FINAL PROJECT
ALGORITHMIC MOTION PLANNING
ASEN 5519
Shrivatsan K.

1. Motivation

For the final project, I wanted to do something which would be a good summary of the course, and also provide a pedestal leading up to solving more complicated implementations and concepts. RRTs are everywhere. Most of the recent research seems to be revolving around RRTs or some modified version of it. Also, RRT is the only algorithm that I haven't implemented (Ignoring the fact that my Bug1 didn't work in Homework1). All other algorithms were covered in the assignments. Therefore, I finalized on using RRT in some way or the other for the final project.

Second part of the Course involved task planning and related methods. I wanted to include task planning as part of my project too since it would give exposure to having different levels of abstraction in planning. Even though the river crossing problem is an easy problem to solve and I didn't have to construct any Automatons or use PDDL, I was more interested in integrating task and motion planning.

Finally, I got the idea for this project while watching the demonstration of a robotic arm. The way it moved blocks around seemed insanely fun. That's when I came up with this proposal - to consider the wolf, goat and cabbage as blocks that the robot arm can move from one position to another.

2. Problem Formulation

2.1 River crossing problem

The farmer has a goat, a cabbage, and a wolf with him. He must cross the river and take all of them to the other side of the river. The problem is that he can only fit one object with him in the boat. If he leaves the goat alone with the cabbage, it will eat it. Similarly, he cannot leave the goat with the wolf. Therefore, the task planning problem formulation is to find a way to get all three objects to the other side, respecting the constraints just mentioned.

2.2 Planning for a Robot arm

Now there's no actual river to cross. I define (2,0) as the Start position and (-2,0) as the Goal position for the arm (Consider the Start and Goal positions as two banks of the river). Using inverse kinematics, this give joint angles as (0,0) and (π, 0). The motion planning problem statement is to find a collision free path for the robot arm holding the object, defined as a square of side 0.5, from Start position to Goal position.

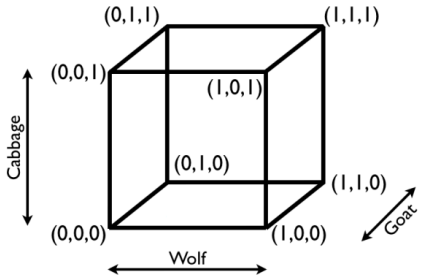
3. Approach

3.1 Task Planning

Let's define a state based on the side of the river the object is in. If it is in the Start Side of the river, assign it state 0. If it is in the Goal side of the river, assign it state 1. Therefore,

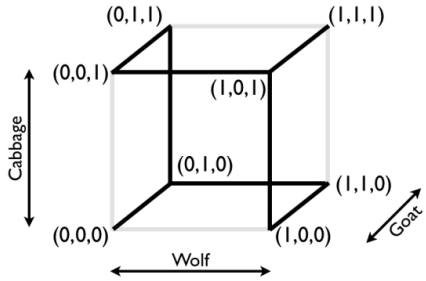
Start State – (0,0,0)
Goal State – (1,1,1)

Where the elements in the tuple represent state corresponding to the wolf, goat and cabbage respectively. Hence, the State Space can be modelled as a 3-dimensional cube, with all the possible states as the vertices of the cube.



State Space Representation (Picture taken from Slide 18 Lecture 18)

As previously mentioned, some of the state transitions are not acceptable such as a transition from (0,0,0) to (0,0,1), wherein the wolf and goat would be left together. Removing all the unacceptable states from our graph, or cube,



State Space Representation (Picture taken from Slide 19 Lecture 18)

Now that we have the graph, all that's to be done now is to find a sequence of transitions or actions to reach from the Start State to Goal State. I used Dijkstra's Algorithm to search the State Space Graph because the algorithm doesn't use any heuristics. Because of the symmetry, size and simplicity of the graph, there is no need of heuristics. If we had options to choose the port on the other side of the river depending on the distance, then heuristics would have made sense.

Pseudo code for my implementation of Dijkstra's algorithm

Initialize each edge with weight one
Initialize a visited list to keep track of visited nodes
Initialize an empty priority_queue
Add start state to visited list and queue
While(priority_queue is non empty):
    If (goal on top of priority_queue):
        break
    else:
        add neighbors of priority_queue[0]
        update neighbors in the visited list
        Increment cost function by keeping track of weights from start state
        Pop top element in priority_queue
        sort priority queue according to cost function

This completes the task planning section for our problem. The workspace consists of a start position, goal position and obstacles. For motion planning, we have to move the items (represented as blocks in the workspace) from start position to goal position, through a collision free region. This task will be achieved by using a 2-link robot arm manipulator.

3.2 Motion Planning

I implemented Rapidly exploring Random Trees to plan for a collision free path (with the end effector holding the block) from Start to Goal. RRT works by first adding the Starting configuration as the root of the tree. It then randomly samples the C-Space to generate a configuration in Obstacle free region. Our goal now is to pull the tree towards this randomly sampled point, which is achieved by finding the leaf in the tree closest to the random sample. We take an epsilon sized step towards the random sample from this leaf. If the sub path between the leaf and configuration reached by taking an epsilon step is collision free, then the branch is added to the tree. The process is repeated until the newly added leaf is within epsilon distance of the goal. The version I implemented is the GoalBiasRRT, wherein the random sample is selected as the goal configuration with a probability of 5%.

Checking for Collision in Sub path

I used Linear interpolation to break down the sub path into a set of 10 waypoints. A sub path is declared collision free if each of these waypoints lie in obstacle-free region. i.e,

$$Path(t) = (1-t)q_a + tq_b,\ t \in [0,1]$$

Where q_a – leaf in tree closest to the random sample
q_b – configuration arrived at by taking an epsilon sized step

Pseudo Code for RRT

Initialize an empty tree
Add start as root
While solution not found do:
    q_{rand} → Get a random sample in collision free space
    q_{near} → Find closest node in tree
    q_{new} → Take epsilon step in direction of q_{rand}
    Check if subpath(q_{near}, q_{new}) is collision free
    If collision free then:
    Add branch(q_{near}, q_{new}) to tree
    If distance (q_{new}, qgoal) < epsilon:
        Return Tree

Once the tree has been generated, it has to be searched to find a sequence of states that will take us from start to goal. I did the search by keeping track of parents of each added leaf, and then using Breadth-First-search traversal on the tree.

Pseudo code for BFS

Initialize an empty queue
Add start to queue
While queue is non-empty:
    Add neighbors of element first in queue
    Save the parent of added neighbors
    Pop the top element
    If goal is reached:
        Trace back to start using parents
        Return path from start to goal

3.3 Task Planning + Motion planning

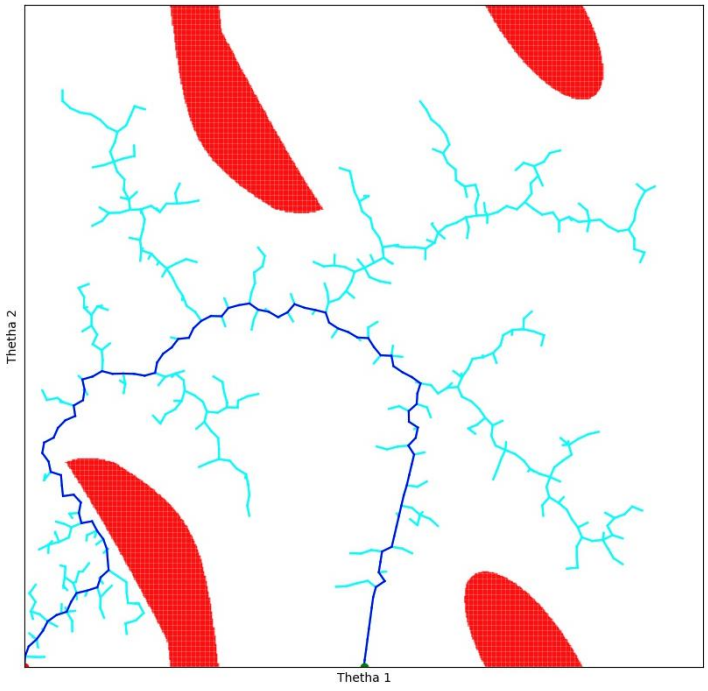
For combining both planning levels, state for the end effector had to be specified, based on which riverbank it was at. This helped to choose whether it needs to pick up a block or go empty handed to the other side and then pick up the block. Consider two adjacent transitions – (0,0,0) and (0, 1, 0). Taking bit wise XOR of each element, only the second bit results in a 1 (0^1 = 1). Therefore, the second bit or the Goat has to change states. Now, since the change is from 0 to 1, this means that the Goat has to be taken from start to goal. If the end effector state was 0, it can pick up the block and proceed to the goal. If the end effector state was 1, it must first comeback to start from goal, and then pick up the block and go to goal. Continuing the process for all state transitions, we get to know the sequence in which objects have to be picked up. After that, it just alternating initial and final configurations in the path found using RRT and traversing it. That is, if the end effector state is 0 – it has to go to goal so initialize configuration is start and final configuration is goal. If the end effector state is 1- it has to go to start so initial configuration is goal and final configuration is start.

Pseudo Code for task and motion planning

```
For ith state in action sequence:
    Object_ transit → XOR (ith state, (i+1)th state)
    If Object_ transit == 1:
        If Object_State[i] = 0:
            If end_effector_State[0]:
                Go to goal
            If end_effector_State[1]:
                Bring End effector to start
                Go to goal
        If Object_State[i] = 1:
            If end_effector_State[0]:
                Bring end effector to goal
                Go to Start
            If end_effector_State[1]:
                Go to Start
```

4. Results

4.1 Configuration Space



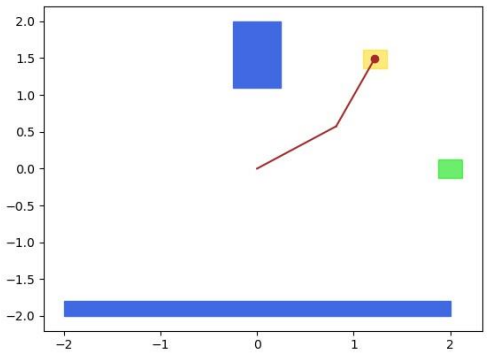
Solution Tree in the Configuration Space

4.2 Benchmarking

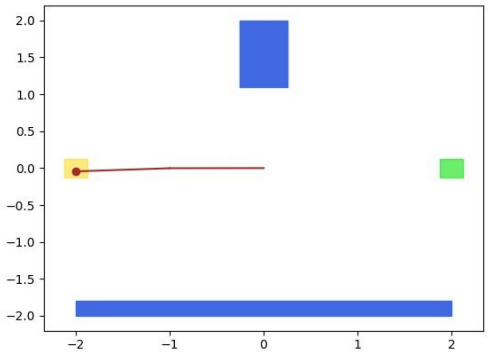
Number of runs: 50
Average Computation time: 69.64 seconds
Average Number of nodes: 1098

I would like to emphasize here that the average computation time is no way near ‘rapid’, but I am quite sure that’s something directly down to my laptop. My laptop takes solid 5 minutes to boot up, which speaks volumes about its speed. The program should run within a matter of seconds in any other decent system.

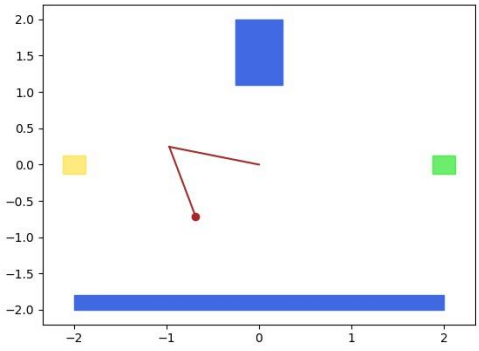
4.3 Workspace



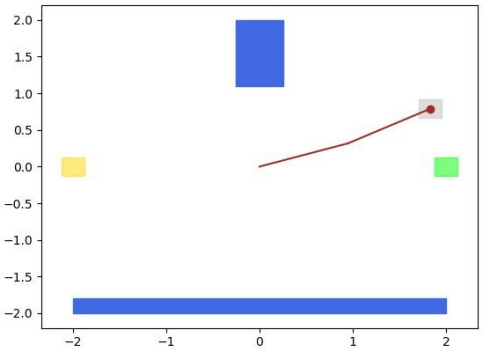
Robot arm picks up the goat first



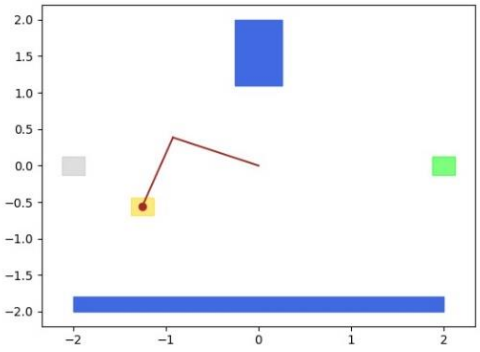
Places the goat at Goal



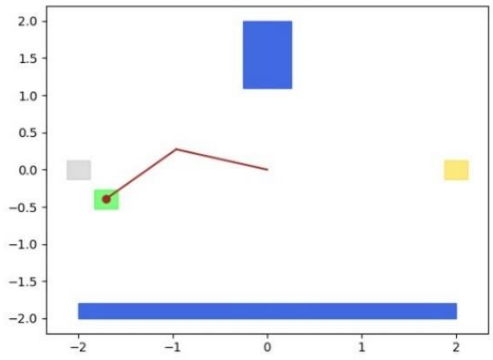
Goes back to pick the next object



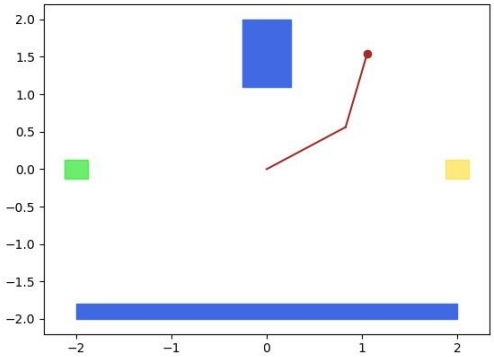
Picks up the Wolf



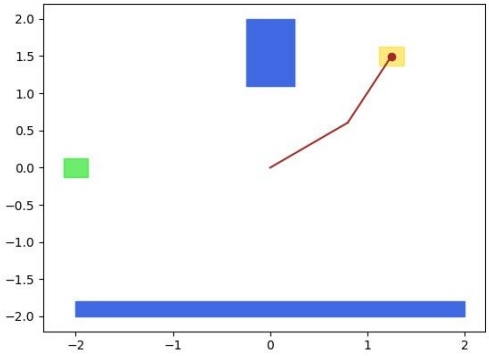
Places the Wolf at Goal and brings back the goat



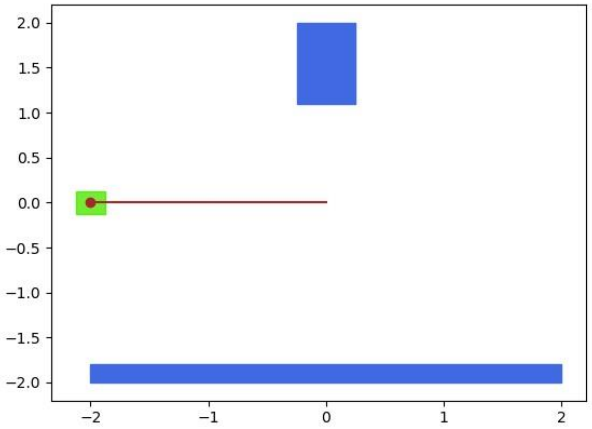
Picks up the Cabbage from start



Goes Back to get the Goat



Picks up the goat from Start



All Objects have made it to the goal with no loss to Flora or Fauna

5. CONCLUSION

Both levels of planning were achieved and integrated smoothly. I have developed more respect for C-Spaces. My RRT implementation was not converging for 3 days, simply because I was not paying attention to the geometry of C-Space (which also indicates that I have a good implementation of RRT). I think understanding and getting the C-Space right was the most important part of the problem since it forms the basis for Sampling points, collision checks, connecting points and so on. I am pleasantly surprised that I made it this far in the course in one piece. I would like to thank you Dr. Lahijanian. The talk we had over not dropping the course did help and I ended up learning a lot of interesting stuff.

Suggested Improvements

- I treated the C-Space as Euclidean whereas it has toroidal topology. To reach from Configuration B from Configuration A, there are 4 possible ways since both links can rotate clockwise and counterclockwise (For example, Link 1 rotating clockwise and Link 2 rotating counterclockwise is one of the possible transitions). Collision and length have to be checked for all of the 4 possible motions.
- For complex environments, more links are required for the arm, otherwise the end effector is not very ‘effective’ at reaching different parts of working space.
- Implementing ConnectedRRT, RRT*, Bidirectional RRT.