CS5112

Final Project: Considering Path Planning with Global Knowledge

Due on 12/19/2020

Noah Rush: njr86, Zhonghan Pei: zp59

February 2, 2021

1 Introduction

1.1 Background

Motion planning or path planning is a long-studied field focused on moving an object from some starting point to an ending point, while avoiding any potential obstacles. Planning can return optimal trajectories and speeds of many components of the robot or character, depending on the scenario and parameters of the situation. Path planning occurs when the robot, hereafter referred to as agent for generality, either has local or global knowledge of its environment. With global knowledge, the agent knows where all the obstacles are, whereas with local its knowledge is constrained some way, often through vision. There are many applications and potential applications of motion planning, including obvious ones in robotics, video games and robotic surgery.

1.2 Objective

In this paper we aimed to implement and analyze some common global knowledge path-planning algorithms, in order to better understand the situational usage of these algorithms. We attempted to implement with both static (non-moving) and dynamic obstacles. We did not consider Newtonian motion constrictions, so our results are more applicable to the computer simulation and video games fields, where you can control objects kinematically (without respect to force or mass). In our path finding considerations, there is no pre-defined graph defining nodes where the agent can move, instead it is a free space that we will refer to as the configuration space. We considered our configuration space as 2-dimensional, and each agent could make a decision about its next step. Our obstacles were polygonal, and didn't have to lie parallel to either x-y plane.

2 Methods

2.1 Static

In these methods, the algorithms are given the x,y position of the starting point (sp), target point (tp), and the vertices of all the obstacles (obs).

2.1.1 Greedy

The first algorithm we implemented was a brute force algorithm where the agent attempts to move towards the target at every step. The algorithm generates a series of rotations for the robot to take before each step forward. First the robot would calculate the rotation towards the end target.Subtracting the starting point from the target point gives a vector in the direction of the target. We can then get the target rotation by taking the inverse tangent of the vector.[1]

$$r_{optimal} = tan^{-1}(\frac{tp.x - sp.x}{tp.y - sp.y}) \tag{1}$$

To get back a normalized unit direction from this rotation, we simply take the sin and cosine to get the x and y components. This becomes our try direction.

$$dir_{try} = [sin(r_{optimal}), cos(r_{optimal})] \tag{2}$$

Adding this dir_{try} to the agent's current position, gives us our new test position. We test to see if at this position the agent would collide with any obstacles. (see section 2.3). If there is no collision, the agent moves forward, updating its position. If there is a collision, the agent adjusts its attempted rotation. First it calculates the direction between its current position and the center of the obstacle, again using equation 1, and then rotates in opposite direction from the center of the obstacle. In figure 1, the agent attempts to move toward the target, but senses a collision will occur. So it calculates the rotation angle to the center of the obstacle. Since it is positive,

[1]Since tan^{-1} returns results between $-\pi/2$ and $\pi/2$ We assume the target is in a positive y-direction from the starting point, or that the agent can re-orient its axis so that the starting point is the origin and the target is in the top 2 quadrants of the coordinate plane.

it rotates to the left (in the negative direction to avoid the object but stay oriented towards the target.

In many situations, this process will work for the robot getting to the target, but it has obvious pitfalls. The first is that it doesn't really take advantage of its global knowledge, and will often not find the optimal path. It could direct the agent directly towards the center of an obstacle wall, instead of seeing it coming and heading towards a corner wher it could more quickly avoid the obstacle. It could also go directly towards a particular large obstacle that will take a while to get around, instead thinking about it beforehand. Finally, it could fail to ever reach the target, by getting caught in a loop and having no knowledge of prior states.

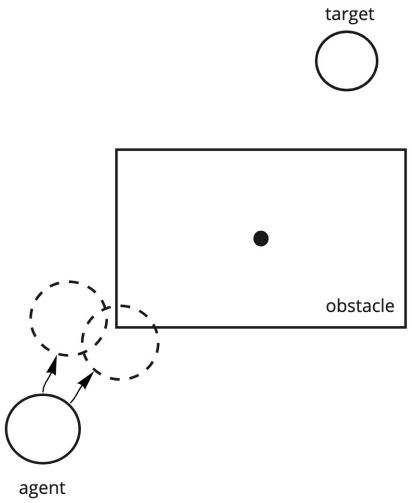


Figure 1: Brute Force Rotation Example

2.1.2 Dijkstra Corners

In this project we implemented Dijkstra with object avoidance, where we adjusted the old algorithm:

Denote the node we started with as N_s. Let D be the distance of node D be the distance from the initial node to N_s. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step. [4]

1. Create a vertex graph with all nodes on the boundaries of all objects, connect all these nodes and denote all edges with distance between nodes
2. Define a line intersection function denoted in 2.3.1, test line intersection between nodes and edges, remove edges that collide each other
3. Mark all nodes unvisited. Create a set of all the unvisited nodes called the unvisited set. Assign a tentaive distance to every node: set it to zero for our initial node and to infinity for all other nodes. Set the initial node as current.
4. Starting from the current node, calculated the tentative distance of all its neighbors and compare the newly acquired value with the assigned one and assign the smaller value.
5. Remove the current node from the unvisited set when all its neighbor nodes are calculated and the same node will not be considered unvisited again.
6. When considering the routing find between two nodes, and the destination node is marked as visited, or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.

7. Otherwise, select a node from the unvisited set marked with the smallest tentative distance, set it as the new "current node", and go back to step 3. [4]

Djikstra has time complexity $O(E + VlogV)$ and space complexity $O(V)$

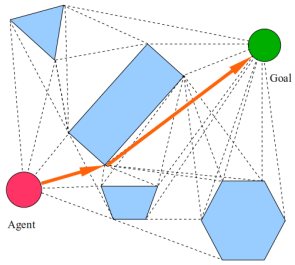


Figure 2: Dijkstra Application on Object Avoidance[9]

2.1.3 A*

A* is a graph traversal and path search algorithm, which aims to find the a path to the goal with smallest cost. It maintains a tree structure originating from the original node and extends by one edge each time until the termination criteria is satisfied. At each iteration A* considers its cost $f(n) = g(n) + h$, where $g(n)$ is the cost from the start node s to n and $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal. In the original paper by Peter Hart and Nils Nilsson, they proposed the algorithm as the following:

1. Mark s as open and calculate $f(s)$
2. Select the open node n whose value of f is smallest, resolve ties arbitrarily but always in favor of any node $n \in T$ T = the set of target nodes
3. If $n \in T$, mark n closed and terminate the algorithm
4. Otherwise, mark n closed and set s = n. Calculate f for each successor of s and mark as open each successor not already marked . [**A***]

In our algorithm, A* considers every integer square as a member of the graph, The graph is the grid of our configuration space. Our heuristic function $h(n)$ is the euclidean norm between n and the target point t. Since we are trying to find a shortest-path, this euclidean distance should be an optimal heuristic, and we will have both time and space complexity O(n), where n is the length of the solution path.

2.1.4 Rapidly-exploring random trees (Optimized)

In this implementation, we also applied Rapidly-exploring random tree (optimized), hereafter reffered to as RRT*. RRT* is an improvement over RRT.
In RRT, For a metric space X, and initial state x_{init} to a goal region $X_{goal} \subset X$, fixed obstacle region $X_{obs} \subset X$ must be avoided and the region RRT explores will be denoted as X_{free}. RRT starts a tree with its root node at the starting point, and has no edges. At each iteration, RRT picks a random point in X_{free} and attempts to connect it to its nearest neighbor in the tree. (Checks for collisions) This process continues until the graph contains a node in the goal region. [6] One issue with RRT is that it produces very cubic trees [3], with lots of inefficient turns. This is one issue that RRT* approves upon.

The optimized and updated version is usually known as RRT^*, RRT^* records the distance each vertex from its parent vertex, known as the cost. When the closest node is found, a neighborhood of vertices is examined in a fixed radius. If there exists a node with a cheaper cost than the proximal node found, the cheaper node replaces the proximal node. Secondly, RRT* adds the rewiring of the tree. After a vertex has been connected to the cheapest neighbor, the neighbors

are again examined. Then check the Neighbors if they are rewired to the newly added vertex and if it will decrease the cost. When the cost does indeed decrease, the neighbor then rewires to the newly added vertex, making the routing more smooth[3].

RRT and RRT* are randomized algorithms where there are no guarantees on time and space, it depends on number of iterations needed to converge. RRT* is intuitively more time-computationally expensive because it checks the neighbors twice to find the closest node and to rewire. the space complexity will be equal to the number of nodes added to the tree. Levalle and Kuffner [7] prove that given a possible path k, RRT will converge given infinite iterations. Additionally, they prove a successful bound on iterations, saying if k exist, k is no more than k/p, where p in the minimum ratio between an area k must travel through to get to its goal, and the total free space. In other words, iterations will increase when there is a narrow path that must be found to get to the goal.

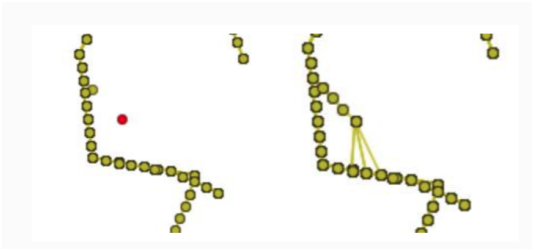


Figure 3: Once the red node is sampled, possible neighbors reevaluate for cheaper cost nodes to the root and rewire accordingly.[3]

2.1.5 Potential Field

The traditional algorithms used to find results have the following two drawbacks. First, the allowed shapes are too restricted to be applicable in general cases. Second, there might be a chance the algorithm will fail to find a solution.

In the cases of potential field, objects and obstacles will carry electric charges, and the resulting scalar potential field is used to represent the free space. Collisions Avoidance works between the obstacles and the robot through a repulsive force between them, which is the negative gradient of the field.[5]

The potential-field-based algorithm described above can be generalized as:

1. The free space is represented by a graph consisting of a finite number of nodes and edges, corresponding to points and edges along MPV.
2. Each node is assigned a cost depending on the width of free space at the node.
3. A candidate path is found that minimizes both the path length and the chance of collisions.
4. The local planner modifies the candidate path to derive a final collision-free path and orientations of the robot.
5. If a collision-free path cannot be found, remove the edge at which the unavoidable collision occurs, and go to 3.
6. Repeat 3 -5 until a solution is found or no candidate path exists.
7. If a solution is found, further optimize the solution with numerical algorithm. [5]

Time complexity for the potential algorithm is O(n*—obs—), where n is the grid/graph size, as it calculates the potential for every node on the grid, and within each node, each obstacle contributes its repulsive potential.

2.2 Dynamic

Apart from trying static obstacles, we also implemented one agent with dynamic obstacle simulations, and what we will focus on in this section, multi-agent pathfinding scenarios, where multiple agents have a start point and a goal, and must avoid one another to get to the goal.

2.2.1 Decentralized / Distributed

The idea behind the decentralized modeling is that the target/robot has no global knowledge of the intentions of the the other agents. The object maintains full-knowledge of static obstacles, and knows only the current position of the other agents, and each agent attempts to find its own solution.

We developed a simple reward based algorithm based on scoring each cell, and recalculating with every time-step.

1. Create a grid based environment with static obstacles and define a collision detection algorithm which ensures that all static obstacles don't collide
2. Create an action dictionary of 9 different actions from staying at the same place to going up, right, left, down, up-right and etc.
3. Create however many agent with a start node and a goal node.
4. At each step, create a rewards table with reference to new position and new position of the other agents. Score every cell around the agent's current location (defined by the action dictionary)
   - If next state is out of the grid, assign negative 10 to reward
   - If next state collide with static obstacle location, assign negative 10 to reward
   - If the next state is a dynamic obstacle or within 1 cell of a dynamic obstacle, assign negative 10 to reward.
   - Otherwise, assign a reward equal to decrease in euclidean distance between the two cells.
   - Assign a negative value greater than -10 to zero, so the agent won't stay in the same place.

   Unfortunately, this algorithm is prone to oscillations, where an agent can get trapped between obstacles and oscillate back and forth between two positions. This also can happen in the potential algorithm outlined above. Our solution to this was to check if an agent had visited to same location 3 times, then we could blacklist that node to prevent the agent from visiting it and getting caught in the same loop.

2.2.2 Centralized

In contrast to a decentralized system, a centralized algorithm knows the location of all agents and obstacles, as well as the intentions (goal nodes) of each agent. This allows the centralized planner to optimize over all the routes and attempt to find a lowest possible total path cost.

To test a centralized multi-agent pathfinding algorithm, we implemented the conflict-based search algorithm (CBS).[10] CBS aims to deal with the fact that expanding A* to multi-agent pathfinding is exponential in state space to the number of agents k. CBS instead aims to solve the multi-agent problem by decomposing it into a large number of single-agent problems, which are linear in the size of the graph.

To work, CBS grows a *constraint tree*, where each node contains a set of constraints, a set of solutions, and the total cost. Constraints are defined per agent, for example, agent 1 cannot be at (6,3) at time =7. Within each node, we run A* on each agent given their constraints, then we test for conflicts. When there is a conflict (2 or more agents at the same place at the same time), the nodes grow two children nodes with new constraints to each node. (In the case of more than 2 conflicts, we still grow two nodes. The conflict is still there, and will arise in deeper nodes). Once there are no conflicts in a node, the node is marked as a goal. If multiple goals are reached at the same level, they break ties based on lower cost. The cost is the total number of steps all agents take to reach their respective goals.

2.3 Collision Detection

This section describes the collision detection algorithms used in 2.1.1 (greedy), and 2.1.2 (Djikstra Corners). In 2.1.2, we use collision detection between two polygons, our player and the obstacle it is near. This can be broken down into two tests, checking if the edges are intersecting, and checking if the smaller object would completely go into the obstacle. For the Djikstra corners algorithm, we are also testing for line to line intersection.

2.3.1 Line Line Collision Detection

Say we have 2 line segments a and b, defined by points P_1 and P_2, and P_3 and P_4. (In two-dimensional space, so $P_1 = (x_1, y_1)$). Any point along line a, P_a can be defined as[2]

$$P_a = P1 + u_a(P_2 - P_1). \tag{3}$$

u_a can be thought of as how far towards P_2 P_a is from P_1. So when $u_a = 0$, $P_a = P_1$, and when $u_a = 1$, $P_a = P_2$ Similarly for line b, we have

$$P_b = P3 + u_b(P_4 - P_3). \tag{4}$$

If we set these two lines equal we get

$$P1 + u_a(P_2 - P_1) = P3 + u_b(P_4 - P_3) \tag{5}$$

which gives us two equations and 2 unknowns

$$x1 + u_a(x_2 - x_1) = x3 + u_b(x_4 - x_3) \tag{6}$$

$$y1 + u_a(y_2 - y_1) = y3 + u_b(y_4 - y_3) \tag{7}$$

Which we can solve(see appendix) and get

$$u_a = \frac{(x_4 - x_3)(y_1 - y_3) - (y_4 - y_3)(x_1 - x_3)}{(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)} \tag{8}$$

$$u_b = \frac{(x_2 - x_1)(y_1 - y_3) - (y_2 - y_1)(x_1 - x_3)}{(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)} \tag{9}$$

And based on what we said before, we know that line segments a and b are intersecting if both u_a and u_b are between 0 and 1, inclusive. If the denominator is zero, it means the lines are parallel, and with the exception of one special case (the lines are the same), this means there is no collision.

2.3.2 Polygon within Polygon Detection

In the brute force method, since we are testing the tentative new position, there is a possibility that the agent has gone inside the obstacle, and in that cases none of the edges are intersecting.(See figure 2) We only run this test if the edge test fails, since it is cheaper. For this test, we check if each vertex in the smaller polygon is inside the larger one. The intuition for deriving this equation is that a point is inside a polygon if when casting a ray outward from the point, it crosses an odd number of lines. See figure 3. [12]

Our algorithm takes advantage of this by setting a boolean called collision and setting it to false. Then each point in the inner polygon is tested against every line in the outer polygon, each time if the test is passed, we have a crossing and the collision boolean is flipped. When there are an odd number of crossings, the boolean collision will end up true. Our test involves shooting a ray from the point to the right. First we check if the point is within the y-range of the lines. v_c and v_n correspond to two adjacent vertices, and our test point is specified by (p_x, p_y) (Figure 4 shows the scenario where this returns true)

$$(v_c.y > p_y)! = (v_n.y > p_y) \tag{10}$$

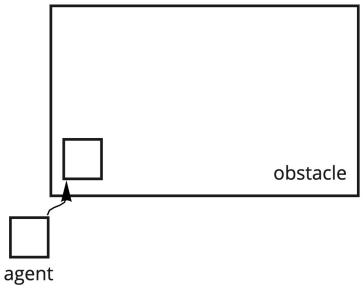


Figure 4: Agent enters obstacle

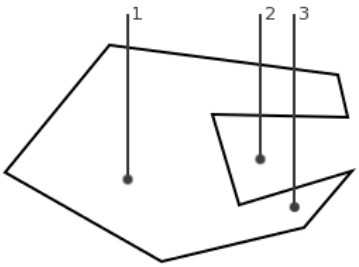


Figure 5: Points inside polygon cross an odd number of times.[12]

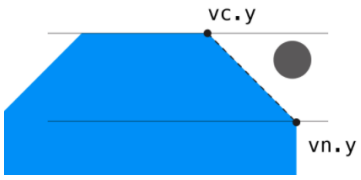


Figure 6: Test point is between $v_c.y$ and $v_n.y$ so we can test test a horizontal ray against the line segment these points define.[11]

The next equation is derived from the ray-casting describes above (see appendix for derivation).

$$p_x < (v_n.x - v_c.x) * (p_y - v_c.y)/(v_n.y - v_c.y) + v_c.x \tag{11}$$

If the both tests return true, there is a crossing. Again it odd number of crossings from the point means it is inside the polygon.

3 Experimentation

To analyze the static algorithms, we created random graphs of different sizes in the 2-d plane with a set number of obstacles in random places. We referenced Atsushi Sakai's excellent robotics repository for many of these algorithms.[8] For each scenario, we ran each algorithm 100 times with new random coordinates for the scenarios. The experiments were run in python.

For the dynamic algorithms we implemented a version of CBS[1], and our own greedy distributed algorithm. Similarly, we set up several tests with random static obstacles, and dynamic obstacles with random starting and end points within the configuration space.

Figure 7 shows examples for each static scenario, color-coded by algorithm. The black patches are the obstacles, and the agent starts at (0,0). Figure 8 shows still frames from a distributed dynamic simulation.

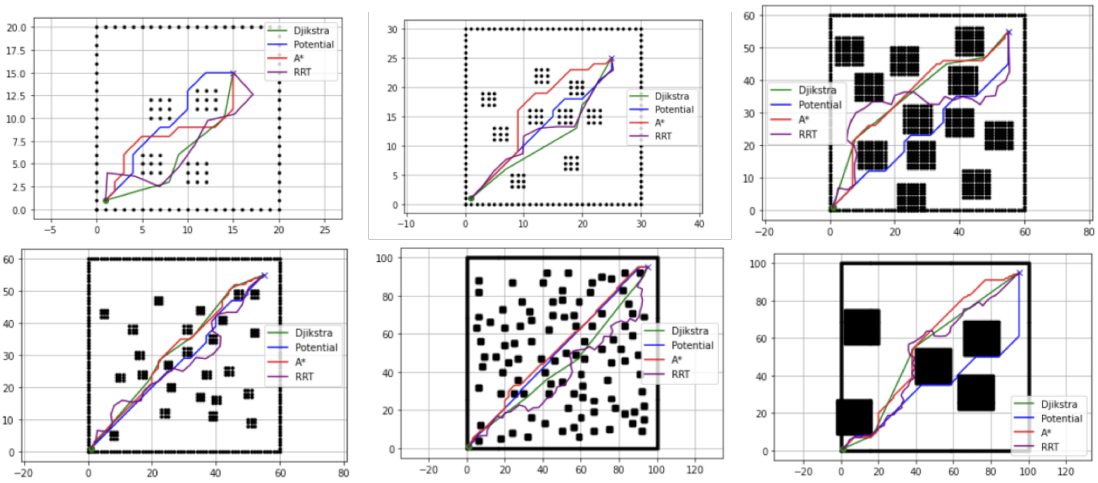


Figure 7: Results for Simulations

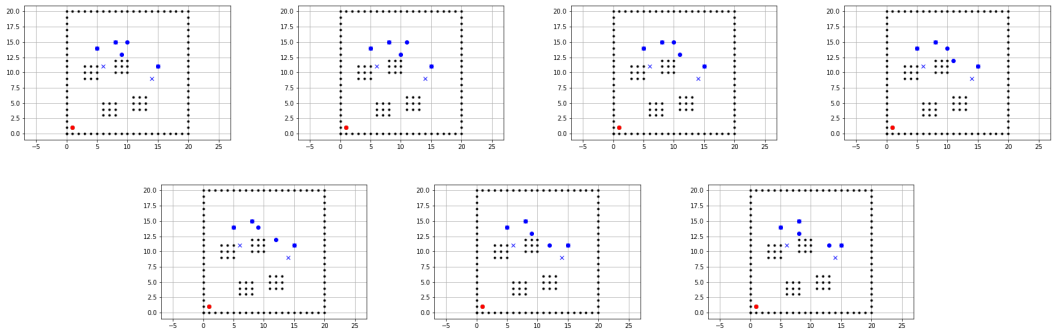


Figure 8: Sequential frames from the distributed multi-agent path finding, notice how the blue dot at the top backs up and to the right to let another agent through before proceeding to its target down and to the left.

4 Analysis

Figures 9 and 10 show experimental analysis of our static algorithms based on time and distance.

4.1 Static

4.1.1 Distance

The real distance is the straight line distance between our starting point and our target point. For all scenarios, the djikstra corners algorithm returned the closest distance to the shortest possible distance. The second-best was usually the Potential algorithm, though A* performed better on a size 100 grid, with 5 obstacles. Finally, bringing up the rear were RRT and RRT*, with RRT* outperforming RRT as expected.

4.1.2 Time

Time efficiency[2] is an important consideration with path planning, as these calculations can inform robots moving in real-time scenarios, or computer simulations dealing with dynamically updating obstacles.

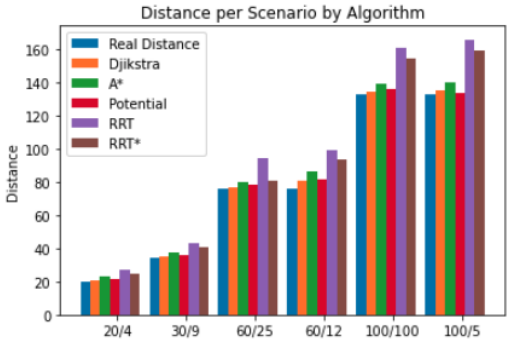
The fastest performing algorithms in our implementation were Djikstra, RRT, RRT* and A*, with some interesting caveats. A*, RRT and RRT* computation steps scale up with configuration

[2]We note that the algorithms we tested are not ones we wrote, and that python is not a low-level language, so these times could probably be sped up and are not definitive, but nonetheless, they do provide some interesting data points, and do seem to correlate with our analysis of the algorithms.

space. A* builds a graph based on the distance to the target, and RRT builds a graph based on random nodes that could be anywhere with the grid size. On the other hand, in our implementation of Djikstra corners, the graph scales up with the number of obstacles. This has less to do with the graph-searching algorithm, but more with how the graphs are designed, corners of obstacles vs. every grid space. As we can see there are advantages to setting up the graph either way.

It was also unclear in our initial scenarios which of RRT/RRT* or A* scales up more efficiently with just a grid size of 100, so we also ran tests on a grid size of 150, and see a more significant disparity between RRT/RRT* and A*, with RRT/RRT* taking less time. (See Table in figure 8.) However, another factor that negatively impacts RRT and RRT*'s efficiency is the size of the obstacles. As you can see from our results, RRT and RRT* underperformed A* in our 100/5 and 60/12 scenarios. With fewer, larger obstacles, the amount of possible paths to the goal actually decreases, and while the A* guides the path planning towards those paths, while both RRT's use random sampling, which means they could run up lots of iterations without finding useful points in narrow paths.[3] The RRT algorithms perform better when there are lots of possible paths, either when the obstacles are smaller or their just overall more free space, see how both RRT's actually outperform their performance in 100/100, in the 150/30 simulation. This is consistent with the findings of Levalle and Kuffner[7] expressed earlier, about the number of iterations depending on the ratio of constrained spaces to world size.

The Potential Field algorithm performs well in small grid sizes, but scales up with grid size, obstacle size (basically for each node an obstacle occupies, this impacts the potential field calculations), and amount of obstacles, so it has a similar jump as Djikstra Corners in the 100/100 simulation, yet does not fall back down like Djiksta does in the 100/5 simulation.



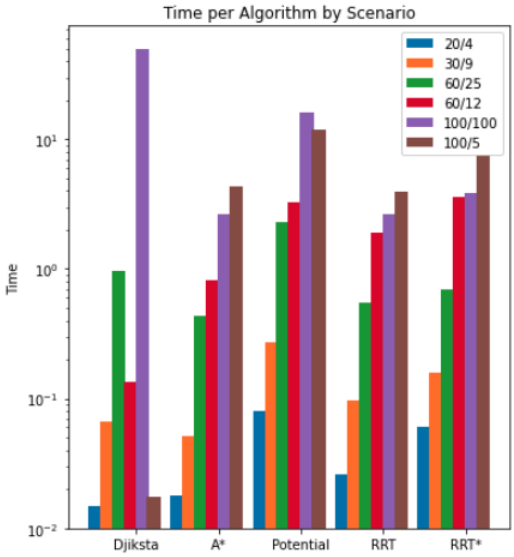
	20/4	30/9	60/25	60/12	100/100	100/5
Real	19.798	33.941	76.367	76.367	132.936	132.936
Djikstra Corners	20.965	35.0	77.0721	80.501	134.391	134.969
A*	23.176	37.435	79.641	86.228	139.249	140.404
Potential	21.427	35.71	78.1	81.914	136.523	134.0719
RRT	27.375	43.267	94.635	99.211	161.297	166.018
RRT*	24.792	40.467	81.061	93.5	154.55	159.0721

Figure 9: Average distance of path found from starting point to target point by algorithm. Real is the direct Euclidean distance between the 2 points.

4.2 Dynamic

We analyzed the path cost (how many total steps every agent took) in our multi-dynamic simulations in figure 10. As expected, the main benefit of the Centralized path planning was a reduction in cost. The Centralize path planning can utilize every grid space, while in a dynamic scenario the agents have to give each other space to maneuver. This has a trickle-down effect where agents in the distributed system shut down more paths, possibly even shortest paths, and force other agents to traverse around objects and look for 2nd or even 3rd best paths, in addition to the increased costs of just avoiding the other agents. Our centralized algorithm takes longer to run, but it produces more optimal paths, and can be run before execution time in lots of common real-world scenarios.

[3] We thought about recording iterations of RRT in these experiments, but they are pretty much exactly correlated with time, so higher time, means more iterations.



	20/4	30/9	60/25	60/12	100/100	100/5	150/30
Djiksta Corners	0.015	0.067	0.957	0.136	50.189	0.0175	0.0
A*	0.018	0.051	0.432	0.812	2.668	4.336	3.827
Potential	0.0796	0.275	2.32	3.282	16.164	11.75	0.0
RRT	0.0261	0.097	0.551	1.91	2.629	3.968	2.08
RRT*	0.0601	0.16	0.7008	3.577	3.871	7.632	3.02

Figure 10: Average computational time (seconds) of each algorithm.

	5/20/4/3x3	6/30/9/3x3	8/50/10/5x5
Distributed	76.7	126.96	272.88
Centralized(CBS)	62.38	115.44	256.56

Figure 11: Costs for different multi-agent pathfinding paradigms. (Agents/Grid Size/Number of Obstacles/Obstacle Size)

5 Conclusion

In this project, we aimed to investigate path planning algorithms with global knowledge, and learn a little bit about their situational advantages and disadvantages. We investigated ways to define the traversal graph, either by defining the whole grid, or targeting the corners of obstacles, and found to simply test the limiting factor before deciding could have substantial performance benefits. We explored randomly generated algorithms, that perform very well with lots of open space, and can outperform graph searching-algorithms while only adding on a small amount of distance to the total trip. Finally we looked, at the Potential Field Algorithm, which has an interesting structure and can find almost optimal paths, but could not scale up effectively in our implementation. In addition, we considered and implemented multi-agent path planning algorithms from two categories of implementation, distributed and centralized. We were able to confirm the benefits of centralized planning, a lower cost of movement, but understand the time costs and the need to run the algorithms prior to runtime.

Obviously this is just scratching the surface of path planning, and in the future we would like to explore implementing more dynamic algorithms, nonholonomic situations and clever ways to improve upon these algorithms like compacting open space into a single node for the Potential Field algorithm, or using intelligent sampling for RRT.

6 Appendix

6.1 Deriving Line Line Equation

We have

$$x_1 + u_a(x_2 - x_1) = x_3 + u_b(x_4 - x_3) \tag{12}$$

and

$$y_1 + u_a(y_2 - y_1) = y3 + u_b(y_4 - y_3) \tag{13}$$

So

$$y_1 - y_3 u_a(y_2 - y_1) = u_b(y_4 - y_3) \tag{14}$$

$$x_1 - x_3 + u_a(x_2 - x_1) = u_b(x_4 - x_3) \tag{15}$$

$$u_b = \frac{y_1 - y_3 + u_a(y_2 - y_1)}{y_4 - y_3} \tag{16}$$

$$u_b = \frac{x_1 - x_3 + u_a(x_2 - x_1)}{x_4 - x_3} \tag{17}$$

Setting them equal we get

$$\frac{x_1 - x_3 + u_a(x_2 - x_1)}{x_4 - x_3} = \frac{y_1 - y_3 + u_a(y_2 - y_1)}{y_4 - y_3} \tag{18}$$

Multiply both sides by the the denominator of the other over itself.

$$\frac{(x_1 - x_3)(y_4 - y_3) + u_a(x_2 - x_1)(y_4 - y_3)}{(x_4 - x_3)(y_4 - y_3)} = \frac{(y_1 - y_3)(x_4 - x_3) + u_a(y_2 - y_1)(x_4 - x_3)}{(x_4 - x_3)(y_4 - y_3)} \tag{19}$$

Denominators cancel and we can solve for the numerator

$$(x_1 - x_3)(y_4 - y_3) + u_a(x_2 - x_1)(y_4 - y_3) = (y_1 - y_3)(x_4 - x_3) + u_a(y_2 - y_1)(x_4 - x_3) \tag{20}$$

$$u_a(x_2 - x_1)(y_4 - y_3) - u_a(y_2 - y_1)(x_4 - x_3) = (y_1 - y_3)(x_4 - x_3) - (x_1 - x_3)(y_4 - y_3) \tag{21}$$

$$u_a((x_2 - x_1)(y_4 - y_3) - (y_2 - y_1)(x_4 - x_3)) = (y_1 - y_3)(x_4 - x_3) - (x_1 - x_3)(y_4 - y_3) \tag{22}$$

$$u_a = \frac{(y_1 - y_3)(x_4 - x_3) - (x_1 - x_3)(y_4 - y_3)}{(x_2 - x_1)(y_4 - y_3) - (y_2 - y_1)(x_4 - x_3)} \tag{23}$$

Which rearranged terms not withstanding, equals equation 8, we can see how we could do a similar process to solve for u_b

6.2 Deriving Polygon Point Equation

We have a test point p, (p_x, p_y) and we are testing against a line defined by vertices $v_n, (v_n.x, v_n.y)$ and $v_c(v_c.x, v_c.y)$. We define the line defined by v_c and v_n as

$$y = \frac{v_n.y - v_c.y}{v_n.x - v_c.x}x + m \tag{24}$$

We plug in v_c to solve for m.

$$m = v_c.y - \frac{v_n.y - v_c.y}{v_n.x - v_c.x}v_c.x \tag{25}$$

Complete equation for our line

$$y = \frac{v_n.y - v_c.y}{v_n.x - v_c.x}x + v_c.y - \frac{v_n.y - v_c.y}{v_n.x - v_c.x}v_c.x \tag{26}$$

Since our test ray will be horizontal, we want this equation for x

$$\frac{v_n.y - v_c.y}{v_n.x - v_c.x}x = y - (v_c.y - \frac{v_n.y - v_c.y}{v_n.x - v_c.x}v_c.x) \tag{27}$$

$$x = (y - v_c.y + \frac{v_n.y - v_c.y}{v_n.x - v_c.x} v_c.x) \frac{v_n.x - v_c.x}{v_n.y - v_c.y} \tag{28}$$

$$x = ((y - v_c.y) * \frac{v_n.x - v_c.x}{v_n.y - v_c.y} + v_c.x) \tag{29}$$

This is the equation for the line segment between v_n and v_c, we already know if we will perform this test only if p is constrained between $v_c.y$ and $v_n.y$, so setting $y = p_y$ will give use where the horizontal raycast meets this line. Since we are raycasting to the right, we want $x > p_x$, so

$$px < ((yp_x - v_c.y) * \frac{v_n.x - v_c.x}{v_n.y - v_c.y} + v_c.x) \tag{30}$$

will return true when the raycast hits to the right.

References

[1] Ashwin Bose. *Multi-Agent path planning in python*. Dec. 2020. URL: https://github.com/atb033/multi_agent_path_planning.

[2] Paul Bourke. *Points, lines, and plane*. Dec. 2020. URL: http://paulbourke.net/geometry/pointlineplane/.

[3] Tim Chin. *Robotic Path Planning: RRT and RRT**. Dec. 2020. URL: https://medium.com/@theclassytim/robotic-path-planning-rrt-and-rrt-212319121378.

[4] E.W. Dijkstra. "A note on two problems in connexion with graphs". English. In: *Numerische Mathematik* 1 (1959), pp. 269–271. ISSN: 0029-599X. DOI: 10.1007/BF01386390.

[5] Y. K. Hwang and N. Ahuja. "A potential field approach to path planning". In: *IEEE Transactions on Robotics and Automation* 8.1 (1992), pp. 23–32. DOI: 10.1109/70.127236.

[6] Sertac Karaman and Emilio Frazzoli. "Sampling-based Algorithms for Optimal Motion Planning". In: (2011). arXiv: 1105.1186 [cs.RO].

[7] S. M. LaValle and J. J. Kuffner. "Randomized kinodynamic planning". In: *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*. Vol. 1. 1999, 473–479 vol.1. DOI: 10.1109/ROBOT.1999.770022.

[8] Atsushi Sakai. *PythonRobotics*. Dec. 2020. URL: https://github.com/AtsushiSakai/PythonRobotics.

[9] Muthyam Satwik. *Finding-Shortest-Path-Avoiding-Obstacles*. Dec. 2020. URL: https://github.com/satwik-m/Finding-Shortest-Path-Avoiding-Obstacles.

[10] Guni Sharon et al. "Conflict-based search for optimal multi-agent pathfinding". In: *Artificial Intelligence* 219 (2015), pp. 40–66. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2014.11.006. URL: http://www.sciencedirect.com/science/article/pii/S0004370214001386.

[11] Jeffrey Thompson. *Collision Detection*. Dec. 2020. URL: http://www.jeffreythompson.org/collision-detection.

[12] Sid Vind. *Point-in-polygon: Jordan Curve Theorem*. Dec. 2020. URL: https://sidvind.com/wiki/Point-in-polygon:_Jordan_Curve_Theorem#:~:text=The%20Jordan%20Curve%20Theorem%20states,but%20point%202%20isn't..