RBE 550 Motion Planning

Assignment Advanced Search Algorithhm

Aadesh Varude
avarude@wpi.edu

Q1 . Informed RRT* Explained

- We start the algorithm by finding the path and keeping the c best for path length as 0, once the path is found we update the c best cost as the traversal cost between the start and the goal.
- The sampling is done as RRT* till the c best is 0 or negative then new point is calculated as RRT* is performed, once the path is found the c best is assigned and then the sampling is done based on an ellipsoidal search region.

```
        def sample(self, goal_bias=0.05, c_best=0):
    '''Sample a random point in the area
    arguments:
        goal_bias - the possibility of choosing the goal instead of a random point
        c_best - the length of the current best path (For informed RRT)

    return:
        a new node if this node is valid and added, None if not.

    Generate a new point
    '''
    # Generate a new point

    #### TODO ####
    if(c_best<=0):
        new_point = self.get_new_point(goal_bias)
    else:
        new_point = self.get_new_point_in_ellipsoid(goal_bias,c_best)

    return new_point
```

- The get new point in the ellipsoid to take a distance between the goal and start node and creates an ellipse around the line and center as the midpoint of the line, by computing the rotation matrix of that ellipse to world frame and projecting the unit ball radius samples on to the ellipse

```
def get_new_point_in_ellipsoid(self, goal_bias, c_best):
    '''Choose the goal or generate a random point in an ellipsoid
       defined by start, goal and current best length of path
    arguments:
        goal_bias - the possibility of choosing the goal instead of a random point
        c_best - the length of the current best path

    return:
        point - the new point
    '''
    # Select goal
    if np.random.random() < goal_bias:
        point = [self.goal.row, self.goal.col]

    #### TODO ####
    # Generate a random point in an ellipsoid
    else:
        c_min=self.dis(self.start,self.goal)
        x_center=Node((self.start.row+self.goal.row)/2 ,(self.start.col+self.goal.col)/2)

        # Compute rotation matrix from elipse to world frame - C
        mat=np.array([[(self.start.row - self.goal.row)/c_min],[(self.start.col-self.goal.col)/c_min],[0]])
        indentity=np.array([1,0,0]).reshape(1,3)
        M=np.dot(mat,indentity)
        U,S,V=np.linalg.svd(M,True,True)
        diag=np.dot(np.diag([1,1,np.linalg.det(U)*np.linalg.det(np.transpose(V))]),V)
        C=np.dot(U,diag)

        # Compute diagonal matrix - L
        r1=c_best/2
        r2=np.sqrt(c_best**2-c_min**2)/2
        r3=np.sqrt(c_best**2-c_min**2)/2
        L=np.diag([r1,r2,r3])

        # # Cast a sample from a unit ball - x_ball
        u = random.random()
        v = random.random()
        r = u**0.5
        theta = 2* math.pi *v
```

```
            x = r*math.cos(theta)
            y = r*math.sin(theta)
            x_ball=np.array([x,y,0]).reshape(3,1)
            # # Map ball sample to the ellipsoid - x_rand
            CLx=np.dot(C,np.dot(L,x_ball))
            x_rand=CLx+np.array([[x_center.row],[x_center.col],[0]])

            point=[x_rand[0][0],x_rand[1][0]]

        return point
```

- Later the new point is extended and rewired just as we did in RRT*.

2. D* explained

- D* first looks for a path from the start to the goal by processing the states from goal to the start node.
- Once the path is found the nodes are traversed from the start to the goal and parsed through the prepare and repair function and repair and replan function to tackle the dynamic obstacles.

```
            def run(self):
        ''' Run D* algorithm
            Perform the first search from goal to start given the pre-known grid
            Check from start to goal to see if any change happens in the grid,
            modify the cost and replan in the new map
        '''
        #### TODO ####
        # Search from goal to start with the pre-known map

            # Process until open set is empty or start is reached
            # using self.process_state()
        self.insert(self.goal,0)
        while(len(self.open)>0 or self.start.tag!="CLOSED"):
            k_min=self.process_state()
        # Visualize the first path if found
        self.get_backpointer_list(self.start)
        self.draw_path(self.grid, "Path in static map")
        if self.path == []:
            print("No path is found")
            return

        new_node=self.start
        while(new_node!=self.goal):
            self.prepare_repair(new_node)
            self.repair_replan(new_node)
            self.get_backpointer_list(new_node)

            self.draw_path(self.dynamic_grid, "Path in progress")

            if self.path == []:
                print("No path is found")
                return

            new_node=new_node.parent
```

- The process state function checks the values of k and h of the minimum node and its neighbors and updates the values of the cost function of the min node and its neighbors. It has a RAISE state and Lower State condition for updating the cost values.

```
        def process_state(self):
        ''' Pop the node in the open list
            Process the node based on its state (RAISE or LOWER)
            If RAISE
                Try to decrease the h value by finding better parent from neighbors
                Propagate the cost to the neighbors
            If LOWER
                Attach the neighbor as the node's child if this gives a better cost
                Or update this neighbor's cost if it already is
        '''
        #### TODO ####
        new_node=self.min_node()
        if new_node==None:
            return -1
        k_old=self.get_k_min()
        neighbors=self.get_neighbors(new_node)
        self.delete(new_node)
        if k_old < new_node.h:
            for neighbor in neighbors:
                if neighbor.h <= k_old and new_node.h > neighbor.h+ self.cost(new_node,neighbor):
                    new_node.parent=neighbor
                    new_node.h = neighbor.h+ self.cost(new_node,neighbor)
        if k_old==new_node.h:
            for neighbor in neighbors:
                if neighbor.tag=="NEW" or (neighbor.parent==new_node and
                      neighbor.h!=self.cost(new_node,neighbor)+new_node.h) or (neighbor.parent!=new_node and
                      neighbor.h>self.cost(new_node,neighbor)+new_node.h):
```

```
                neighbor.parent=new_node
                self.insert(neighbor,self.cost(new_node,neighbor)+new_node.h)
        else:
            for neighbor in neighbors:
                if neighbor.tag=="NEW" or (neighbor.parent==new_node and
                      neighbor.h!=self.cost(new_node,neighbor)+new_node.h):
                    neighbor.parent=new_node
                    self.insert(neighbor,self.cost(new_node,neighbor)+new_node.h)
                else:
                    if neighbor.parent != new_node and neighbor.h > self.cost(new_node,neighbor)+new_node.h:
                        self.insert(new_node,new_node.h)
                    else:
                        if neighbor.parent != new_node and new_node.h > self.cost(new_node,neighbor)+neighbor.h
                              and neighbor.tag=="CLOSED" and neighbor.h>k_old:
                            self.insert(neighbor,neighbor.h)
        return self.get_k_min()
```

- The prepare and repair function checks for the dynamic obstacles and insets them in the open list with infinite h value and further checks its neighbors and modifies their cost using the modify cost function, where the modify cost function inserts the closed tagged neighbors with there h values in the list.

```
    def modify_cost(self, obsatcle_node, neighbor):
        ''' Modify the cost from the affected node to the obstacle node and
            put it back to the open list
        '''
        #### TODO ####
        # Change the cost from the dynamic obsatcle node to the affected node
        # by setting the obstacle_node.is_obs to True (see self.cost())

        # Put the obsatcle node and the neighbor node back to Open list

        #### TODO END ####

        if obsatcle_node.tag == "CLOSED":
            self.insert(obsatcle_node,obsatcle_node.h)
        return self.get_k_min()


    def prepare_repair(self, node):
        ''' Sense the neighbors of the given node
            If any of the neighbor node is a dynamic obstacle
            the cost from the adjacent node to the dynamic obstacle node should be modified
        '''
        #### TODO ####
        # Sense the neighbors to see if they are new obstacles

            # If neighbor.is_dy_obs == True but neighbor.is_obs == Flase,
            # the neighbor is a new dynamic obstacle

                # Modify the cost from this neighbor node to all this neighbor's neighbors
                # using self.modify_cost

        #### TODO END ####
        # print("in pr",node.row)
        neighbors=self.get_neighbors(node)
        for neighbor in neighbors:
            if neighbor.is_dy_obs==True and neighbor.is_obs==False :
                if neighbor.tag == "CLOSED":
                    self.insert(neighbor, math.inf) # setting the closed node to initial conditions as the node
                          needs to be reprocessed
                neighbor.is_obs=True # markin dynamic obstacle as obstacle
                # Modify the cost from this neighbor node to all this neighbor's neighbors
                new_neighbors=self.get_neighbors(neighbor)
                for new_neighbor in new_neighbors:
                    self.modify_cost(node,new_neighbor)
```

- Lastly the repair and replan function processes the states using the process state function till the kmin is -1 that the open list becomes empty or the k min becomes larger than the existing node cost value. And gets the parent of the nodes by using the back pointer function.

```
       def repair_replan(self, node):
        ''' Replan the trajectory until
            no better path is possible or the open list is empty
        '''
        #### TODO ####
        # Call self.process_state() until it returns k_min >= h(Y) or open list is empty
        # The cost change will be propagated

        #### TODO END ####
        flag=1
        while(flag==1):
            k_min=self.process_state()
            if k_min >= node.h or k_min==-1 :
                flag=0
        self.get_backpointer_list(node)
```

Explain in your own words, how does D* replan a path by updating the cost?
First, it generates an initial path and assigns all the h and k values to the nodes. Once it has got the basic path it starts updating the path for the optimal path search from the start to the goal, while doing so it also takes into account if any changes are made in the path cell just in case a cell turn into an obstacle then it will process that cell along with its neighbors and all the affected cell due to that one obstacle cell, in our code, we do that using preparerepair function, further the after resetting all the affected nodes cost we using the repairreplan function to replan the optimal path.

Why does D* can replan faster than A* or Dijkstra?
The D* algorithm is faster as it relies on the previous initial path found to update the path to optimality whereas the A* or Dijkstra algorithms re-explore the entire map which make D* better and faster.
By showing and comparing the results of RRT* and informed RRT*, what is the advantages of using the latter?
As shown in the below results one can observe that the search space after getting the initial path is restricted in the ellipsoidal domain and not the entire map this increases the chances of getting the optimal path with each iteration as the search will be based in the region between start and goal and not the entire map hence the Informed RRT* is better.

Results

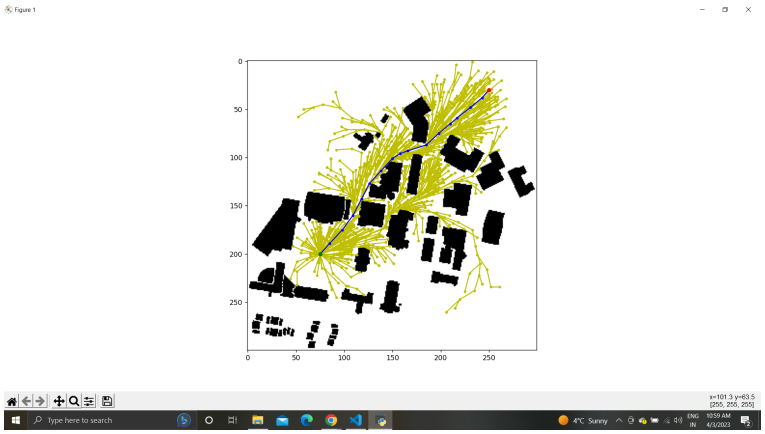

Figure 1: Images of Informed RRT*.

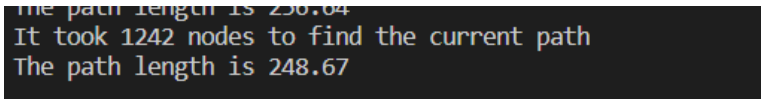

Figure 2: Images of Cost Informed RRT*.

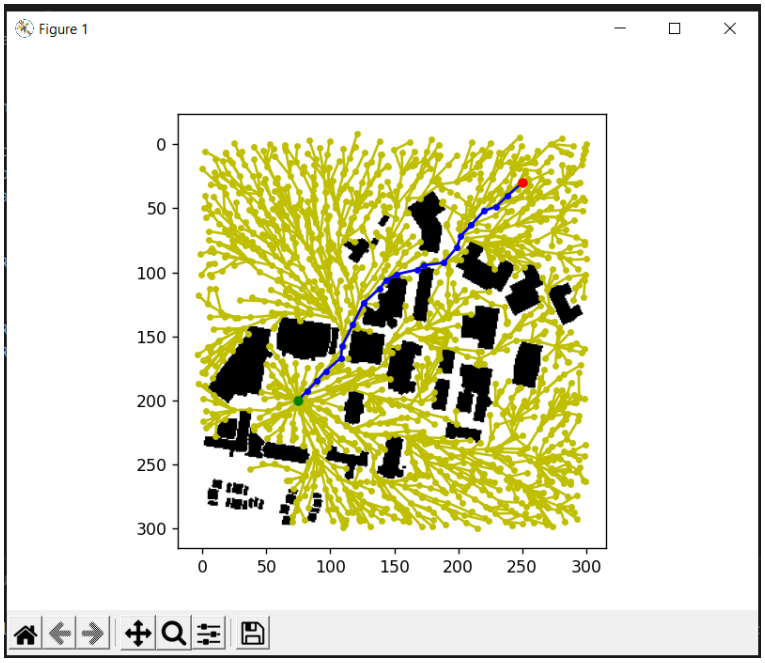

Figure 3: Images of RRT*.

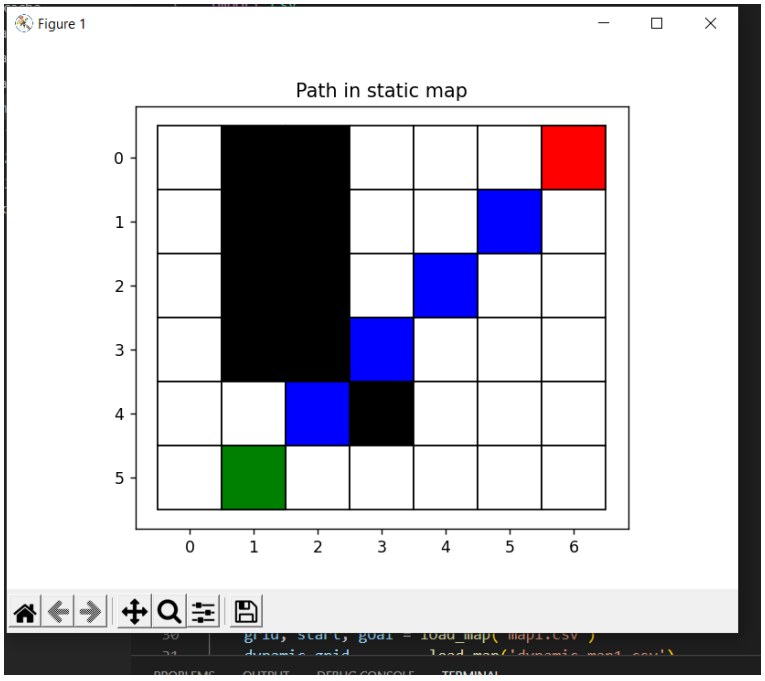

Figure 4: Static path using D*.

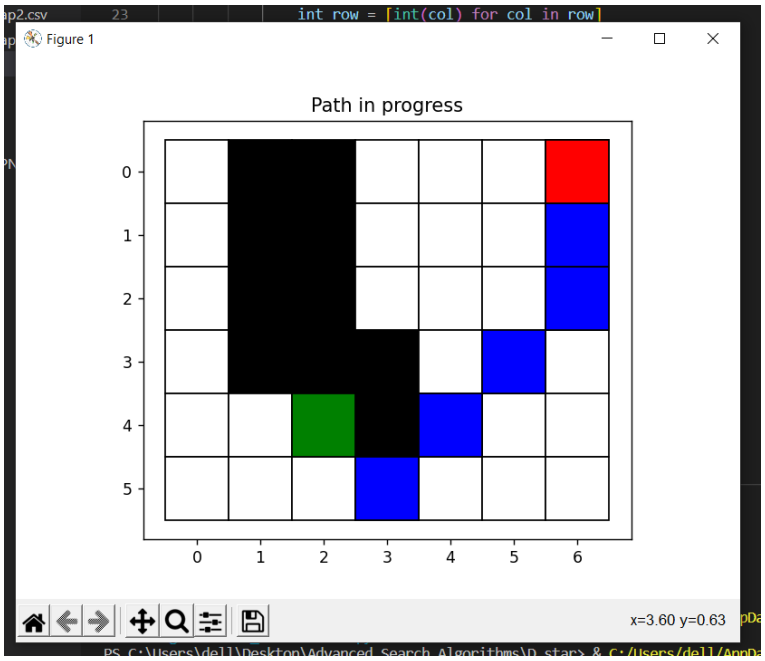

Figure 5: Replanned path after the dynamic obstacle introduction, using D*.

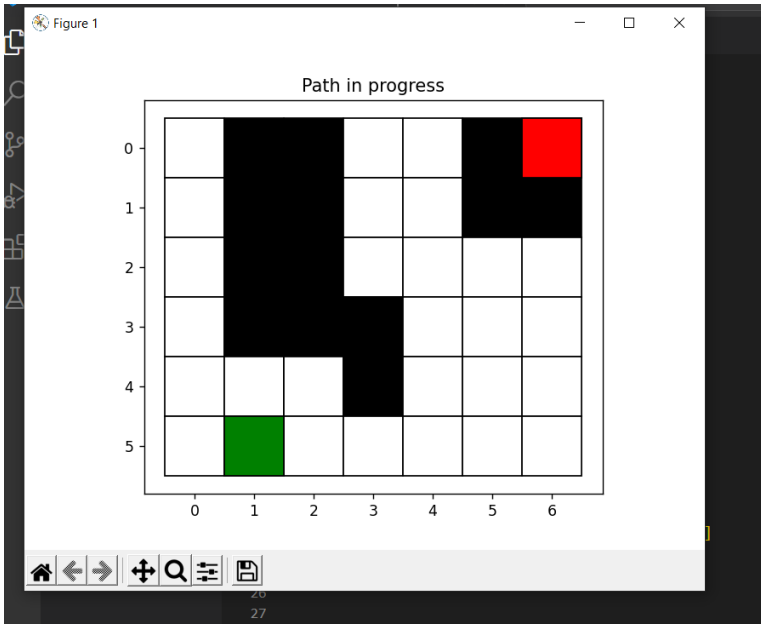

Figure 6: Replanned path after the dynamic obstacle introduction, using D*.

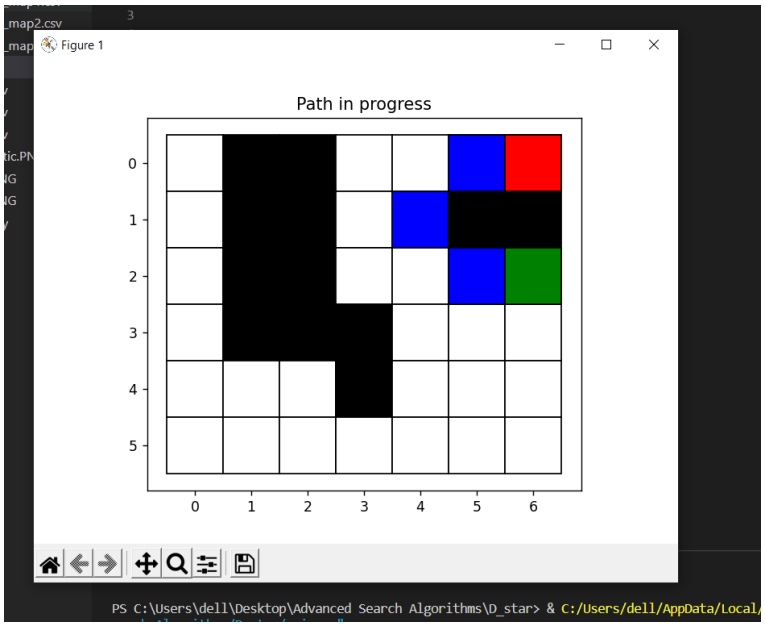

Figure 7: Replanned path after the dynamic obstacle introduction, using D*.

References

1. https://extremelearning.com.au/how-to-generate-uniformly-random-points-on-n-spheres-and-n-balls/
2. https://math.stackexchange.com/questions/2696217/rotation-of-an-ellipse-using-a-rotation-matrix
3. https://github.com/AtsushiSakai/PythonRobotics/blob/master/PathPlanning/DStar/dstar.py
4. https://github.com/AtsushiSakai/PythonRobotics/blob/master/PathPlanning/InformedRRTStar
5. https://arxiv.org/pdf/1404.2334.pdf
6. Class slides for D* and Informed RRT*
7. http://web.mit.edu/16.412j/www/html/papers/original$_{d}star_{i}cra94.pdf$