Prompt 2: Multi-robot path planning on a grid
A.
The basic idea of this project can be divided into several steps:

The first step is to find the original path from robots' start positions to the final positions. In this step, each robots' original path is created independently and all obstacles each of them needs to consider is only environmental obstalces.

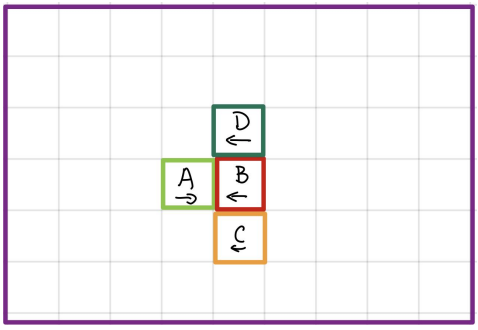
Since the start and goal configurations are sufficiently unpacked, this step should always be complete. If I use A star algorithm, assuming there are V vertices in the map, the time complexity should be O(VlogV).

The second step is to move each of the robots to its goal in sequence and do the coordination when there exists a square satisfying the "unpacked" condition.

The number of round robin is N(N-1), if I assume that there is conflict in every step, the time complexity of the system should be O(N*VlogV)+O(N(N-1)(VlogV)).

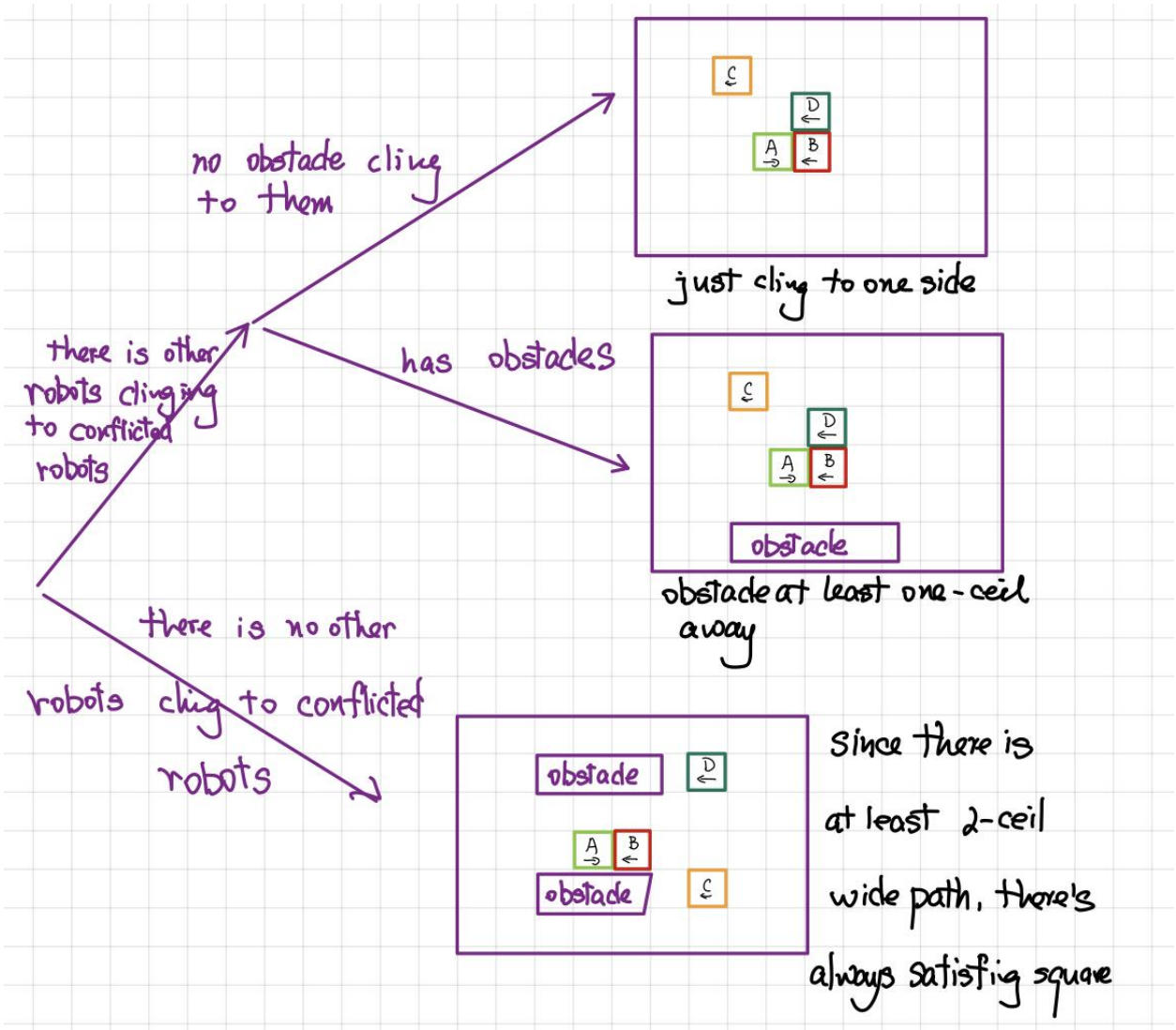
I think this square coordinate problem will not always allow robots to make progress, it will encounter deadlocks in many situations.

For example, like the situation shown below.



The robot A and B encounter conflict but can not find a coordinate square. In fact, there are many solutions to solve this problem. For example, if they wait until D and C move away, they can find the square. However, since the problem asks that once it can not find satisfied square, it return failure, the project will meet the failure in this situation.(I will solve this problem in my problem B's coding).

In this algorithm, only if robots have potential conflict when it is in the following situation will the algorithm succeed.



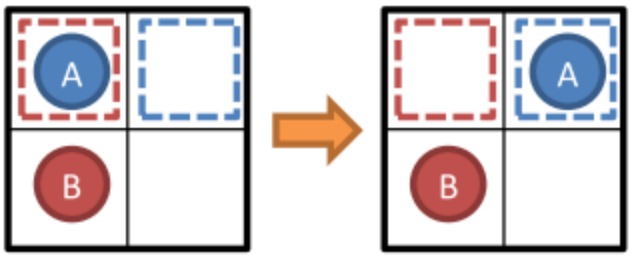
B.

For the algorithm mentioned above, I use the following strategy to deal with the conflict to make an improvement on the rate of success.

When there is a potential conflict, instead of finding the square to decide failure or not, I find the confliction pattern first. Let's assume that robot A and robot B have potential conflict.

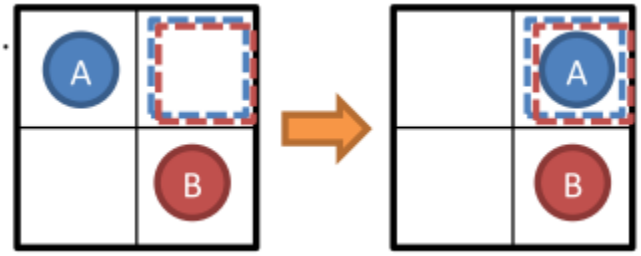
If robot A wants to move to another place, like shown below, I will let robot A wait no matter if there is a square or not.

1.



If both of them want to move to the same place, like what I showed below, I will let the robot with smaller order move into there.

1/2.



If they want to move into each other's position, I will let the one I control right now seek for the new path in the square and fix the other one. This can be done by using A star to find a new path from the robot's current location to the goal point with setting the other robot as an obstacle. If there is no square in this situation, I will let it wait since the only reason this can happen is that there are some other robots clinging to the conflicted robots. It will probably move away after some control loop.

If one of them has reached the final position, I will also let the other one bypass it by finding a new path by A star. If there is no square, I will let the robot wait.

The basic idea of this project can be divided into several steps:

The first step is to find the original path from robots' start positions to the final positions. In this step, each robots' original path is created independently by the Astar algorithm and all obstacles each of them needs to consider is only environmental obstalces.

The second step is to move each of the robots to its goal in sequence. My strategy to detect potential conflicts is that I record each robots' current position and when one robot wants to move to its next position, I scan all robot's current positions and find out if any of them is in that robot's next position.

If there is no potential collision, make the robot move to its next position. If not, check the collision type. I divided the collision type into three types:

Let's define the robot that wants to move as robot A and the robot on robot A's next step position as robot B. If robot B's next step position is not robot A's current position (type1), then let robot A stop and wait until robot B moves away.

If robot B's next step position is robot A's current position (type2), or if robot B is in its goal position and does not move anymore (type3), then we need robot A to move around robot B. Since my strategy is to fix robot B and move robot A, I generate a new path for robot A from its current location to final position, this time I define robot B's current location as an obstacle.

My pseudocode is shown below:
1.load in the map, get location of robot's start position, goal position and location of obstacles and map size information.
2.for i in all robots:
3. Find robot i's original path by Astar algorithm
4.while there is any robot not reach its goal position:
5. for i in all robots:
6. detect there is a potential conflict or not:
7. if no potential conflict:
8. let the robot moves
9. if there is a potential conflict:
10. check the conflict pattern:
11. if conflict pattern is 2 or 3:
12. add robot B's current location into obstacle and replanning robot A new path by Astar

Here are some core codes for the project:

For potential collision check:

```
# multi astar algorithm
def collision_check(occupied_matrix, newnode):
    for i in range(occupied_matrix.shape[0]):
        if newnode[0] == occupied_matrix[i][0] and newnode[1] == occupied_matrix[i][1]:
            return i
    return None
```

For checking collision pattern:

```
index = collision_check(occupied_matrix, original_path[i][1])
if index != None:
    # there is a collsion, check the collsion pattern then decide the move
    if occupied_matrix[index][0] == agent_goal[index][0] and occupied_matrix[index][1] == agent_goal[index][1]:
        collision_type = 1
    elif original_path[index][1][0] == original_path[i][0][0] and original_path[index][1][1] == original_path[i][0][1]:
        collision_type = 2
    else:
        collision_type = 3
```

For replanning the path for robot in type1 and type2 collision:

```
if collision_type == 1 or collision_type == 2:
    # re path planning
    start = (original_path[i][0][0], original_path[i][0][1])
    end = (agent_goal[i][0], agent_goal[i][1])
    obstacle_new = copy.deepcopy(obstacle)
    # print('original_path:',original_path)
    obstacle_new.append([original_path[index][0][0], original_path[index][0][1]])
    original_path[i] = astar(map_size, obstacle_new, start, end)
    original_path_no_start[i] = astar(map_size, obstacle, start, end)[1:]
```

This method can be successful for map 2 and map3, but will fail for map1.

The results for map2 and map3 are shown below: squares are robots and black blocks are the obstacles.

map2-video:https://drive.google.com/file/d/1HjvKP4nqx6CU7In2SIz8yI4HctpE96j9/view?usp=sharing

map3-videl:https://drive.google.com/file/d/1u45-IQH2sfn65mgCvEZtM3Vs0ckgJ_p2/view?usp=sharing

Waiting steps means the step where one robot detects there is a type1 collision so that it waits until the robot in its next position moves away.

	Steps with waiting steps	Steps without waiting steps
Map 2	58	42
Map 3	91	73

The reason why map1 would fail is shown below:

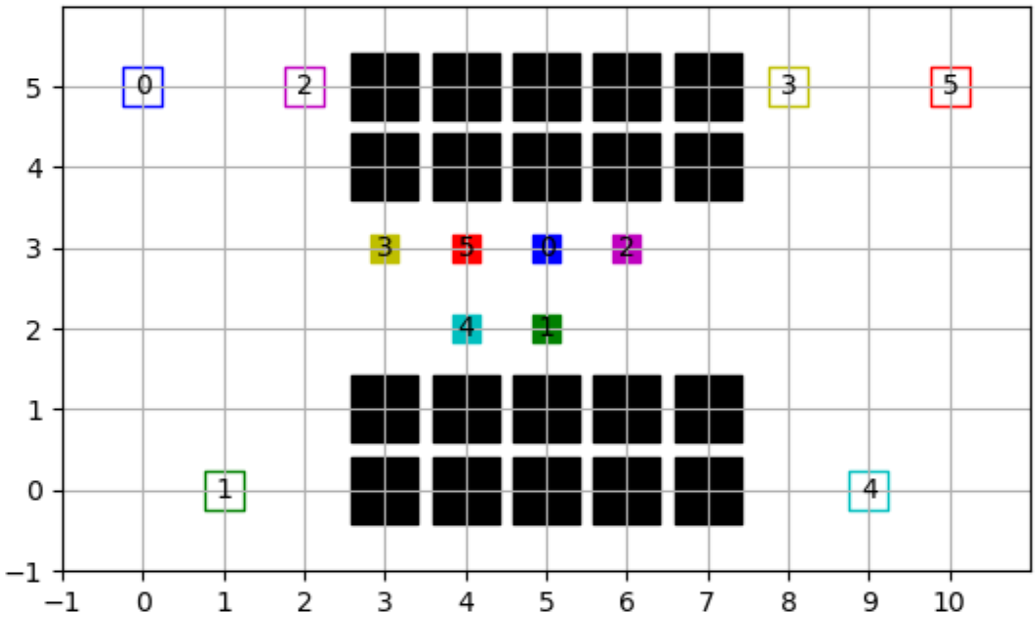


Fig 1. Map1's bug in sequence control

As you can see, when the map1 in this situation, robot3,4 and 5 want to move to the right while robot 0,1 and 2 want to move to its left. In this case, robot 0145 can not make its way toward their goals without the other reversing its progress.

In other words, in my project, take robot5 as an example, when it wants to move to its right, it meets robot 0 and the collision type is 1 or 2, so it waits or takes robot 0 as an obstacle and replanning the path. The new path will move through robot 4. When it wants to move following the new path, it meets robot 4, and it does the collision check again and does the same thing(wait or replanning the path). This will make it into a dead loop.

Since I need to solve this problem and apply advanced algorithms for robots to move simultaneously, I decided to answer question D here first.

Since the control logic asks us to let the robots move on their path one by one and only deal with the conflict when it occurs, which means that the robot can not predict the conflict many steps before and avoid it, it seems like when map1's bug occurs, it can not solve it. Also, if we want every robot to move back when they are in a deadlock, the control logic should be very complex since each robot can only move back when there is no robot in its back at the beginning of the movement. This problem can only occur in the 2-cell wide narrow path since only there will all robots stuck together. Robots can always find new paths in a wide field in my algorithm.

In that case, I want to avoid the type 2 and 3 collision in the 2-cell wide narrow path. The strategy is shown below.

The first step is to find the original path from robots' start positions to the final positions. In this step, each robots' original path is created independently by the Astar algorithm and all obstacles each of them needs to consider is only environmental obstalces.

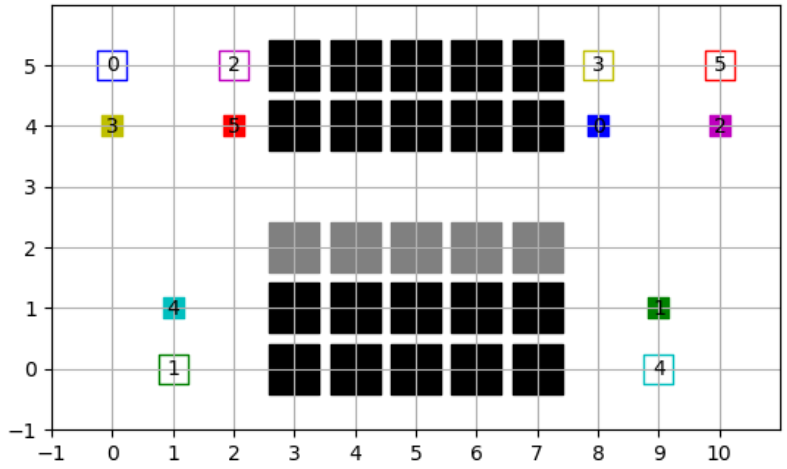
Secondly, I find the 2-ceil wide narrow path no matter its from left to right or from up to down, the code is shown below:

```
def narrow_pattern(map_size, obstacle):
    narrow_upper = []
    narrow_lower = []
    narrow_left = []
    narrow_right = []
    narrow_total = []
    for i in range(map_size[0]-3):
        for j in range(map_size[1]):
            if ([i, j] in obstacle) and ([i+1, j] not in obstacle) and ([i+3, j] in obstacle):
                narrow_upper.append([i+1, j])
                narrow_lower.append([i+2, j])
                narrow_total.append([i + 1, j])
                narrow_total.append([i + 2, j])
    for i in range(map_size[0]):
        for j in range(map_size[1]-3):
            if ([i, j] in obstacle) and ([i, j+1] not in obstacle) and ([i, j+3] in obstacle):
                narrow_left.append([i, j+1])
                narrow_right.append([i, j+2])
                narrow_total.append([i, j+1])
                narrow_total.append([i, j+2])
    if narrow_right != []:
        return narrow_left, narrow_right, narrow_total
    if narrow_upper != []:
        return narrow_upper, narrow_lower, narrow_total
    return None
```

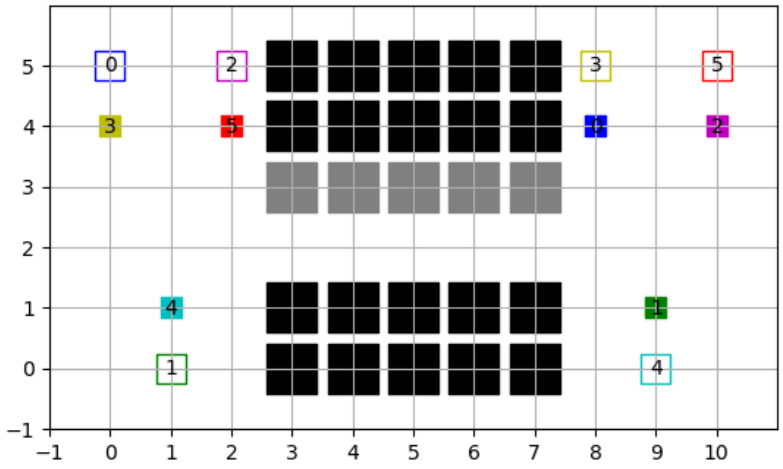
After that, I scan each robot's original path and find out if it needs to move through the narrow path or not. If it needs to move through it, I would check which side of the narrow path the robot entered from. Let's define two sides as side1 and side2.

If part of the robot's original path is in the narrow path, I will replan the path for it. The replanning logical is shown below:

If the robot enters the 2-cell wide narrow path from side 1, when I replan the path, I add the following obstacles for it, gray blocks are the new obstacles. In that case, robot moving from this side can only use the upper half of the 2-cell wide narrow path.



On the other hand, robots entering the 2-cell wide narrow path from side 2 can only use the lower half of the 2-cell wide narrow path.



In that case, robots entering from different sides can not meet in the narrow path, which means that there is no type2 collision in the narrow path.

The pseudocode is shown below:

1.load in the map, get location of robot's start position, goal position and location of obstacles and map size information.
2.for i in all robots:
3. Find robot i's original path by Astar algorithm
4. check if there is 2-cell wide narrow path or not
5. if there is a path:
6. for i in all robot:
7. if part of this robot's original path is in the narrow path:

8. check which side does the robot enter the path, add corresponding obstacles and replanning the path
9. while there is any robot not reach its goal position:
10. for i in all robots:
11. detect there is a potential conflict or not:
12. if no potential conflict:
13. let the robot moves
14. if there is a potential conflict:
15. check the conflict pattern:
16. if conflict pattern is 2 or 3:
17. add robot B's current location into obstacle and replanning robot A new path by Astar

This method can be successful for all maps

The results for all maps are shown below: squares are robots and black blocks are the obstacles.
map1_video:https://drive.google.com/file/d/1I1QiJtSOT6D9_-RdAIJQWBvHOQqzGNJK/view?usp=sharing

map2-video:https://drive.google.com/file/d/1vK13G8BGnjXSV10jbf92hBmhfw34ghlp/view?usp=sharing

map3-videl:https://drive.google.com/file/d/1dBgrUlLpOsnu7MGV5uYPvYZOeOwjmmwt/view?usp=sharing

	Steps with waiting steps	Steps without waiting steps
Map 1	75	73
Map 2	58	42
Map 3	76	73

It seems like the advanced algorithm improves both the success rate and efficiency, since the steps with waiting steps is 91 for map 3 with the original algorithm but 76 with advanced algorithm and the algorithm can work successfully on all three maps.

C.
I use the advanced algorithm on this multi control algorithm to let all robots move together.

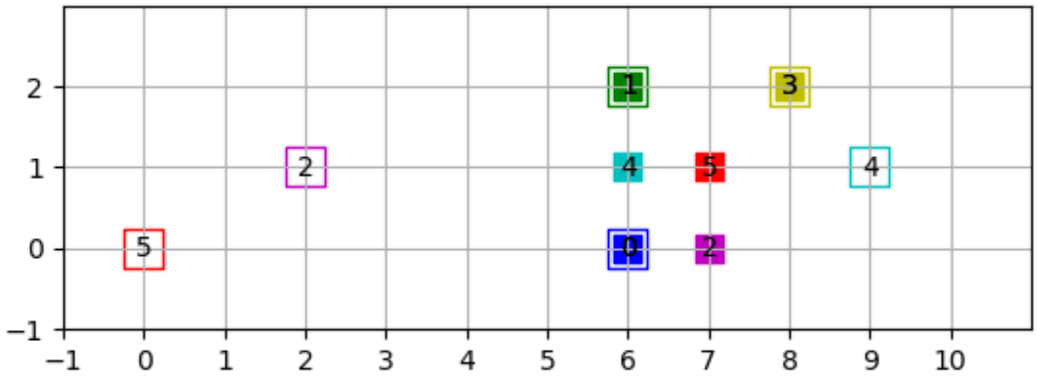
The algorithm can works successfully on map1 and map3, as shown below:

map1-video:https://drive.google.com/file/d/1ydzhJ5_A9-jTmSaUcHwq30odtWWgiora/view?usp=sharing

map3-videl:https://drive.google.com/file/d/1rRiid5Ki89aoeOfSb-PZ78XQLg_lnxbo/view?usp=sharing

	Steps
Map 1	16
Map 3	16

The reason why this algorithm can not work on map2 is that when it works in map2, it will moves into a situation like this:



Pic2 bug for map2 in simultaneously control

The potential collision that makes the system stuck is between robot 4 and robot 5. Since the robot 1 and 0 have reached their goal, it will be seen as obstacles at this time. Since either robot 4 or 5 cannot make progress toward its goal without the other reversing its progress, the system meets deadlock.

Although the system does not have a 2-cell wide path, since robot 0 and 1 reach the goal and I fix them after they move there, it creates a 1-cell wide path.

I think if the time allows me to do a further modification, I will change the priority of those robots which reach their goals and let them move in this situation. Unfortunately, I got covid and had a serious fever recently, so I really do not have more energy to deal with this problem. Maybe I will try to do further modification after I recover.

D.
I have written the report of this part from page 6-9.