours as soon as the node is created instead of checking for the possible neighbours once all nodes are generated. It changed the time from O(n2) to O(n log n)

[ABCD]

A-BCD

B-ACD

C-ABD

D-ABC

[AB]🡪 B-A

[ABC] 🡪 C-AB

[ABCD] 🡪 D-ABC

***Output:***

The below is the time taken by solution with and without obstruction & passage sampling.

|  |  |  |
| --- | --- | --- |
| **Test Case** | **With Obstruction and Passage Sampling(secs)** | **Without Obstruction and Passage Sampling (secs)** |
| **3g1\_m0** | 31.82408666 | 33.80501103 |
| **3g1\_m1** | 16.25316977 | 15.31164813 |
| **3g1\_m2** | 142.4343402 | 8.506937027 |
| **3g2\_m1** | More than 120 secs | 37.30695486 |
| **3g2\_m2** | 44.94112396 | More than 120 secs |
| **3g3\_m1** | 109.9467001 | 51.38534021 |
| **4g1\_m1** | 56.98339987 | More than 120 secs |
| **4g1\_m2** | 109.9467001 | More than 120 secs |
| **4g3\_m1** | 251.1175039 | More than 120 secs |
| **4g3\_m2** | More than 120 secs | More than 120 secs |
| **4g4\_m1** | More than 120 secs | More than 120 secs |
| **4g4\_m2** | More than 120 secs | More than 120 secs |
| **4g4\_m3** | More than 120 secs | More than 120 secs |

* Due to the sampling techniques near obstructions and between passage, we were able to solve many use cases, which we couldn’t solve by only discretisation sampling
* We also see a huge reduction in time, especially with more complex test cases i.e. the test cases with more obstruction

**Part – 3**

***Program Analysis:***

1. The program does not have a fall-back strategy. Since we use random sampling to pull configurations, we do not have a fall-back strategy to use a new random sample point after traversing through certain number of nodes. We do have a limit to terminate the search after a certain number of iterations, but it would never refresh the state the graph and start fresh with a new set of samples.

2. We used Random sampling for simple maps and a strategy to sample configs near the edges of obstructions and bounds to handle complex maps. We do not have a specific weight assigned and we use equal samples from both the strategies. This would cause unnecessary samples to be added to the graph which increases the state graph size.

To reduce the number of samples, we wanted to build a program which pull equal samples from both strategies first. Then based on the number of valid neighbours from each strategy, we wanted to give weightage to the strategy being used. This would ensure that only useful samples are created.

3. Our program would also does not check if the initial and goal states have the same grappled point. From the test cases, we could figure out that the number of grapple points needed to solve a map is

Odd – If both initial and goal have the sample arm (ee1 or ee2) attached to the grapple point

Even – If both initial and goal have different sample arm (ee1 or ee2) attached to the grapple point

So right now, if there are three grapple points it would try to connect from point to 2 and from point 2 to point three. It would never check if the goal is possible from just point 2 but always try to move to point 3.

4. When we are trying to bridge between the initial and the goal states, we never try a new sampled bridge after a specific number of iterations. We wanted to check for 5 iterations and if we couldn’t find the path between the two states, we wanted to try a new bridge state. We couldn’t implement this due to time constraints.

5. Our program works till 4 arms. It does not solve the 5-arm problem since we did not build a strategy to generate a bridge between two points with 5 arms. If we had time, we would have used angles to build the initial 4 arms and used the length to calculate the final arm dimensions.

6. Our strategy does not handle 4 grapple points. It works till three grapple points and once it reaches the third point it, tries to search for a solution within it.

**Next steps:**

1. Implement a fall strategy so that after 5 iterations we sample a new set of points and start the search from scratch.
2. Use random sampling strategy and edge based strategy initially and start giving more weightage to the one performing well.
3. Check the number of points needed to solve a problem spec. If the number needed is 1 less than the available grapple points in the spec, make to try possible combination with the list of grapple points ignoring one grapple point in each iteration.
4. Build a strategy similar to the 4arm config generation for 5 and 6 arms
5. Update the program to be robust so that it can handle any number of grapple points.