Homework 1

1. A) 1) Backward Iteration

cost to go to 6 from other nodes in:
0 steps: G5
1 steps: G4
2 steps: G3
3 steps: G2
4 steps: G1

	1	2	3	4	5	6
G5	∞	∞	∞	∞	∞	0
G4	∞	∞	∞	1	5	1
G3	6	2	4	2	6	2
G2	5	3	5	3	7	3

2) Forward iteration
cost to go to other nodes from 1 in:
0 steps: C1
1 steps: C2
2 steps: C3
3 steps: C4

	1	2	3	4	5	6
C1	0	∞	∞	∞	∞	∞
C2	∞	3	2	∞	1	∞
C3	∞	∞	5	4	6	6
C4	∞	∞	∞	8	9	5

1) B) 1. Admissible- the euclidean distance is admissible because its value will always be less than or equal to the distance between start and goal configurations.
   2. Inadmissible - inadmissible because this value is not guaranteed to always be less than or equal to the actual distance from goal

3. admissible - since euclidean distance is admissible, half its value will also be admissible

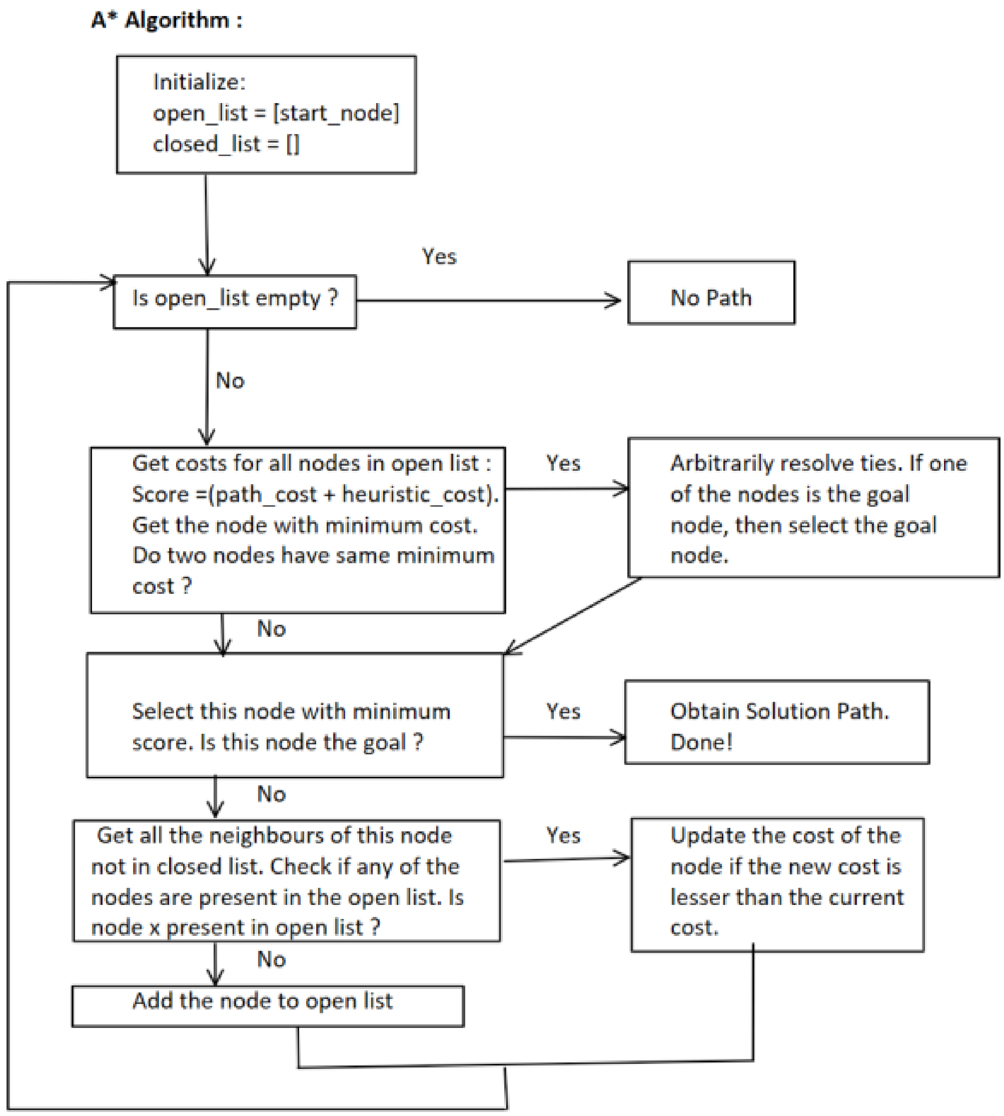
4. Admissible- this is guaranteed to be less than or equal to distance between the configurations as no other path is shorter than the optimal path

1) c) ordering from least informed to most informed
   1. Euclidean distance in 6DOF space from start to goal divided by 2
   2. Euclidean distance in 6DOF space from start to goal
   3. Sum of total 6DOF distance travelled by optimal solution to reach goal state

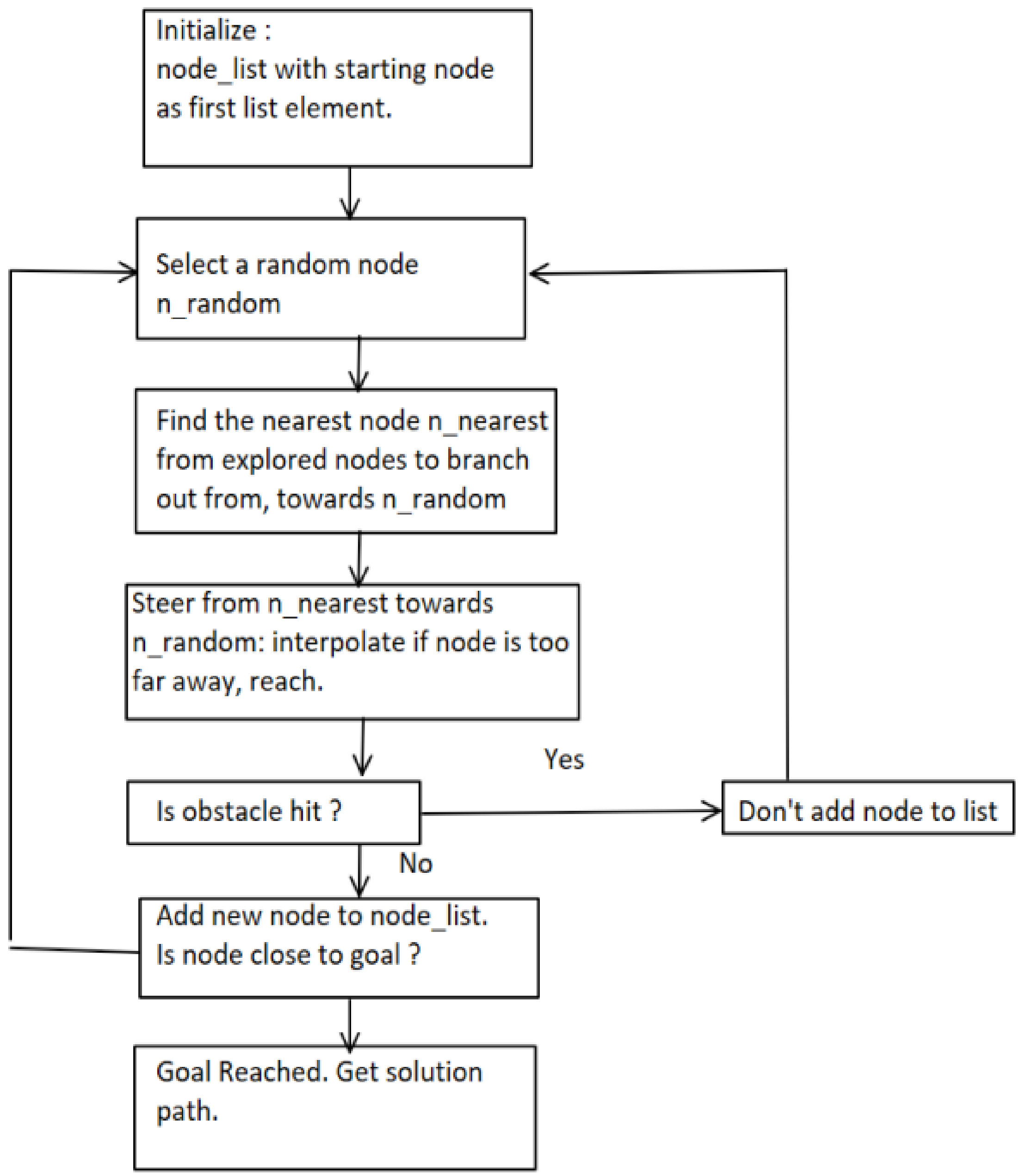
Explanation- the distance traveled by optimal path will have the highest value amongst the three heuristics listed above. Therefore it is the most informed value, as its value will be closest to the actual distance to goal.

2) **Programming Assignment**

Step 1-

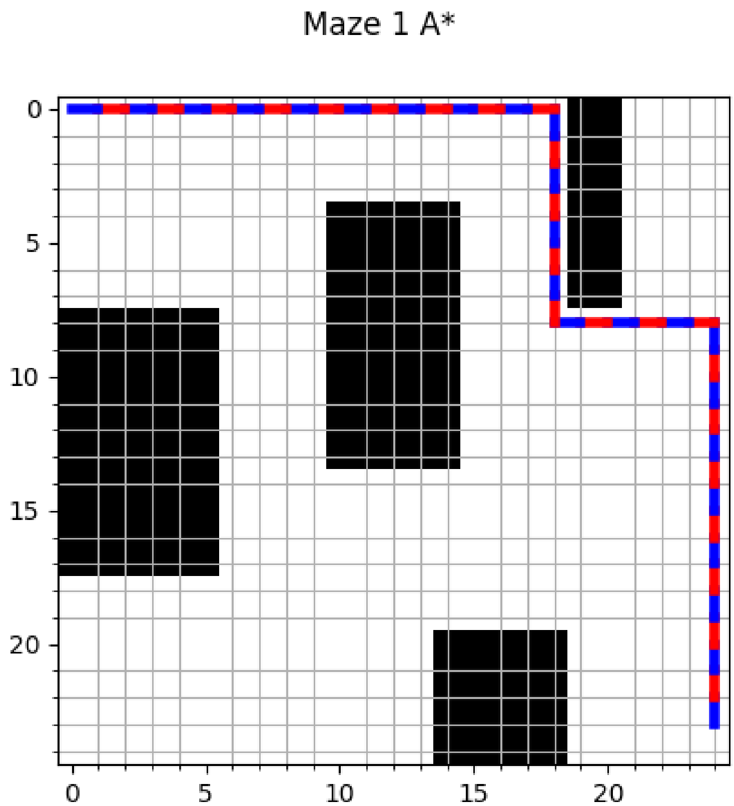


RRT Algorithm :

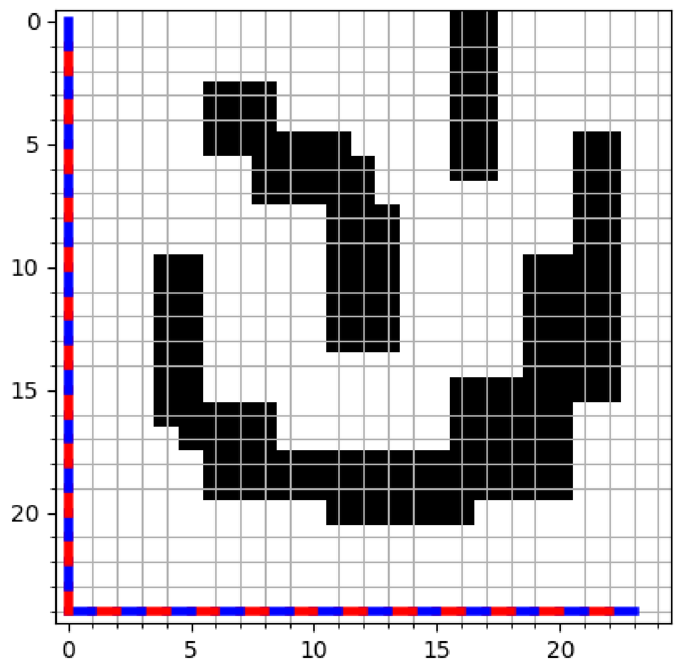


Step 2

1)The heuristic I used is Euclidean distance.



Maze 2 A*



2) A* with epsilon greedy for Maze 1

Time	Epsilon	Number of nodes expanded	Path Length
1	10	235	48
1	5.5	232	48
1	3.25	230	48
1	2.125	226	48
1	1.56	223	48
1	1	418	48
0.25	10	235	48
0.25	5.5	232	48
0.25	3.25	230	48
0.25	2.125	226	48

0.25	1.56	223	48
0.25	1	418	48
0.05	10	235	48
0.05	5.5	232	48
0.05	3.25	230	48
0.05	2.125	226	48
0.05	1.56	223	48
0.05	1.07	360	48

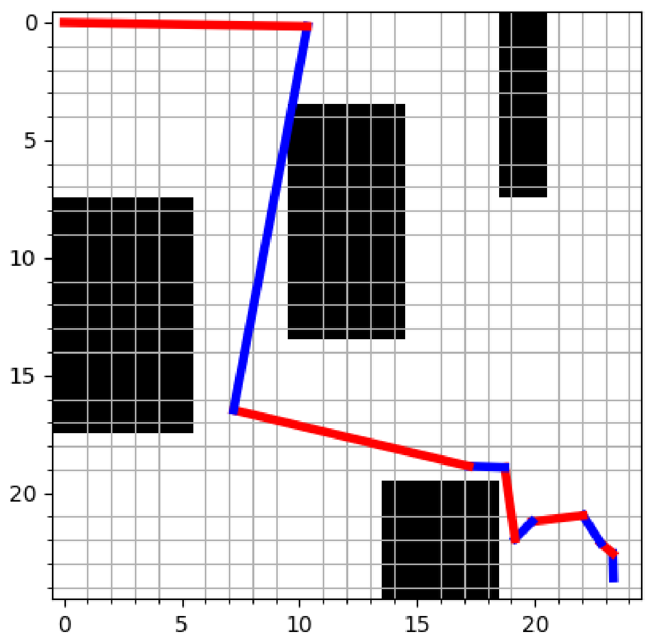
2) A* with epsilon greedy for maze 2

Time	Epsilon	Number of nodes expanded	Path Length
1	10	205	70
1	5.5	205	70
1	3.25	278	70
1	2.125	329	58
1	1.56	320	58
1	1	394	48
0.25	10	205	70
0.25	5.5	205	70
0.25	3.25	278	58
0.25	2.125	329	58
0.25	1.56	320	58
0.25	1	394	48
0.05	10	205	70
0.05	5.5	205	70
0.05	3.25	278	58
0.05	2.125	329	58
0.05	1.56	320	58
0.05	1.07	361	48

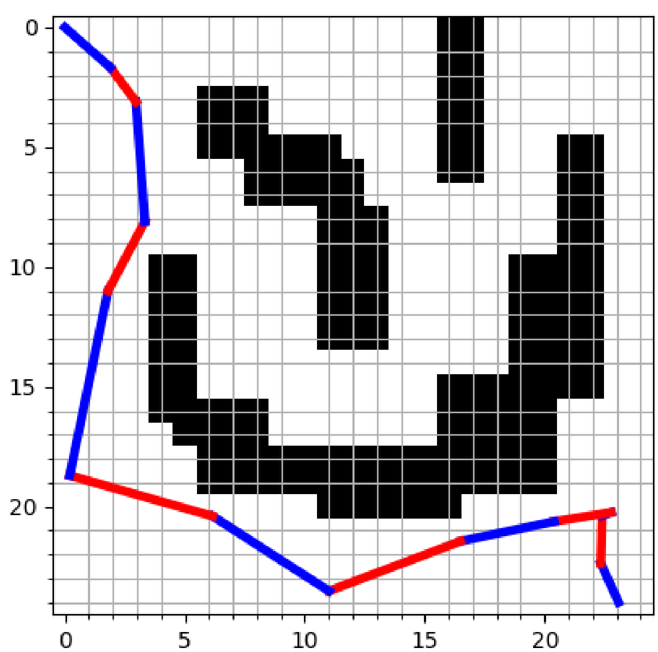
.

3) RRT

Maze 1 Time to solve: 0.04s, Path length: 48.13



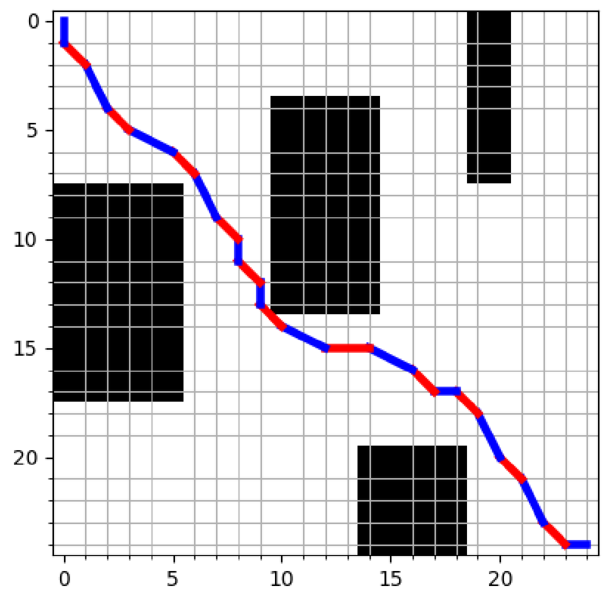
Maze 2 Time to solve: 0.02s, Path length: 48.9



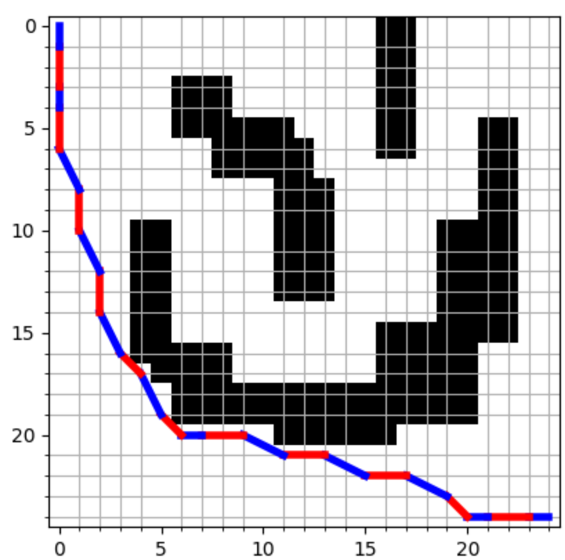
Step 3

1) The heuristic I used is euclidean distance.

4D-Maze 1



4D-Maze 2



2) Maze 1- 4D, A* using epsilon greedy

Time	Epsilon	Number of nodes expanded	Path Length
1	10	2807	39.14
1	5.5	2807	36.79
1	3.25	2807	36.79
1	2.125	2800	36.79
1	1.56	2800	36.79
1	1.28	2806	36.79
0.25	10	2807	39.13
0.25	5.5	2807	36.79
0.05	10	2807	39.13

Maze 2- 4D, A* using epsilon greedy

Time	Epsilon	Number of nodes expanded	Path Length
1	10	2403	42.24
1	5.5	2401	42.06
1	3.25	2417	41.07
1	2.125	2420	40.89
1	1.56	2426	40.89
1	1.28	2426	40.89
0.25	10	2403	42.24
0.25	5.5	2401	42.06
0.05	10	2403	42.24

Discussion Questions

1) The algorithm can be modified by sampling some nodes at a fixed distance smaller from the starting node. Collisions are checked for paths with cost less than a fixed budget. We also check if the goal vertex is present in this queue. If there are no collisions, we consider nodes with least cost to the goal by using a heuristic. For the node with the least estimated cost, we consider that as the vertex and repeat the above steps. If there are collisions, we remove those edges from the graph and reconnect the nodes to the nearest nodes.
2) RRT Connect would be faster than RRT as two trees are generated, one from the start point and other from the goal. However, a key disadvantage is that the number of nearest neighbors searched increases. It is also time consuming to find neighbors in the 4D search space.
3) I would search the environment from the start node, calculate the priorities of the neighbors, select the node with the highest priority, like in the case of regular A*. All the explored nodes will be stored with their priorities in a dictionary. The parent of each node selected so far will be stored while the algorithm is computing. If an obstacle is discovered, the algorithm will backtrack to the last node and select the node with the second best priority.
4) I would sample nodes which have a belief of uncertainty less than a certain threshold value. These belief nodes will be connected to the vertex and added to a queue. The queue will be searched using uniform cost search and kalman filtering.

Code A*

```
import numpy as np
import time
from maze import Maze2D, Maze4D
from priority_queue import PriorityQueue


class Astar():
   def __init__(self, maze, epsilon=1):
       self.maze = maze
       self.start = maze.start_state
       self.goal = maze.goal_state
       self.start_index = maze.get_start()
       self.goal_index = maze.get_goal()
       self.opened = PriorityQueue()
       self.closed = PriorityQueue()
       self.came_from: dict[int, int] = {}
       self.cost_so_far: dict[int, int] = {}
       self.path = []
       self.nodes_expanded = 0
       self.epsilon = epsilon

   # 2D:given index return the Euclidean value of distance to goal
   # 4D:given index return euclidean distance to goal over speed
   # (heuristic calculation)
   def heuristic(self, a):
       if isinstance(self.maze, Maze2D):
           (x1, y1) = self.maze.state_from_index(a)
           (x2, y2) = self.maze.state_from_index(self.goal_index)
           return np.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2) * self.epsilon
       elif isinstance(self.maze, Maze4D):
           (x1, y1, dx1, dy1) = self.maze.state_from_index(a)
           (x2, y2, dx2, dy2) = self.maze.state_from_index(self.goal_index)
           return np.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2) / np.sqrt(
               (dx1 - dx2) ** 2 + (dy1 - dy2) ** 2) * self.epsilon

   # return the g_n from the closed list
   def get_g_n(self, a):
       return self.closed.get_priority(a)

   # return the f_n from the open list
   def f_n(self, a):
       return self.opened.get_priority(a)

   # calculates the cost of going between two neighbors
   def cost(self, a, b):
       if isinstance(self.maze, Maze2D):  # cost of going to a neighbor will
always be one (stated in spec)
           # robot can only go 1 unit in cardinal directions so always 1
```

```
            return 1
        elif isinstance(self.maze, Maze4D):  # cost is euclidean distance
            (x1, y1, dx1, dy1) = self.maze.state_from_index(a)
            (x2, y2, dx2, dy2) = self.maze.state_from_index(b)
            return np.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2) / np.sqrt((dx1 -
dx2) ** 2 + (dy1 - dy2) ** 2)


    # solves A*
    def solve_private(self):
        self.opened.insert(self.start_index, 0)
        while self.opened.pq:
            # pick n_best
            n_best = self.opened.pop()

            if self.closed.pq:  # if things exist in the closed list
                self.closed.insert(n_best, self.cost_so_far[n_best])
            else:
                self.closed.insert(n_best, 0)
            self.nodes_expanded += 1

            # if goal
            if n_best == self.goal_index:
                # finish
                break

            for neighbor in self.maze.get_neighbors(n_best):
                if self.closed.test(neighbor):
                    continue
                if not self.opened.test(neighbor):
                    self.opened.insert(neighbor,
                                      self.get_g_n(n_best) + self.cost(n_best,
neighbor) + self.heuristic(neighbor))
                    self.came_from[neighbor] = n_best
                    self.cost_so_far[neighbor] = self.get_g_n(n_best) +
self.cost(n_best, neighbor)
                else:
                    if self.cost_so_far[n_best] + self.cost(n_best, neighbor) <
self.f_n(neighbor):
                        self.opened.insert(neighbor, self.cost_so_far[n_best] +
self.cost(n_best, neighbor))
                        self.came_from[neighbor] = n_best
                        self.cost_so_far[neighbor] = self.cost_so_far[n_best] +
self.cost(n_best, neighbor)

    # solves A*
    def solve(self):
        self.solve_private()
        self.generate_path()
```

```
        return self.path


    # generates the path via back propagation
    def generate_path(self):
        beginning = self.goal_index
        while not beginning == self.start_index:
            prior = self.came_from[beginning]
            thing = self.maze.state_from_index(prior)
            self.path.append(thing)
            beginning = prior


# runs the sample code from test_maze for maze1
# solver_num is which solver to run, plot is do I actually plot it
def testMaze2D(m, solver_num, title, plot=True):
    print('Goal: {}'.format(m.goal_state))
    print('Goal index: {}'.format(m.goal_index))
    print('Test state_from_index: {}'.format(m.state_from_index(m.get_goal())))
    print('Test index_from_state: {}'.format(m.index_from_state(m.goal_state)))
    print('Neighbors of start position:')
    print([m.state_from_index(pos) for pos in m.get_neighbors(m.start_index)])
    solver = Astar(m)
    if solver_num == 1:
        path = solver.solve()
    else:
        path = solver.solve2()
    for x in range(m.cols):
        for y in range(m.rows):
            state = (x, y)
            assert m.state_from_index(
                m.index_from_state(state)) == state, "Mapping incorrect for
state: {state}".format(state=state)
    if plot:
        m.plot_path(path, title)


# runs the sample code from test_maze for maze2
# solver_num is which solver to run, plot is do I actually plot it
def testMaze4D(m2, solver_num, title, plot=True):
    # m2.plot_path(path, 'Maze 2')
    solver = Astar(m2)
    if solver_num == 1:
        path = solver.solve()
    else:
        path = solver.solve2()
    if plot:
        m2.plot_path(path, title)
```

```
# completes step 2, part 1 with both mazes, solver_num is which solver I use
def step2_1(m, m2, solver_num):
   testMaze2D(m, solver_num, "Maze 1 A*")
   testMaze2D(m2, solver_num, "Maze 2 A*")


# completes step 2, part 2 with passed in maze, solver_num is which solver I
use
# completes step 3, part 2 as well
def step2_2(m, solver_num):
   for time_sec in [1, 0.25, 0.05]:
       print("TIME LIMIT {}".format(time_sec))
       start = time.time()
       epsilon = 10
       while not epsilon == 1:
           solver = Astar(m, epsilon)
           if solver_num == 1:
               solver.solve()
           else:
               solver.solve2()
           print("Epsilon {}".format(epsilon))
           print("nodes expanded {}".format(solver.nodes_expanded))
           print("path_length
{}".format(solver.cost_so_far[solver.goal_index]))
           end = time.time()
           if (end - start > time_sec):
               #print("Time exceeded: {}".format(end - start))
               break
           epsilon = epsilon - 0.5 * (epsilon - 1)
           if epsilon < 1.001:
               epsilon = 1
       if epsilon == 1:
           if solver_num == 1:
               solver = Astar(m)
               solver.solve()
           else:
               solver = Astar(m)
               solver.solve2()
           print("Epsilon {}".format(epsilon))
           print("nodes expanded {}".format(solver.nodes_expanded))
           print("path_length
{}".format(solver.cost_so_far[solver.goal_index]))


# completes step 2, part 1 with one maze, solver_num is which solver I use
def solve_4d(m1, m2, solver_num):
   testMaze4D(m1, solver_num, "4D-Maze 1")
   testMaze4D(m2, solver_num, "4D-Maze 2")
```

```
if __name__ == "__main__":
    m1 = Maze2D.from_pgm('maze1.pgm')
    m2 = Maze2D.from_pgm('maze2.pgm')
    #step2_1(m1,m2,1)
    #step2_2(m1,1)
    #step2_2(m2,1)
    m3 = Maze4D.from_pgm('maze1.pgm')
    m4 = Maze4D.from_pgm('maze2.pgm')
    #solve_4d(m3, m4, 1)
    step2_2(m4,1)
```

Code RRT

```
import numpy as np
import time
import random
from maze import Maze2D

class rrt():

    def __init__(self, maze):
        self.maze = maze
        self.start = maze.start_state
        self.goal = maze.goal_state
        self.start_index = maze.get_start()
        self.goal_index = maze.get_goal()
        self.end_point = (-1, -1)
        self.path = []
        self.parents = {}
        self.vertices = set()

    def solve(self, limit=100000):
        self.vertices.add(self.start)
        self.parents[self.start] = (0, 0)

        counter = 0
        while True:
            # pick a random point in the map
            if counter % 4 == 0:
                x = random.uniform(0, 24)
                y = random.uniform(0, 24)
            else:
                x = random.random() * 24
                y = random.random() * 24
            # check if point is in an obstacle
            if self.maze.check_occupancy((x, y)):
                counter += 1
```

```
                continue
            # find nearest neighbor
            nearest = self.find_nearest(x, y)
            # is there a collision if I draw a line
            if self.maze.check_hit((x, y), (nearest[0] - x, nearest[1] - y)):
                counter += 1
                continue
            # no collision so add to graph
            self.vertices.add((x, y))
            self.parents[(x, y)] = nearest
            # is nearest within 1 of the graph?
            if self.reached_goal(x, y):
                self.end_point = (x, y)
                print("solved {}".format(counter))
                break
            counter += 1

    def find_nearest(self, x, y):
        q_arr = np.array((x, y))
        arr = np.array(list(self.vertices))
        distances = np.linalg.norm(arr - q_arr, axis=1)
        min_index = np.argmin(distances)
        return (arr[min_index][0], arr[min_index][1])

    def generate_path(self):
        current = self.end_point
        sum = 0
        while not current == self.start:
            self.path.append(current)
            sum += np.sqrt(
                (current[0] - self.parents[current][0]) ** 2 + ((current[1] -
self.parents[current][1]) ** 2))
            current = self.parents[current]
        self.path.append(self.start)
        return self.path, sum

    def reached_goal(self, x, y):
        if (x - self.goal[0]) ** 2 + (y - self.goal[1]) ** 2 < 1:
            return True
        return False
if __name__ == "__main__":
    m1 = Maze2D.from_pgm('maze2.pgm')
    start = time.time()
    solver = rrt(m1)
    solver.solve()
    path, cost = solver.generate_path()
    end = time.time()
    m1.plot_path(path, 'Maze 2 Time to solve: ' + str(round((end-start),2)) +
's, Path length: '+str(round((cost),2)))
```