Path Finding Optimization

Final Report

Remy Nicolas Hidra(11934990010)
Claire Emmanuelle Laverne(11934990009)
Changyo lee(118020990052)

Contents

Abstract

This project deals with Path Finding Optimization of a mobile robot based on a grid map. The main body of the project introduces the *Dijkstra, A** and *Bi-Directional A** algorithms. These modifications are focused on computational time and path optimality. Individual modifications were evaluated in several scenarios, which varied in a static environment. And by doing experiments, Path Finding Optimization of multi agents were demonstrated by *A** in the project.

1. Introduction

Path finding is an important area of research, it is an important point between artificial intelligence and robotics. In order to complete the task in a mobile and secure way, mobile robot path finding is an important indicator of intelligence. According to certain evaluation criteria, path planning is a collision-free path that mobile robot will find and reach the goal of a state from the initial obstacle environment.

Path finding method is dependent on a state space. State space represents all the possible positions and orientations of a robot. There are several ways to describe the state space. Usually it is represented by a planning algorithm. The most used representation of state space is grid map. The grid map makes it easy to update new information about the environment and *Dijkstra Algorithm* is a representative path planning based on a grid map. [1]

- *Dijkstra Algorithm*

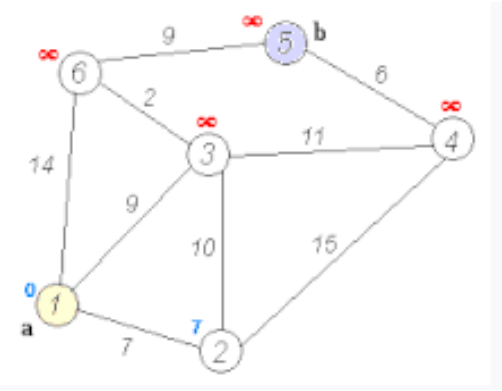

Figure 1. Dijkstra's Algorithm

Dijkstra Algorithm is a basic graph searching algorithm that find a single-source shortest path in the graph. The configuration space is approximated as a discrete cell-grid space, lattices, among others. [2]

For a given source node in the graph [Figure .1], the algorithm finds the shortest path between that node and every other. It can also be used to find the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.[3]

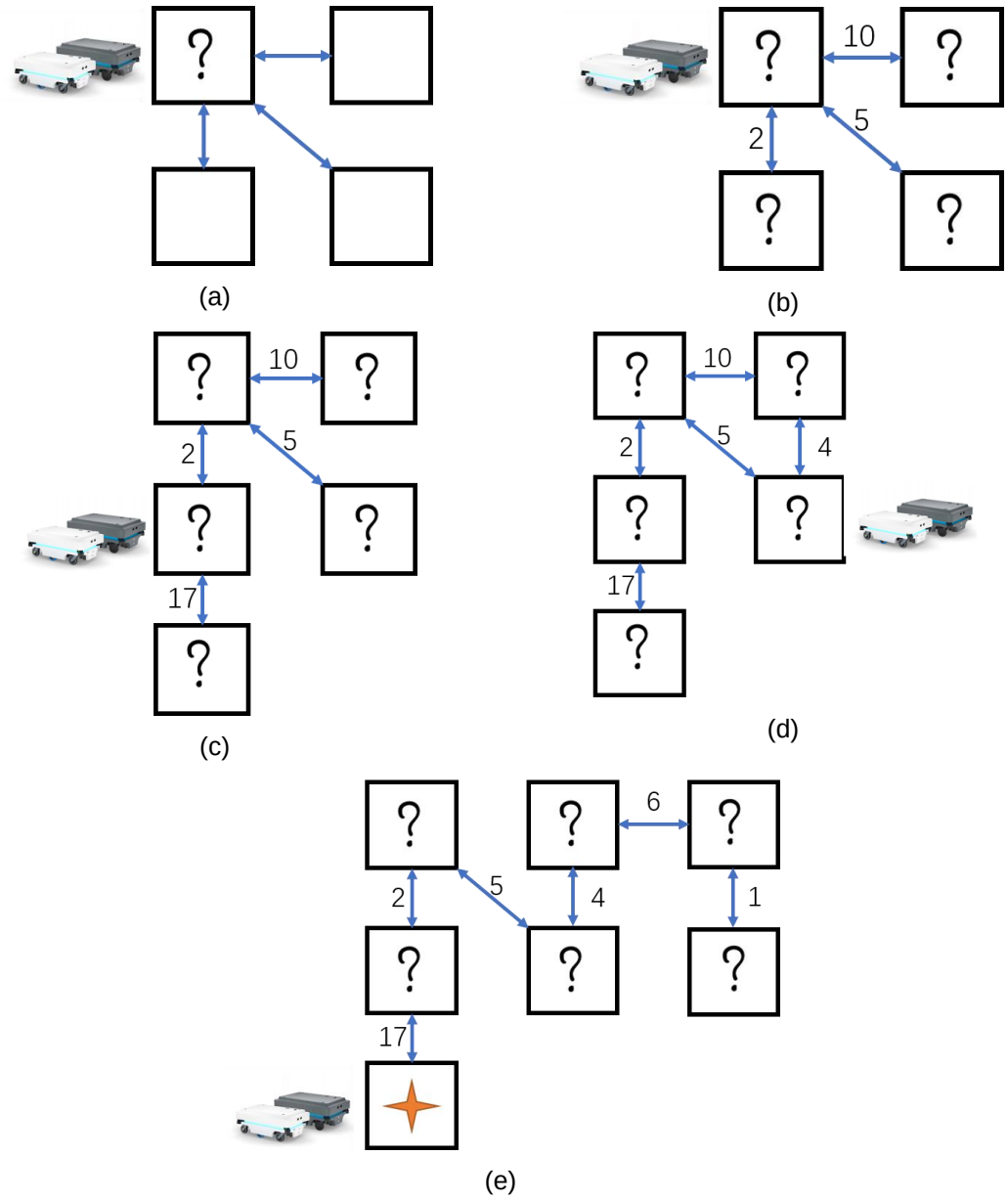

Figure 2. Dijkstra's Algorithm's work

For instance, we would like to find the shortest path between two intersections on a map [Figure2]: First, the robot explores the entrance room, in case that's where the target is [Figure .2(a)]. If it's not there, the robot determines all the rooms which are connected to the entrance room, and priorities them by distance, shortest first [Figure .2(b)]. Then robot explores the nearest room to the entrance for the target [Figure .2(c)]. If it's not there, robot determines all the rooms connected to that room, priorities them by the distance from the entrance room, and add it to the priority queue [Figure .2(d)]. The next nearest room to entrance room gets explored in turn. This time robot finds an alternative corridor to a room already on robot's priority list. The alternative route is shorter than the route already prioritized, then update the priority – meaning the robot will explore that room much sooner. If the alternative route was longer, though, the robot keeps the original priority [Figure 2(d)]. The robot keeps exploring until the target room becomes the highest priority room – via the shortest possible route

from the entrance – or until the robot runs out of rooms, determining that there is no target to be found.

```
function Dijkstra(Graph, source):
    for each vertex v in Graph:                 // Initialization
        dist[v] := infinity                     // initial distance from source to vertex v is set to infinite
        previous[v] := undefined                // Previous node in optimal path from source
    dist[source] := 0                           // Distance from source to source
    Q := the set of all nodes in Graph          // all nodes in the graph are unoptimized - thus are in Q
    While Q is not empty                        // main loop
        u := node in Q with smallest dist[ ]
        remove u from Q
        for each neighbor v of u:               // where v has not yet been removed from Q
            alt := dist[u] + dist_between(u, v)
            if alt < dist[v]                    // Relax (u,v)
                dist[v] := alt
                previous[v] := u
    return previous[ ]
```

The *Dijkstra Algorithm* proceeds as follows [11]:

1) While Q is not empty, pop the node v, that is not already in S, from Q with the smallest $dist(v)$. in the first run, source node s will be chosen because $dist(s)$ was initialized to 0. In the next run, the next node with the smallest $dist$ value is chosen
2) Add node v to S, to indicate that v has been visited
3) Update $dist$ values of adjacent nodes of the current node v as follows: for each new adjacent node u,

- If $dist\ (v) + weight(u,v) < dist(u)$, there is a new minimal distance found for u, therefore update $dist(u)$ to the new minimal distance value
- Otherwise, no updates are made to $dist(u)$
- Note : the weight of an edge(u,v) is taken from the value associated with (u,v) on the graph

The Algorithm has visited all nodes in the graph and found the smallest distance to each node. $dist$ now contains the shortest path tree from source s.

2. **Problem Statement**

The Dijkstra algorithm visits all the nodes, finding the minimum path cost from the start node to every other node, do wastes a lot of time while processing. Many of those nodes may not be on the path with the lowest cost.

3. **Optimizing Algorithm**

In this project, we introduce two algorithms: A* and Bi-Directional A* to better find an optimized path.

3.1 A*

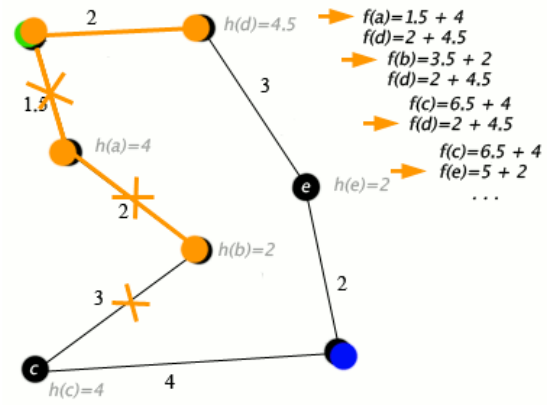

Figure 3. A* Algorithm

A Algorithm* is a graph searching algorithm that enables a fast node search due to the implementation of heuristics. It is an extension of Dijkstra's graph search algorithm [4].

It is formulated in terms of graph weights: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost. It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until termination criterion is satisfied [Figure 3]. At each iteration of its main loop, *A** needs to determine which of its paths to extend. It does base on the cost of path and an estimate of the cost required to extend the path all the way to the goal. *A** selects the path that minimizes

$$f(n) = g(n) + h(n)$$

$$h(n) = \sqrt{(x_n - x_{destination})^2 + (y_n - y_{destination})^2}$$

Where n is the next node on the path, $f(n)$ is the total estimated cost of path through node n, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal [Figure 4].

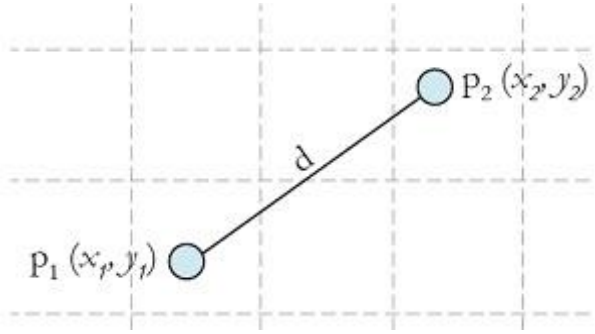

Figure 4. The Euclidean Distance Heuristic

A* terminates when the path it chooses to extend is a path from start to goal or if there are no paths eligible to be extended. The heuristic function is problem-specific.

```
Closed list = []
Q_I.Insert(x_I)and mark x_I as visited (open list)\
while Q_I not empty do
  sort(Q_I)
  if x = x_G or x ∈ Q_G
      return SUCCESS

  n ← open.pop()
  U(x) ← enpand(n)
  forall u ∈ U(x)
     x' ← f(x,u)
    if goal(x') = true then return makePath(x')
    if x' ⋂ closed = 0 then Q_I.Insert(x')
  closed ← n
return FAILURE
```

The *A* Algorithm* proceeds as follows [10]:

1) At any point during the search, there will be five kinds of states

- ***Unvisited***: states that have not been visited by yet. Initially, this is every state except x_I
- ***Dead***: stats that have been visited, and for which every possible next state has also been visited. A next state of x is a state x' for which there exists a $u \in U(x)$ such that $x' \leftarrow f(x,u)$. In sense, there states are dead because there is nothing more that they can contribute to the search; there are no new leads that could help in finding a path plan.
- ***Alive***: States that have been encountered, but possibly have unvisited next stats. There are considered alive.
- The **open list** or simply "open", is the list of all available un-expanded nodes. It is usually implemented as a priority queue.
- The **closed list** is an optional record of all expanded nodes. It is used to prevent repeated search and infinite loops.

2) The set of alive states is stored in a priority queue, Q

3) A while loop executed, which terminates only when Q is empty. This will only occur when the entire graph has been explored without finding any goal states, which result in a ***FAILURE***.

4) In each while iteration, the highest ranked element, x of Q is removed if x lies in x_G, then it reports ***SUCCESS*** and terminates; otherwise the algorithm tries applying every possible action $u \in U(x)$

3.2 Bi-Directional A*

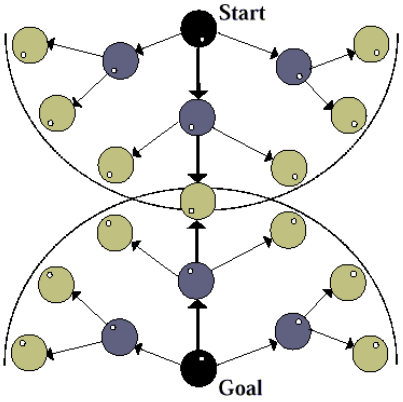

Figure 5. Bi-Directional A* Algorithm

A* Algorithm is searching from the start to the goal, Bi-Directional A* Algorithm can start two searches in parallel – one from the start to the goal, and one from the goal to the start. When they meet, it means that they found valid path and it stops to search a node.

The idea behind bidirectional searches is that searching results in a “tree” that fans out over the map [Figure 5]. The front-to-front variation links the two searches together. Instead of choosing the best forward-search node:

$$F\_forward = g(start, x) + h(x, goal)$$

or the best backward-search node:

$$F\_Backward = g(y, goal) + h(start, y)$$

this algorithm chooses a pair of nodes with the best $g(start, x) + h(x, y) + g(y, goal)$. [7]

```
Closed list = []
Q_I.Insert(x_I)and mark x_I as visited
Q_G.Insert(x_G)and mark x_G as visited
while Q_I not empty and Q_G not empty do
    if Q_I not empty // Forward Search
        sort(Q_I)
        if x = x_G or x ∈ Q_G
            return SUCCESS
        n ← open.pop()
        U(x) ← enpand(n)
        forall u ∈ U(x)
            x' ← f(x,u)
            if goal(x') = true then return makePath(x')
            if x' ⋂ closed = 0 then Q_I.Insert(x')
        closed_f ← n
    if Q_G not empty // BACKWARD_SEARCH
        sort(Q_G)
        if x' = x_I or x ∈ Q_I
            return SUCCESS
        m ← open.pop()
        U^-1(x') ← expand(m)
        forall u^-1 ∈ U^-1(x')
            x ← f^-1(x',u^-1)
            if start(x) = true then return makePath_b(x)
            if x ⋂ closed = 0 then Q_G.Insert(x)
        closed_b ← m
return FAILURE
```

The *Bi-Directional A* Algorithm* proceeds as follows [11]:

1) Forward search is used same as A* Algorithm
2) Backward search also is used by A*, but the process is from Goal(state) to Start(state)
3) The search terminates with success When two search tree (forward search, backward search) meet.
4) Failure occurs if either priority queue has been exhausted.

In this project, we will use the Dijkstra algorithm to implement path finding, and we will also implement A* and Bi-Directional A*, as optimizations of Dijkstra algorithm in a randomly generated map. Then, we will compare how optimized are those three algorithms and we will implement one on a multi-agent environment.

4. **Result**

To test our algorithms, we implemented them. We used Python 3, because we were all experienced with it and it is an easy language to do any kind of fast prototyping. We implemented the Dijkstra, A* and Bi-Directional A* algorithms. There are two versions of A*, one classic and one we called “optimized”. We developed the optimized version after testing the first one, which had some several performance issues. We believe those issues are due to a bad data structure management during runtime of the algorithm. Those optimizations are related only to the implementation, not the math of the algorithm.

We did a comparison between the Djikstra and the A* algorithm, and then between the A* and Bi-Directional A* algorithm. We do not believe it is relevant to compare Djikstra and Bi-Directional A*, since the latter is still very similar to A* on a surface level.

For each algorithm, we generate a set of 1000 random square grid. In a grid, we always set the starting point in one corner, and the goal point at the opposite corner. The generating algorithm guarantee that we will always find a path and a point on the grid is either free, an obstacle, the start or the goal. Lastly, all of our path finding algorithms consider the diagonal box as a valid move. We use this testing procedure on each grid size, from a 10-side grid to a 100-side grid, with increments of five boxes on each side.

Firstly, for the comparison between Djikstra and A*, we get the following results:

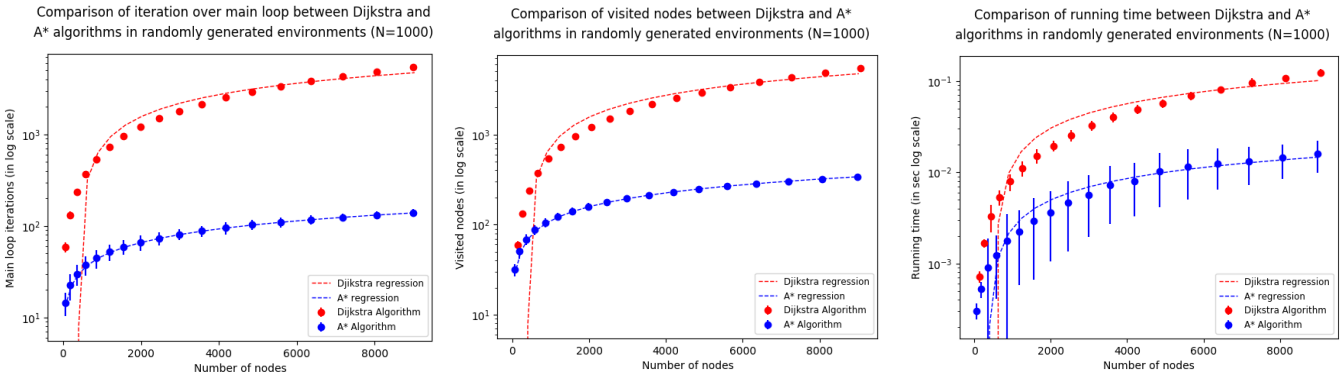

Figure 6. Results of our tests of the Djikstra and A* algorithms

The first graph features the number of main loop iterations, while the second graph represents the size of the graph built by the algorithms, against the size of their environment. The third graph is the running time of each algorithm. The first two graph are very similar, and we can deduce from them that A* is at least seven to height times more efficient than Djikstra, in term of memory consumed and time efficiency. To put things in perspective, we can see from the running time graph, that for the same task, the Djikstra algorithm may take 0.1 second, while A* may take 0.01 second. From theses results, we can safely deduce that A* is definitely more efficient than Djikstra, in a randomly generated environment.

Secondly, we did the same comparison between the A* and Bi-Directional A* algorithm. We show the following results:

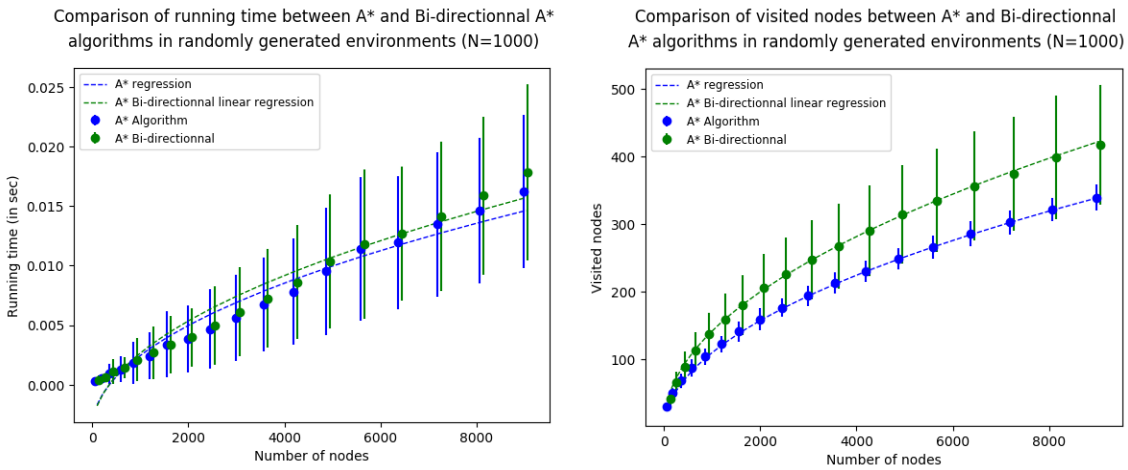

Figure 7. Results of our tests of the A* and Bi-Directional A* algorithms

We can see in the first graph the difference in running time between the two algorithms. Already, it is obvious that the Bi-Directional A* does not perform better than the A* algorithm. In some cases, it even performs worse, especially when the grid size increases. Though, the standard deviation of the results is very high so it is quite difficult to draw any real conclusion. However, by looking at the second graph, of the number of visited nodes in the grid, we can better interpret the algorithm. Indeed, for big grid sizes, especially bigger than a 20 or 30-side square grid, the Bi-Directional algorithm visits at least more nodes than the classic A*. To resume, on the contrary of our theory, our Bi-Directional A* algorithm performs worse than A*.

This result can be explained by looking at our experiment setup. Indeed, because the grid generating algorithm puts the starting and the goal points on opposite corner, while random obstacles appear between the two, there is a certain probability that the two path finding algorithms will go in different directions. In other words, for the Bi-Directional A*, the search from the start may go first in the top right direction, while the search from the goal may go first in the lower left direction. In this case the path might not meet until one has found its goal, which would mean a total inefficiency, because one search has been useless in order to solve the path finding problem. However, in a line of sight situation, which is, without any obstacle between the start and the goal, the Bi-Directional A* might perform better, for the reasons said previously. A maze situation would probably have analogous results.

5. **Application**

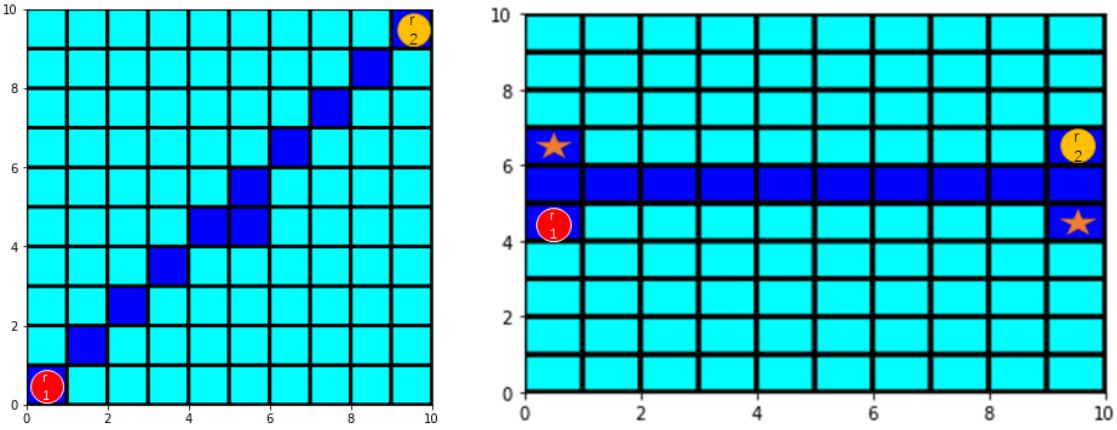

Figure 8. Multi Agent path Finding by A* (left:map1, right:map2)

Path Finding Algorithm are used on various robots (Domestic robots, UAVs, Industrial robots, Logistic robots) in different environments. In this project, we already experienced how to find an optimized path by using Dijkstra, A*, Bi-Directional A* algorithms and in our result, A* is a better choice for path planning and we wonder if A* is also available in a multi-agent path-finding system.

We create two different grid maps and use two robots [Figure 8]. We find out that using A* Algorithm is not working in multi-agent system at the same time. Therefore, we need to design an A* algorithm for Multi-robot system in different environments.

```
robot1 initalized path : [(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6), (7, 7), (8, 8), (9, 9)]
robot2 initalized path : [(9, 9), (8, 8), (7, 7), (6, 6), (5, 5), (4, 4), (3, 3), (2, 2), (1, 1), (0, 0)]
collision state at 5
before collision states of robot are robot2_state:(4, 4), robot1_state = (5, 5)
found possible move for robot2 state : [(4, 5), (6, 6)]
no collision found
final path-planning of robot1 : [(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (4, 4), (5, 5), (6, 6), (7, 7), (8, 8), (9, 9)]
final path-planning of robot2 : [(9, 9), (8, 8), (7, 7), (6, 6), (5, 5), (4, 5), (4, 4), (3, 3), (2, 2), (1, 1), (0, 0)]
```

Figure 9. Multi Agent path Finding Algorithm by A* (Map1)

At first, we use A* Algorithm to initialize the path of each robots and check a collision between robot1 and robot2 paths. If it found a collision, the algorithm checks available move state for robot2. If it has available move state at the collision state, it adds an available move state to the path list of robot2 after collision state, and recheck for collisions between them. If no collision was found, then they can move together at the same time.

```
[(4, 0), (5, 1), (5, 2), (5, 3), (5, 4), (5, 5), (5, 6), (5, 7), (5, 8), (4, 9)]
[(6, 9), (5, 8), (5, 7), (5, 6), (5, 5), (5, 4), (5, 3), (5, 2), (5, 1), (6, 0)]
collision state at : (5, 1)
collision state at : (5, 2)
collision state at : (5, 3)
collision state at : (5, 4)
collision state at : (5, 5)
collision state at : (5, 6)
collision state at : (5, 7)
collision state at : (5, 8)
robot1_path_possible
can not syncronize, robot 2 move first
robot2 path planning:[(6, 9), (5, 8), (5, 7), (5, 6), (5, 5), (5, 4), (5, 3), (5, 2), (5, 1), (6, 0)]
```

Figure 10. Multi Agent path Finding Algorithm by A* (Map2)

In the second case, we check for a collision on their path planning just as before, if it found a collision, the algorithm checks available move state for robot2. If no available move state was found for robot2, it means there is no way two robots can move at the same time. Therefore, the algorithm makes robot2 move first, and when robot2 reached its target position, robot1 starts to move to its target position.

From the multi-agent environment that we created, A* algorithm is hard to implement on path finding optimization. It is vulnerable on dynamic environment (multi-agent system) and as Map1 test, when collision and found available state, it recalculates A* again, it has high computational cost. Because of those problems, multi-robot path finding algorithms has been developed by many researchers. Representative algorithms are: Conflict-Based Search (CBS), Enhanced Conflict-Based Search (ECBS), Conflict-Based Search with Optimal Task Assignment (ECBS-TA), Prioritized planning using SIPP (Safe Interval Path Planning) [9]

6. **Reference**

1. David González, Joshué Pérez, Vicente Milanés, and F. N. (n.d.). A Review of Motion Planning Techniques for Automated Vehicles.
2. Wang, H., Yu, Y., & Yuan, Q. (2011). Application of Dijkstra algorithm in robot path-planning. *2011 2nd International Conference on Mechanic Automation and Control Engineering, MACE 2011 - Proceedings*, (2010011004), 1067–1069.
3. **https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm**
4. Duchon, F., Babinec, A., Kajan, M., Beno, P., Florek, M., Fico, T., & Jurišica, L. (2014). Path planning with modified A star algorithm for a mobile robot. *Procedia Engineering*, *96*, 59–69.
5. https://en.wikipedia.org/wiki/A*_search_algorithm
6. Pijls, W. (2008). A new bidirectional algorithm for shortest paths. *Intelligence*
7. Chen, J., Holte, R. C., Zilles, S., & Sturtevant, N. R. (2017). Front-to-end bidirectional heuristic search with Near-Optimal node expansions. *IJCAI International Joint Conference on Artificial*
8. Post, H., & Pijls, W. (2009). Yet another bidirectional algorithm for shortest paths. *Econometric Institute Report EI 2009-10*, 1–9.
9. Roman Bartak, Philipp Obermeier, Torsten Schaub, Tran Cao Son, Roni Stern.Multi-Agent Pathfinding: Models, Solvers, and Systems
10. **http://planning.cs.uiuc.edu/node40.html#fig:gfs**
11. **http://planning.cs.uiuc.edu/node50.html**