UNIVERSITY OF BIRMINGHAM

Multi-agent path finding with formation

Bsc Computer science

Student Name:	Guanzhi Chen
Supervisors:	Dr Masoumeh Mansouri
Student ID:	2149112
Date:	13th April, 2023
Academic Year:	2022/2023
Word Count:	4528
Total Pages:	18

Contents

Abstract

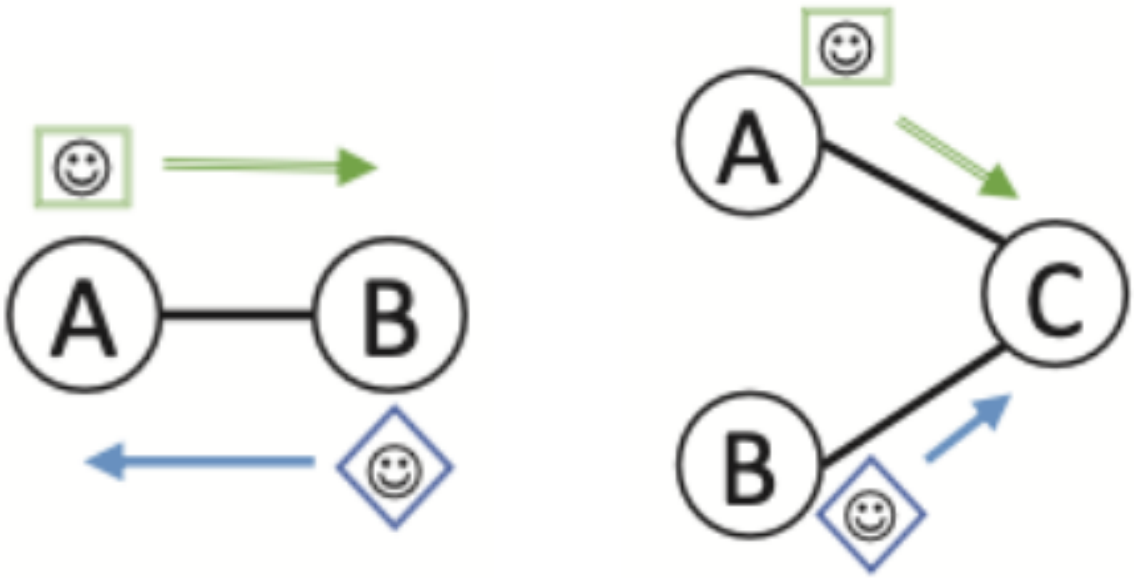
Multi-agent path-finding problems are of great importance to the development of robotics and AI. It aims to give a set of collision-free paths for all the agents in a given environment. MAPF-F (Multi-agent path finding with formation) introduce a problem within the domain of MAPF where agents coordinate in groups and keep a fixed formation during movement in order to hold an object that is much larger than the size of the agent itself. The result proves that the problem is solvable by adapting CBS(Conflict based search)[1], which uses a two-level solver to solve the MAPF problem, returning the optimal solution if one exists. By adding additional steps in both the high-level and low-level parts of CBS to achieve boundary detection, individual agent's pathfinding and conflict detections for groups of agents. The problem has great potential to be extended for future work.

CHAPTER 1

Introduction

1.1 | Introduction

Multi-agent path finding problem is crucial in 21 century. When given a map, a set of agents, sets of starting locations and goal locations for robots, they can move to another cell or stay where it was. When robots reach their goal location, they are treated as obstacles on the map. The algorithm needs to return a set of paths for each robot from its starting location to the goal location without colliding with each other. There are different kinds of conflicts, two major conflicts are swap conflict 1.1a and vertex conflict 1.1b. Swap conflict is when two robots try to swap positions with each other, vertex conflict is when two robots try to move into the same position. Path planning for robots is very tricky, especially in a warehouse environment where there are not much space left for robots to navigate and they change the map in real time thus every robot has to consider the location of all the other robots on the map when planning. The solution of the plan is evaluated by the makespan, which is the number of steps all the robots take to reach their goal location on the map.



(a) swap conflict **(b)** vertex conflict

Fig. 1.1. conflict example

1.2 | Application

MAPF problem has various applications in the industry including warehouse management[2], airport towing[3], UAV traffic management[4], automated valet parking[5], video games[6] and pipe routing [7], which aims at placing collision-free pipes from given start locations to given goal locations in a known 3D environment. In the video games industry, MAPF is also used to select routes for games character to move to different locations on the map without colliding with other players or NPCs.

1.3 | MAPFF

I introduce a new problem of MAPF called Multi-agent path finding with formation (MAPF-F), the problem includes a new type of robot group robots, consisting of four individual robots at the corner of a square shape group. The group keeps the same square formation during movement and allows other robots to pass with the robot group using the space except for the corner. This not only allows the automated warehouse system to move larger and irregular goods but also increases the efficiency of the whole system. As we can see in 1.2,

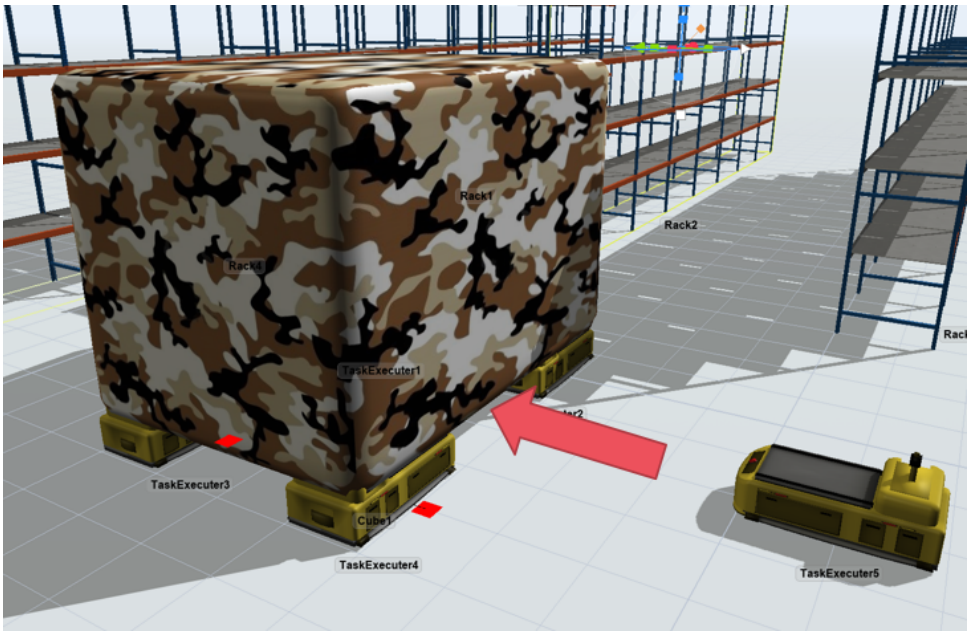


Fig. 1.2. application of MAPF-F

the application of MAPF-F is that the robot can carry goods that are much larger than conventional MAPF, this can be used in warehouses where the demand for moving gigantic objects like fridges, doors, bed, mattress, etc... is high. When group robots are moving large objects, they usually move very slowly to ensure safety, this allows other robots or robot groups to go under the space of the larger group thus improving the efficiency of the whole system. The problem restricted the formation of the group to be the same square during movement because it would be impossible to hold the object if the shape of the robot group is changing. To keep stability during any movement, the key is to make the center of mass of the group as lower as possible and to the center of the group. Generally, the square formation and diamond formation is considered to be the most stable one, square formation is the one shown in image 1.2 where the diamond formation is a 45-degree rotation of square formation. To make robots travel more smoothly when passing a robot group, I use the square formation.

CHAPTER 2

Literature Review

Generally, there are two ways of solving MAPF: coupled (centralized) approaches, which are formalized as a single, global search problem and decoupled, where paths are calculated separately for each agent and different techniques are used to resolve the conflicts between paths. Sharon et al.[8] shows that the behavior of optimal MAPF solver is very sensitive to the characteristics of the problem and there are no universal problems that work well on all the map.

2.1 | Decoupled solver

An example of a decoupled algorithm is Hierarchical Cooperative A* (HCA*) [9], where robots' routes are calculated separately, reserve previous robots' paths into a reservation table that records the time and location of each step. Future path finding for other robots must not collide with the robots already recorded in the table. Other variants of HCA* also exist. HCA* has lots of limitations for example when the number of agents increases, deadlock happens more often causing bad performance. Thus Windowed Hierarchical Cooperative A* (WHCA*)[9] limited the space-time search depth to a dynamic window and spread the computation throughout the route. Decoupled approaches usually run very fast but do not guarantee optimally and completeness.

2.2 | Coupled solver

Increasing Cost Tree Search (ICTS)[8] is a coupled approach to solve MAPF, it utilizes an increasing cost tree, and each node in the tree is an array of

$$[C_0, ...C_K]$$

represents the cost of all the agents' solution, the root of the tree is an array consisting of optimal solution cost for each agent while ignoring all the other agents. Each child of increasing cost tree is produced by increasing one of the agent's costs by one. A node is a goal node if there are no conflicts between all the agents' paths.

2.3 | CBS

Conflict based search (CBS)[1] is a decoupled and coupled approach to solving MAPF. CBS is a two-level algorithm, at the low level, the decoupled part, an A* search is performed only for a single agent at a time while ignoring all the other agents' movements. The search will use an admissible heuristic that never overestimates the path's cost to return an optimal path

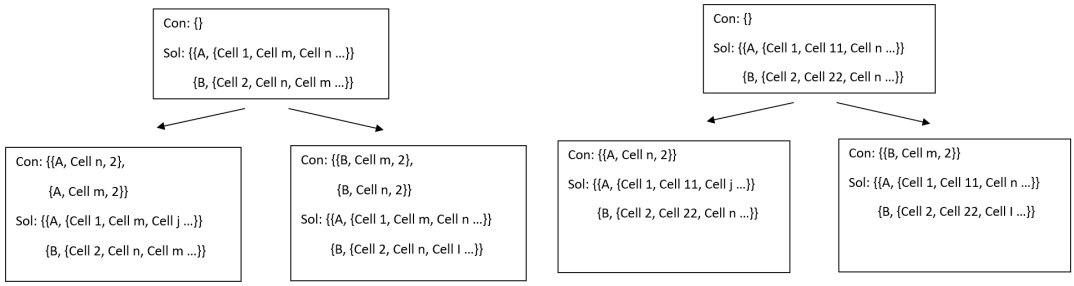
while considering all the constraints to ensure an optimal result. The map can be viewed as a grid map, each grid is 4-direction connected, and some grids will be flagged as obstacles that robots can not pass through. Robots can choose to go in any one of the four directions or stay at their original position for the next move. Once each robot has finished its low level search and found a valid path, the high level search will step in.

Algorithm 1: high-level of CBS

Input: MAPF instance
1 $R.constraints = \emptyset$
2 $R.solution$ = find individual paths using the low-level()
3 $R.cost = SIC(R.solution)$
4 insert R to OPEN
5 **while** OPEN *not empty* **do**
6 P ← best node from OPEN // *lowest solution cost*
7 Validate the paths in P until a conflict occurs.
8 **if** *P has no conflict* **then**
9 **return** P.solution // *P is goal*
10 C ← first conflict (a_i, a_j, v, t) in P
11 **foreach** *agent* a_i *in C* **do**
12 A ← new node
13 A.constriants ← P.constriants + (a_i, s, t)
14 A.solution ← P.solution.
15 Update A.solution by invoking low-level(a_i)
16 $A.cost = SIC(A.solution)$
17 Insert A to OPEN

Fig. 2.1. high level cbs
[1]

At the high level, it's a coupled approach. A constraint tree contains nodes that consist of $constraints$, $solution$ and $cost$ for each node. $constraints$ stops one agent's movement in order to give ways to another agent thus solving a conflict between the two agents, it's an empty list at the start. Each time a new node will copy all its parent node's constraints and add a new one to solve a conflict. $solution$ is a set of valid paths for each agent in the map such that all the paths are consistent with the constraint in the node and the map. $cost$ for each node is calculated by adding all the individual costs for each path in the node together. A search is performed on all the paths in a node to solve conflicts between agents' routes. It compares all the path pair by pair by simulating the movement of all the robots with their path planned by low level search to detect swap conflicts 1.1a and vertex conflict 1.1b. If all robots reach their goal location without conflict with any other robot, then a solution is found. However, when conflict is found, the algorithm will verify the type of conflict and solve it accordingly.
For swap conflict 2.2a, if agent A and agent B are swapping positions in Cell m and Cell n during time k, then high level algorithm will add two nodes in the constraint tree, one to forbid agent A to move to where agent B was and also forbid agent A to stay at where it was for the next move, as in Con:A, Cell n, k+1,A, Cell m, k+1; another one to forbid agent B for the same movement. For vertex conflict 2.2b, only forbidding one of the agents to move to the collision node in the next move for each node in the constraint tree is enough. Once new



(a) solve swap conflict **(b)** solve vertex conflict demo

Fig. 2.2. solve conflicts

constraints are formulated, the algorithm will copy all the previous constraints in the parent node and combine them with the new constraints to put into a new node in the constraint tree. Use low level search to plan for a new path taking into consideration the new constraint we have just added. Put the solution in the new node and update the cost of the node. Because the conflicts are identified pair by pair so each time there will be two new nodes added one for each robot. Each node is a new candidate plan which avoids this specific conflict by forcing one of the agents to explore another path during the conflict. Then among all the nodes in the constraint tree, select the one with the least cost(the sum of costs for all the robots' paths) to continue the high level search until a valid set of paths is found or there is no node in the constraint tree which stands for no solution. CBS is proven to return an optimal solution if one exists.

2.4 | Other solvers

Meta-Agent CBS(MA-CBS)[10] couple groups of agents into meta group agents if the number of internal conflicts exceeds a certain amount to help with increase optimally. Jiaoyang studied Moving agents in Formation(MAiF)[6] problem to bridge the gap between planning collision-free paths and keeping agents in formation by introducing SWARM-MAPF, a combination of swarm-based algorithm and MAPF algorithm. It solves the task when a desire formation is given and plans a set of collision-free paths for all agents while maintaining a close adherence to the desired formation.

Except for Search based solver above, there are other solvers like Reduction Based, reduced MAPF to standard known problems like SAT stc... Yu and LaValle[11] use Integer Linear Programming by transforming the MAPF problems into equations and goals as an objective function. Surynek[12] reduces MAPF problem to SAT, it turns the map, the start and goal locations and the obstacles into Boolean variables and passes, and combines these to produce an SAT formula that answers whether there is a valid solution.

CHAPTER 3

Methodology

3.1 | Definition

The new problem I introduced is Multi-agent path finding with formation (MAPFF) adopted CBS, inheriting the definition of CBS, a constraint for an agent m is given by

$$(m, v, t)$$

where agent m is prohibited at cell v during time t. A valid $path$ for agent m is an array indicating the movement of agent m and satisfying all of its constraints. A valid $solution$ is a set of valid paths for every agent in the environment. A $conflict$ might in two forms, a vertex conflict

$$(m_i, m_j, v, t)$$

1.1b is when two agents i and j try to move into the same cell v at the time t; a swap conflict

$$(m_i, m_j, v_1, v_2, t)$$

1.1a is when two agents i and j try to swap position between two cells at time t.

3.2 | Why CBS does not work on MAPFF

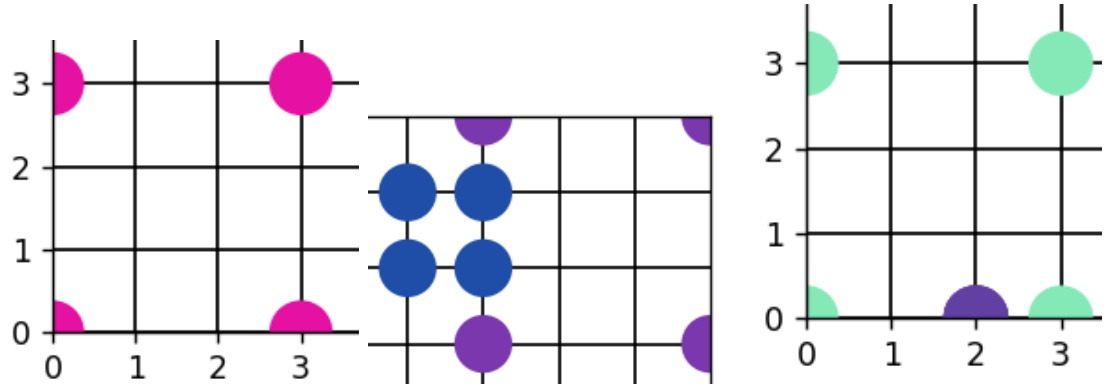
Because the conventional CBS is designed for pathfindings for single agents, they don't form a group or formation during movement. MAPFF needs an algorithm that can deal with the situation where groups of agents and single agents exist together and pass through each other. Both high level and low level of the CBS needs to be changed in order to deal with the new problem.

3.3 | MAPFF definition

We store the coordinate of the robot group only using the lead robots, the bottom left corner one 3.1a (0, 0), and we can use the length of the group n to get the coordinate of the other robots in the group (lead robot.x, lead robot.y + n), (lead robot.x + n, lead robot.y), (lead robot.x + n, lead robot.y + n), which are (0, 3), (3, 0), (3, 3). We use 0 for the length of the group if it's a single agent. A robot(or group robot) can pass by or overlap another group like demonstrate in 3.1b and 3.1c, if the smaller group does not collide with any of the four robots at the corner of the larger group.

3.4 | Method

During the high level CBS validation process where all the valid paths for each robot are compared pair by pair to check for conflicts, all four coordinates of a group robot need to be



(a) group robot with length 3 **(b)** group robot with length 3 **(c)** group robot with length 3

Fig. 3.1. different demo of MAPFF

checked against other agents' coordinates. When adding a new constraint for the group robot to solve the conflict, only the coordinate of the lead agent will be added. The low level also has to be modified in order to achieve the function of group agent pathfinding. It is worth mentioning that although it sounds unrealistic but some academia assume when solving MAPF problems, agent will disappear when it reach their goal location to reduce complexity of the problem. My assumption is agent will count as obstacles when they reach their goal location and will not continue to move to cooperative or give ways to other agents.

3.5 | Solve vertex conflict in MAPFF

In Algorithm1, we spilt the situation into 3 categories to detect conflicts for vertex conflict. The input are the paths for the two agents we are comparing $path1$ and $path2$ made up of arrays of their planned location from time 0; the length of the two robot groups *n1* and *n2*. The first situation is when two groups are all single agents, size 0. In this case, we only need to compare the coordinates for each agent at the same time stamp for the duration of the shortest path between two agents, see line 2 and line 3 Algorithm 1.
The second case is when both groups' sizes are larger than 0, we need to add all four coordinates for each group robot into a new array S to compare if there are any conflicts between the two groups, thus the new array S is of length 8. Line 8 - Line 15 algorithm 1 is adding all the coordinates for two robot groups using their lead robot's coordinates and the size of the group. Line 18 - Line 25 algorithm 1 checks any conflicts between the 8 coordinates one by one, remembering that all these 8 coordinates are for $agent1$ and $agent2$ at the same time stamp.
The third case is when one of the groups is size 0 and another is not, we need to add the four coordinates for the group which has a size larger than 0, and the one coordinate of that size 0 agent into the same array S to check for conflict, and we are comparing these 5 coordinates in S one by one.

3.6 | Solve swap conflicts for MAPFF

For swap conflicts, the steps are similar, but it is important to remember that we are considering the agents' movements between two continuous time stamps rather than one in vertex conflict, it involves the exchange of position between two agents thus when detecting conflicts, we

Algorithm 1 vertex conflict detection for MAPFF

Require: vector[Cell] $path1$,$path2$; INT $n1$,$n2$

▷ path1 and path2 are valid path for agent 1 and agent 2 , n1 and n2 are their group size

if $n1$ and $n2 == 0$ **then** ▷ two single agents
  for i in range (shortest length between $path1$ and $path2$) **do**
    if $path1[i] == path2[i]$ **then** found conflict
    end if
  end for
else if $n1 \neq 0$ and $n2 \neq 0$ **then**
  for i in range (shortest length between $path1$ and $path2$) **do** ▷ both group1 and group2 are **not** single agent

▷ add each corner coordinate of the group at time i for $path1$ and $path2$ into S

    $S \leftarrow newCell(path1[i] \rightarrow x, path1[i] \rightarrow y)$
    $S \leftarrow newCell(path1[i] \rightarrow x + n1, path1[i] \rightarrow y)$
    $S \leftarrow newCell(path1[i] \rightarrow x, path1[i] \rightarrow y + n1)$
    $S \leftarrow newCell(path1[i] \rightarrow x + n1, path1[i] \rightarrow y + n1)$
    $S \leftarrow newCell(path2[i] \rightarrow x, path2[i] \rightarrow y)$
    $S \leftarrow newCell(path2[i] \rightarrow x + n2, path2[i] \rightarrow y)$
    $S \leftarrow newCell(path2[i] \rightarrow x, path2[i] \rightarrow y + n2)$
    $S \leftarrow newCell(path2[i] \rightarrow x + n2, path2[i] \rightarrow y + n2)$
  end for
  $i = 0$

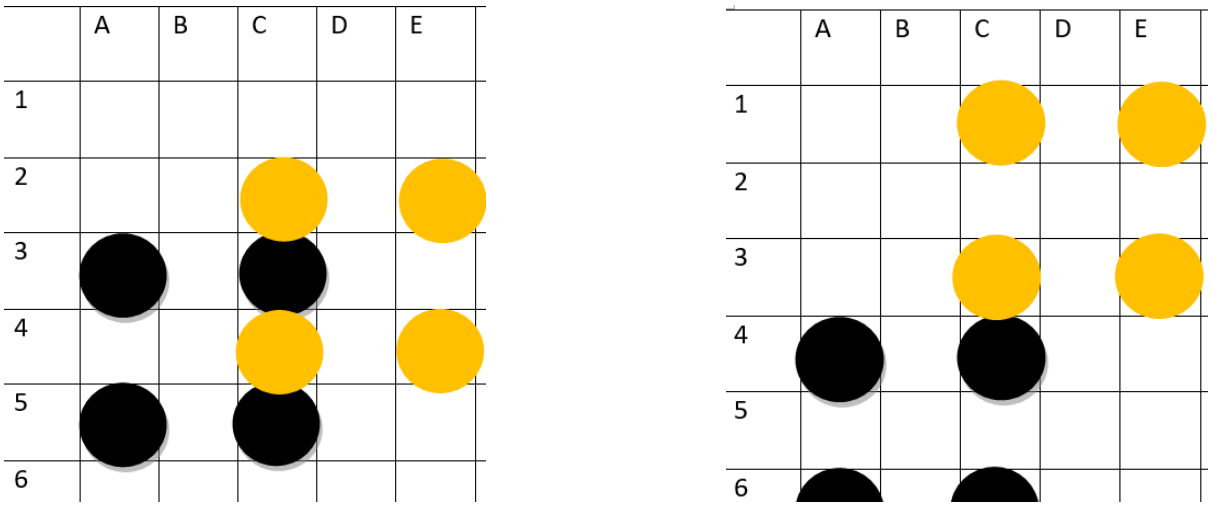
▷ compare each one of the coordinates in S for conflict

  for $i < 7$ **do**
    $j = i$
    for $j < 8$ **do**
      if $S[i] == S[j]$ **then** found conflict,
      else $j++$
      end if
    end for
  end for
else if $n1 == 0$ or $n2 == 0$ **then**
  if $n1 == 0$ **then**
    for i in range (shortest length between $path1$ and $path2$) **do**
      $S \leftarrow newCell(path1[i] \rightarrow x, path1[i] \rightarrow y)$
      $S \leftarrow newCell(path2[i] \rightarrow x, path2[i] \rightarrow y)$
      $S \leftarrow newCell(path2[i] \rightarrow x + n2, path2[i] \rightarrow y)$
      $S \leftarrow newCell(path2[i] \rightarrow x, path2[i] \rightarrow y + n2)$
      $S \leftarrow newCell(path2[i] \rightarrow x + n2, path2[i] \rightarrow y + n2)$
    end for
    $i = 0$
    for $i < 4$ **do**
      $j = i$
      for $j < 5$ **do**
        if $S[i] == S[j]$ **then** found conflict,
        else $j++$
        end if
      end for
    end for
  end if
  if $n2 == 0$ **then**
    ▷ similar as $n1 == 0$
  end if
end if

Algorithm 2 swap conflict detection for two single agents

```
1: for i in range (shortest length between path1 and path2) do
2:     if path1[i] == path2[i+1] and path1[i+1] == path2[i] and path1[i] ≠ path1[i+1]
   then
3:         found conflict
4:     end if
5: end for
```

will check if $agent1$'s position at time t is same with $agent2$'s position at time $t+1$ and if $agent1$'s position at time $t+1$ with $agent2$'s position at time t and if $agent1$'s position at time t is different than $agent1$'s position at time $t+1$, if both these three conditioned are satisfied then it's a swap conflict. This is represented in Algorithm 2. When dealing with swap conflicts for two group agents, algorithm 3 shows that we have two arrays S_1 and S_2 for storing coordinates for the two robot groups at time t and $t+1$. Then we compare each one of them from S_1 with S_2 like we did in algorithm 2. Line 2 - Line 18 of algorithm 3 is the process of adding these two into S_1 and S_2 using the lead robot's coordinates and their size $n1$ and $n2$, there are 8 coordinates in both S_1 and S_2 after this representing all the locations of the two robots at time t and $t+1$. Line 20 - Line 29 of algorithm 3 is comparing S_1 and S_2 for any conflict. m and n in the order of the sequence of how coordinates are put into S_1 and S_2, from Line 2 - Line 5, we put the coordinate of $agent1$ at time t in the sequence of left bottom corner (lead agent), right bottom corner, top left corner, top right corner, and follow the same sequence for $agent2$ at Line 6 - Line 9. So m of 3 means the top right corner of $agent1$ and m of 6 means the top left corner of $agent2$. To better illustrate this, see 3.2a and 3.2b, the two pictures are positioned for two robot groups at twp continuous time stamps, they are having a swap conflict. According to algorithm 3, we will add all the coordinates in 3.2a into S_1 and all the coordinates in 3.2b into S_2. S_1:C4, E4, C2, E2, A5, C5, A3, C3, S_2:C3, E3, C1, E1, A6, C6, A4, C4. When comparing S_1 and S_2 one by one, we will find that $S_1[0]$ equals $S_2[7]$ equals C4 and $S_1[7]$ equals $S_2[0]$ equals C3 and C4 and C3 are different cell, this match the expression at Line 22 of algorithm 3, and C3 and C4 are exactly the coordinates of where the swap conflict occurs. The situation of dealing with a group robot and a single



(a) swap conflict demo 1 **(b)** swap conflict demo 2

Fig. 3.2. swap conflict demo

robot for swap conflict is not much different from how we deal with them in vertex conflict

except that we are adding four corner coordinates of the group robot and one coordinate for the single robot from two continuous timestamps into two arrays to compare them and the way to check for collision is similar to Line 22 of algorithm 3.

Algorithm 3 swap conflict detection for two non-single agents

1: **for** i in range (shortest length between $path1$ and $path2$) **do**
2: $S_1 \leftarrow newCell(path1[i] \rightarrow x, path1[i] \rightarrow y)$
3: $S_1 \leftarrow newCell(path1[i] \rightarrow x + n1, path1[i] \rightarrow y)$
4: $S_1 \leftarrow newCell(path1[i] \rightarrow x, path1[i] \rightarrow y + n1)$
5: $S_1 \leftarrow newCell(path1[i] \rightarrow x + n1, path1[i] \rightarrow y + n1)$
6: $S_1 \leftarrow newCell(path2[i] \rightarrow x, path2[i] \rightarrow y)$
7: $S_1 \leftarrow newCell(path2[i] \rightarrow x + n2, path2[i] \rightarrow y)$
8: $S_1 \leftarrow newCell(path2[i] \rightarrow x, path2[i] \rightarrow y + n2)$
9: $S_1 \leftarrow newCell(path2[i] \rightarrow x + n2, path2[i] \rightarrow y + n2)$
10: $j = i + 1$
11: $S_2 \leftarrow newCell(path1[j] \rightarrow x, path1[j] \rightarrow y)$
12: $S_2 \leftarrow newCell(path1[j] \rightarrow x + n1, path1[j] \rightarrow y)$
13: $S_2 \leftarrow newCell(path1[j] \rightarrow x, path1[j] \rightarrow y + n1)$
14: $S_2 \leftarrow newCell(path1[j] \rightarrow x + n1, path1[j] \rightarrow y + n1)$
15: $S_2 \leftarrow newCell(path2[j] \rightarrow x, path2[j] \rightarrow y)$
16: $S_2 \leftarrow newCell(path2[j] \rightarrow x + n2, path2[j] \rightarrow y)$
17: $S_2 \leftarrow newCell(path2[j] \rightarrow x, path2[j] \rightarrow y + n2)$
18: $S_2 \leftarrow newCell(path2[j] \rightarrow x + n2, path2[j] \rightarrow y + n2)$
19: $m, n = 0$
20: **for** $m < 8$ **do**
21: **for** $n < 8$ **do**
22: **if** $S_1[m] == S_2[n]$ and $S_1[n] == S_2[m]$ and $S_1[m]! = S_1[n]$ **then**
23: found conflict
24: **else**
25: $n++$
26: **end if**
27: **end for**
28: $m++$
29: $n = 0$
30: **end for**
31: **end for**

3.7 | Low-level

After a conflict is found, constraints containing only the lead agents' coordinates will be added to a new node into the corresponding constraint tree like the original CBS, updating the solution that is consistent with the newly added constraints and the best node will be selected from the constraint tree to continue exploring. In the low level search part, in order to plan a valid path for group robots, when exploring the next step in A* search, all of the four corner coordinates of the group agent need to be checked for potential conflict with static obstacles and constraints on the map.
In algorithm 4, function IsValid() will use the length of the group and the coordinate given

as the argument to check for conflicts, for example, given IsValid(5,7) and the length of the group is 3, the function will check Cell(5,7), Cell(8,7), Cell(5,10), Cell(8,10) for any invalid cell. From line 4 to line 18 of algorithm 4, the algorithm checks for the up, down, left, right, and itself of the current cell for future movement.

Algorithm 4 modified A* outline

```
1: OPEN add start
2: while len(OPEN) > 0 do
3:     S ← findbestnode(OPEN)
4:     if IsValid(Cell(S → x,S → y)) then
5:         Successor ← (Cell(S → x, S → y))
6:     end if
7:     if IsValid(Cell(S → x + 1,S → y)) then
8:         Successor ← (Cell(S → x + 1, S → y))
9:     end if
10:    if IsValid(Cell(S → x,S → y + 1)) then
11:        Successor ← (Cell(S → x, S → y + 1))
12:    end if
13:    if IsValid(Cell(S → x − 1,S → y)) then
14:        Successor ← (Cell(S → x − 1, S → y))
15:    end if
16:    if IsValid(Cell(S → x,S → y − 1)) then
17:        Successor ← (Cell(S → x, S → y − 1))
18:    end if
19:    for K ← Successor do
20:        if k == GOAL then                          ▷ found valid path
21:        end if
22:        UpdateCost(K)
23:        if K in CLOSE then
24:            continue
25:        end if
26:        if K not in OPEN then
27:            OPEN add K
28:        end if
29:        CLOSE add K
30:    end for
31: end while
```

3.8 | Treat agents as obstacles when they reach goal locations

To achieve the effect of treating agents as obstacles when they reach their goal position, a semiconstraint is introduced, the agent will become a permanent obstacle when it reaches its goal location. A semiconstraint

$$(m, c, t)$$

is agent m will become a permanent obstacle at cell c after time t. Algorithm 5 demonstrate the steps to achieve this, starting from the first agent in the map, when $agent_0$ reaches its goal position, it will add itself as a semiconstraint in the semiconstraint list and jump to line 1 to start again, in this time, the last element of semiconstraint list's agentID is not smaller than the current agents' agentID, so the for loop will continue and goes to $agent_1$. When a valid solution has been found for $agent_1$, the current agents' agentID is larger than $SemiConstraint.pop().agentID$ at line 6, so $agent_1$'s goal position will be added as a semiconstraint and jump to line 1, start again from $agent_0$. These steps are necessary to ensure some agents who will be executed later in the process are not going to block some previous agents' paths when they reach their goal location. One example of this will be if $agent_a$'s ID is before $agent_b$'s, and $agent_b$'s goal location is on $agent_a$'s path. When $agent_a$ finishes executing, it won't know anything about $agent_b$'s goal location, and causing a conflict. So each time after a new semiconstraint is added, the algorithm has to start again from the first agent to check if the newly added semicontraint affects its path.

Algorithm 5 agent treated as obstacles when they reach goal position

```
1: for each agent k in the map do
2:     solution ← lowlevelsearch(K, Map, SemiConstraint)
3:     if SemiConstraint isEmpty then
4:         SemiConstraint ← GOAL
5:         JUMP to line 1 and start from the first agent
6:     else if SemiConstraint.pop().agentID < K.agentID then
7:         SemiConstraint ← GOAL
8:         JUMP to line 1 and start from the first agent
9:     end if
10: end for
```

CHAPTER 4

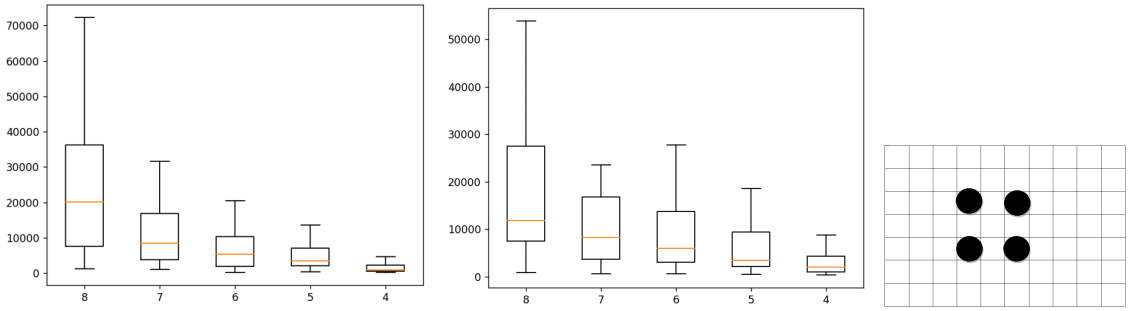
Results

4.1 | Results

Tests show that the algorithm can solve the MAPFF problem in an appropriate amount of time. Tests are running on different maps with different numbers of groups for each map, map1 is a 10 * 7 map with four obstacles in the middle as shown in 4.3c, and map2 is a 10 * 7 empty map. We set the testing algorithm to use each setup 50 times and plot the box chart of the time took to run these 50 times, extreme values are removed. Start location and goal location are randomly generated each time and will not overlap with each other or with their own start and goal location. When the number of groups is larger than 8, the result is very unstable and will have a very high value, which does not produce a useful result for testing. In some special situations, the algorithm will be stuck for a quite long time during the low level A* search to coop with the semi static obstacles, the agents which are treated as obstacles when they reach their goal locations. As we talked about in Algorithm 5, given n agents in a map, the low level search needs to be activated at least $2(n+1)$ times, and the node in the high level constraint tree will grow exponentially to ensure an optimal result. Theoretically, the algorithm should always return an optimal result it there is one existing, but in the real world, under some special situations, that process may take very long.
4.3a and 4.3b are box charts plotted, the x-axis is the number of agent groups, and the y-axis is the time taken to finish running in ms. 4.3a use the map 4.3c, 4.3b use an empty map without any obstacles. The top bar and lowest bar shows the maximum and minimum value, and the box in the middle shows the median, upper quarter, and lower quarter value. As we can see from the two plot, time cost increase as the number of robot groups increases which is what we were expecting to see. The time it takes to run with the empty map is significantly less than running on 4.3c. Because the MAPFF problem is very different from normal MAPF problems, it's very difficult to compare the result and performance with other MAPF solvers. Although we could count the group robots as four single robots and add all the group robots together to compare with another solver, the conclusion from the comparison between these are meaningless.

4.2 | Discussion

The high level algorithm2.1 mentioned before selecting the best node from the $OPEN$ list by using the lowest solution cost, which means the node with the lowest cost of all the paths is the best node. However, I did some experiments with this and found something different. I use 3 different maps 4.3 and the same number of groups of robots. I test the algorithm using four different node selection methods, min cost of all the agent paths, max cost of all the agent paths, min conflict in the node between all the routes, max conflict in the node between all the routes, the time is taken an average of 50 times running. The result shows that using



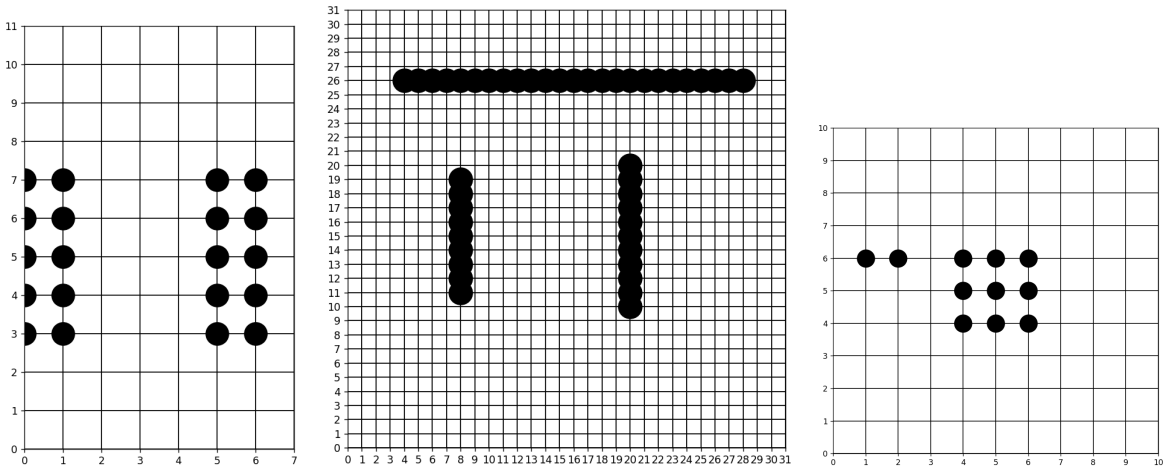
(a) result for map1 **(b)** result for map2 **(c)** map1

Fig. 4.1. result

max cost and min conflicts produces better performance than the other two. But this may not give an optimal result, min cost of the paths ensure an optimal solution but takes longer. The current GUI is implemented using Python matplotlib library, I couldn't find any suitable graphic interface library for C++, so I have to store the result of running into a file and start another Python program to read the file and demonstrate the movement of agents on the map.

	Min cost /ms	Max cost/ms	Min conflict/ms	Max conflict/ms
4	37155	16309	15280	N/A
1	200	95	97	247
3	508	491	193	497

Fig. 4.2. comparing different node selection method



(a) map1 **(b)** map3 **(c)** map4

Fig. 4.3. map for node selection method

CHAPTER 5

Conclusion

5.1 | Conclusion

The multi-agent path finding problem can be solved by adapting CBS, but there are still lots of limitations of the algorithm. The current version is a very preliminary one and is very expensive to run. Lots of work could still be done on the code to refine it and reduce unnecessary complexity. For the high level search part, currently once a new constraint is added in a new node and low level search is invoked to find a valid path consistent with all the constraints, all the agents in the map will be involved during low level search. However only the one agent that the constraint is related to needs to be involved for the low level search. If I have more time and this could be implemented, efficiency will be improved. Future work on this topic could be expanded on the 3D MAPF problem as well as shape-changing formation during movement. This could be applied to a different problem and gives more flexibility to the formation of groups thus increasing performance. During this period, I found a new IDE specifically for C++ programming, Microsoft visual studio. It is more powerful and convenient than the original one I was using, I used to spend hours trying to figure out setting up the .JSON config file and makefile to get the C++ code compiling and running. This will save me lots of time in the future by using Microsoft visual studio, it allows you to view the dynamic content for stack and vector during debugging while visual studio code can only view the start pointer and end pointer of a vector. There are lots of badly used pointers in the code, but due to time constraints, I am not able to fix all of them. Some of them may behave strangely causing unexpected errors due to bad memory management. I set up milestones at the beginning of the project so that I could check on my progress each week, I write the original CBS first, testing it's functionality and then modify it to the MAPFF problem.

CHAPTER 6

Appendices

The outline of the code is from $https://github.com/enginbaglayici/ConflictBasedSearch/$, where the CBS[1] code is. It gives me an idea of how to organize different structures of the program together like HighlevelSolver, LowLevelSolver and TreeNode. But the code inside these files are dramatically different, even for the original CBS, I made huge changes to the code on the conflict detection part and low level A* part.
The Data folder contains the map file, each map file consist of 1. first line size of the map, 11 11 means 11*11 map; 2. second line static obstacles coordinates; 3. starting location of each agent group; 4. the goal location of each agent group; 5. size of each agent group. Main.cpp is the main file, inside you will find the main function. The function read a map and translates it into a map class object then executes the algorithm. When finished, it will write the result into result.txt and write the time taken for each run into statistic.txt for future box chart plots if needed. plot.py is the plot file to plot the movement of agents from a result.txt. plotBox.py plot the statistic.txt file to get the box chart for analysis. ploystartgoal.py will plot the map, obstacles, staring locations, and goal locations using map.txt file to provide user a visual check if any of these are conflicting even before the program is running. Run makefile to compile all the necessary components into main.exe.

Bibliography

1. Sharon, G., Stern, R., Felner, A. & Sturtevant, N. R. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence* **219,** 40–66 (2015).
2. Ma, H., Hönig, W., Kumar, T. K., Ayanian, N. & Koenig, S. Lifelong path planning with kinematic constraints for multi-agent pickup and delivery. *Proceedings of the AAAI Conference on Artificial Intelligence* **33,** 7651–7658 (2019).
3. Li, J., Gong, M., Liang, Z., Liu, W., Tong, Z., Yi, L., Morris, R., Pasearanu, C. & Koenig, S. Departure scheduling and taxiway path planning under uncertainty. *AIAA Aviation 2019 Forum* (2019).
4. Ho, F., Geraldes, R., Goncalves, A., Rigault, B., Sportich, B., Kubo, D., Cavazza, M. & Prendinger, H. Decentralized multi-agent path finding for UAV traffic management. *IEEE Transactions on Intelligent Transportation Systems* **23,** 997–1008 (2022).
5. Okoso, A., Otaki, K. & Nishi, T. Multi-agent path finding with priority for Cooperative Automated Valet Parking. *2019 IEEE Intelligent Transportation Systems Conference (ITSC)* (2019).
6. Li, J., Sun, K., Ma, H., Felner, A., Kumar, T. K. & Koenig, S. Moving agents in formation in congested environments. *Proceedings of the International Symposium on Combinatorial Search* **11,** 131–132 (2021).
7. Belov, G., Du, W., Banda, M. G. D. L., Harabor, D., Koenig, S. & Wei, X. From multi-agent pathfinding to 3D pipe routing. *Proceedings of the International Symposium on Combinatorial Search.*
8. Sharon, G., Stern, R., Goldenberg, M. & Felner, A. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence* **195,** 470–495 (2013).
9. Silver, D. Cooperative pathfinding. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment.*
10. Sharon, G., Stern, R., Felner, A. & Sturtevant, N. Meta-agent conflict-based search for optimal multi-agent path finding. *Proceedings of the International Symposium on Combinatorial Search* **3,** 97–104 (2021).
11. Yu, J. & LaValle, S. M. Planning optimal paths for multiple robots on Graphs. *2013 IEEE International Conference on Robotics and Automation* (2013).
12. Surynek, P. Towards optimal cooperative path planning in hard setups through satisfiability solving. *Lecture Notes in Computer Science,* 564–576 (2012).