CS598KKH Final Exam

Chaoqi LIU

Fall 2023

Multi-robot path planning on a grid

(A) MAPF is NP-complete, it can be considered as a generalization of the sliding tile puzzle which is known to be NP-complete (Ratner and Warrnuth 1986). Let's introduce some notations, the precomputed paths $P = \{p_i\}_{i=1}^N$, robots $R = \{r_i\}_{i=1}^N$, starts $S = \{s_i\}_{i=1}^N$, goals $G = \{g_i\}_{i=1}^N$. It is obvious the lower bound is $O(\sum_{i=1}^N |p_i|)$ where $|p_i|$ represents the length of path p_i. This lower bound says if there are no conflicts between paths in P, then, the number of solution steps is the sum of all steps in each path. However, such situation is rare, conflicts appear in most cases. Let's denote a conflict between agents a_i, a_j in grid v at time t, $C = (a_i, a_j, v, t)$. Given P, we can find all conflicts. For example, consider the example (Figure 2) below, let's name these grids $\begin{bmatrix} v_{11} & v_{12} \\ v_{21} & v_{21} \\ v_{31} & v_{32} \end{bmatrix}$, given $p_1 = v_{11} \to v_{21} \to v_{31}, p_2 = v_{12} \to v_{22} \to v_{32}, p_3 = v_{31} \to v_{21} \to v_{11}$. Then, we are able to know at time 0, a_1 in v_{11}, a_2 in v_{12}, a_3 in v_{31}; at time 1, a_1 in v_{21}, a_2 in v_{12}, a_3 in v_{31}; at time 2, a_1 in v_{21}, a_2 in v_{22}, a_3 in v_{31}; at time 3, a_1 in v_{21}, a_2 in v_{22}, a_3 in v_{21}. We just see one conflict, $(a_1, a_3, v_{21}, 3)$. Similarly, we can continue rollout and find all conflicts. Denote the number of conflicts found c, due to exchange is $O(1)$, we can raise the lower bound to $O(c + \sum_{i=1}^N |p_i|)$.

Next, I'd like to show that we cannot really find a meaningful upper bound for this method. Consider the example in Figure 1. We can see under such configuration, at time $t = l$, we just moved agent1 a_1, and it is a_2's turn, but a_1 blocks a_2's way, as a result, a_1 need to leave its current grid and make a space for a_2. Where can a_1 go? Obviously, down is not possible because it is blocked by a_k which is going to move at the end of this round. The options left are up or left. In my algorithm, I used Manhattan distance (L_1), because robots cannot go diagonal, that's why we should prefer L_1 distance. Under this metric, we realize that going left and right is "equally good" to the algorithm, and if this is really unlucky, the planner picks to go up. Now we come to the time $t = l + 1$, and similarly, again and again, agent a_1 eventually goes to the very top as shown in $t = l + k - 2$. This is just one extreme case robots may meet, but this illustrates that we cannot really tell the number of such cases, and we also cannot assign a big-O to each case. As a result, I don't think it is able for us to derive a meaningful upper bound for this planner.

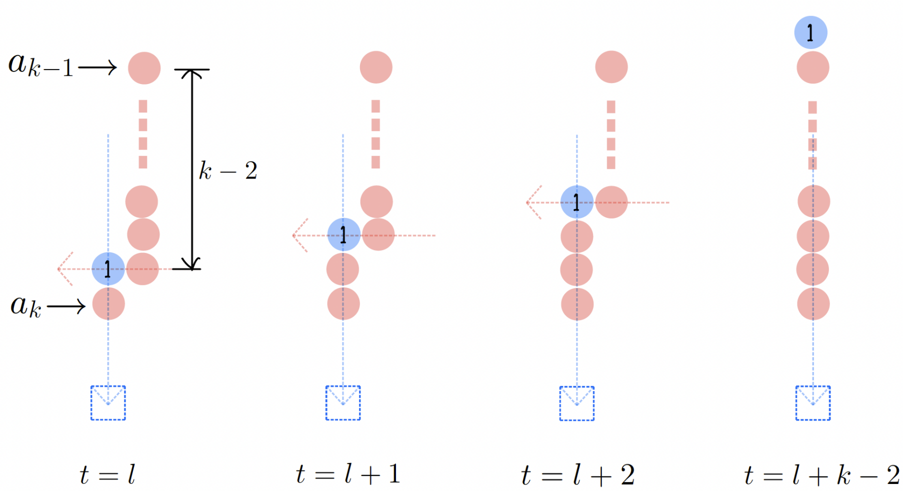

Figure 1: Blue dash line box is the goal grid of agent1 a_1 (marked in blue). The blue dash vertical line is the path p_1 for a_1, red horizontal dash lines are paths $p_2, p_3, ..., p_{k-1}$ for agents $a_2, a_3, ..., a_{k-1}$.

The method is not complete, consider the following counterexample.

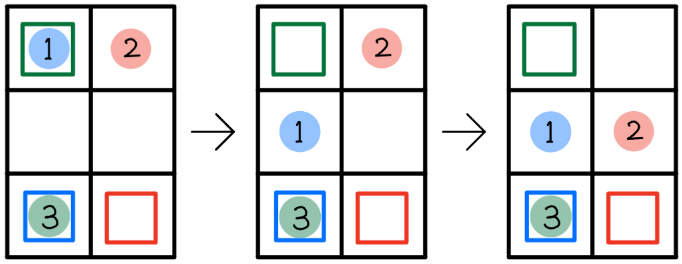

Figure 2: Circles represent agents in the map, numbers on the agents represent the order of execution, colorful squares represent goals. The left most configuration is the initial configuration. Both initial configuration and the goal configuration are "unpacked".

Obviously, in this order, failure will be reported. When it is the agent3's turn, it requires exchange, however, no 2×2 "unpacked" squares can be found – condition 1, "at most one other robot B", is not satisfied. For this technique to succeed, we need "unpacked" squares whenever agents require exchange.

For the proposed method to succeed, we need the following conditions

- Unpacked and conflict-free start and goal configuration.
- No narrow passages less than 2-cell across.
- Given the precomputed paths $P = \{p_i\}_{i=1}^N, \nexists C = (a_i, a_j, v, t)$ (i.e., a conflict between agent a_i, a_j in grid v at time t) where we cannot apply exchange policy – no unpacked 2×2 square around. For example, in the counter-example, we have conflict $(a_1, a_3, v_{21}, 3)$ and no exchange policy can be applied.

Some naive thoughts: I think this is very similar to the halting problem, I don't know if this analog is accurate, but given a MAPF instance and this planner, I don't think it is able to tell the running time and if it is going to succeed or fail, unless we run / simulate it. First, let's think about running time, I understand this can definitely be bounded by some function, but don't think the bound will be meaningful. As I described above, we cannot tell how many exchange we are going to apply due to cases like the one in Figure 1 by directly observing the MAPF instance. If we are allowed to use a variable e to represent the number of exchanges it will encounter, then the running time is $O(e + \sum_{i=1}^N |p_i|)$, but the problem is I see no relationship between P and e unless we simulate. Next, I don't think there exists a program that given the MAPF instance and the planner, it can output SUCCESS or FAILURE without simulate, this said, no preconditions checklist we can come up with to determine it is going to succeed or fail. For example, how can we foreseen the case in Figure 1 without reasoning step by step (in term of time t), which is simulation.

(B) In this prompt, we have lots of design choices, my algorithm included some nice design choices which will be shown later. The pesudocode is presented below.

Algorithm 1 Nonsimultaneous MAPF Solver

Require: map M, robots $R = \{r_i\}_{i=1}^N$, starts $S = \{s_i\}_{i=1}^N$, goals $G = \{g_i\}_{i=1}^N$, paths $P = \{p_i\}_{i=1}^N$, *replan*
  Initialize output command string *result* $\leftarrow \varnothing$, step count *step* $\leftarrow 0$
  moved $\leftarrow$ TRUE
  while $\exists r_i \neq g_i$ and *moved* = TRUE **do**
    moved $\leftarrow$ FALSE
    for $i \leftarrow 1$ to N **do**
      if *replan* = TRUE **then** ▷ replan paths for all agents, details later

```
        for i ← 1 to N do
            p_i ← BFS(r_i, g_i, M)        ▷ BFS has linear running time, in grid cases, the best option
    if r_i = g_i then                                               ▷ current robot is at its goal
        skip its round
    waypoint ← GetClosestPathWaypoint(r_i, p_i, M)       ▷ get the closest (L_1) point on path
    if r_i = waypoint then                                                 ▷ robot is on the path
        desired coord ← next point on the path                            ▷ move along the path
    else if r_i ≠ waypoint then                                             ▷ deviate from path
        desired coord ← greedily pick the (u, r, l, d)-point closest to path   ▷ move back greedily
    occupied ← CheckIfOccupied(desired coord, M) ▷ NONE if not occupied otherwise robot
    if occupied = NONE then
        moves ← MovePassiveAgent(r_i, occupied, M)          ▷ local planner, pesudocode later
        if moves = NONE then                            ▷ local planner failed to find solution
            just skip this one, possibly we have following agent can be moved
    else
        moves ← (r_i, ActionFromCoords(r_i, desired coord))
    moved ← TRUE                              ▷ luckily, we have agents moved if we reach this line
    for move in moves do
        step ← step + 1
        result ← result · moveStr(move) · WHITE_SPACE ▷ moveStr is given in starter code
        r_j, a_j ← move
        moveRobot(r_j, a_j)                               ▷ moveRobot is given in starter code
        optional: visualize
if ∀i, r_i = g_i then
    report success
    return result, step
else
    report failed
```

Note that my algorithm incorporates two strategies to handle robots deviation from path. Robots would deviate from their path if it is not in its round, and another robot want to take its current grid, then, it has to move away to make a space for another robot because it is that robot's round. The first strategy is to replan, and this should be the best option we have, when it's the robot's round, and it is not on its precomputed path, replan will get a new path for it, and hence, no need to go back to its precomputed path. Second option here is just greedily move back to its precomputed path, one can see this part in my above pseudocode clearly.

I will not present the pseudocode for BFS, GetClosestPathWaypoint, CheckIfOccupied, ActionFromCoord since they are too trivial, but I will breifly explain. BFS is breath-first-search, it has linear time, the prompt suggests to use Dijkstra, but in the grid world, edge costs are 1, Dijkstra takes no advantage but a larger running time, thus, we should use BFS, or DFS, such linear time search algorithm. GetClosestPathWayPoint, this just find the closest point on the computed path to the robot's current position, under L_1 metric, just loop through all points on the path, pick the closest one, that's it. CheckIfOccupied, query that grid on map M, if no robots are there, return NONE, otherwise, return the robot index in that grid. ActionFromCoord, returns u, r, d, r given a *from coord* and a *to coord*, assert distance is 1 between them.

Next, I'm going to present pseudocode for the local planner that coordinates two robots under situation described in the above pseudocode – need coordination / has potential conflict. The high level logic behind is when active agent a_i in its round, and want to take a_j's current grid, in such case, we need a_j to move away from its current grid and make a space for a_i. To resolve this, we go through all possible grids a_j can go to, they are four adjacent grids of a_j, as well as those in the surrounding unpacked squares. Next, we need to check if these potential grids to go to are valid. Then, among those valid grids, we go to the one that is "cloest" to a_j's goal. Pseudocode below.

Algorithm 2 Move Passive Agent – local planner

Require: active robot r_i and its path p_i goal g_i, passive robot r_j and its path p_j goal g_j, map M

I define active robot to be the robot that want to move to its desired grid because this is its round, passive robot to be the robot that is in the grid which active robot want to be in. We should notice that, under our case, L_1 distance between r_i, r_j should be 1.

```
unpacked squares ← ∅    ▷ store all possible unpacked 2 × 2 squares around for them to exchange
if r_i, r_j in the same column then
    top ← max(r_i.y, r_j.y)                                  ▷ point.y queries the y component
    bottom ← top - 1
    for offset ∈ {0, 1} do
        left ← r_i.x − 1+ offset                             ▷ point.x queries the x component
        right ← left + 1
        square ← the square bounded by top, right, bottom, left
        if IsUnpackedSquare(r_i, r_j, square) then
            unpacked squares ← unpacked squares ∪ {square}
else if r_i, r_j in the same row then
    right ← max(r_i.x, r_j.x)
    left ← right - 1
    for offset ∈ {0, 1} do
        bottom ← r_i.y − 1+ offset
        top ← bottom + 1
        square ← the square bounded by top, right, bottom, left
        if IsUnpackedSquare(r_i, r_j, square) then
            unpacked squares ← unpacked squares ∪ {square}
possible evade coords ←

        (r_j.x, r_j.y + 1), (r_j.x, r_j.y − 1), (r_j.x + 1, r_j.y), (r_j.x − 1, r_j.y)

    ▷ store possible grids that passive robot can go to, initialized to the adjacent (u, r, d, r)-grids
for square in unpack squares do
    add all grids in square to possible evade coords except passive robot's current grid
valid evade coords ← ∅                        ▷ store all valid grids that passive robot can go to
for coord in possible evade coords do
    if no obstacle in coord and no other robot in coord then
        valid evade coords ← valid evade coords ∪{coord}
if valid evade coords is empty then                          ▷ means no valid place to evade
    report no valid coordination between robot r_i, r_j
    return NONE                ▷ as required by the global planner, we need to return NONE
for coord in valid evade coords do
    waypoint ← GetClosestPathWaypoint(r_j, p_j, M)
    if waypoint is the closest to g_j so far then
        best evade coord ← coord
if best evade coord and r_j has distance 2 under L_2 metric then
    return

        [(r_j, ActionFromCoords(r_j, best evade coord)), (r_i, ActionFromCoords(r_i, r_j))]

            ▷ passive robot step out of any possible unpack squares, no exchange policy required
else        ▷ passive robot will go to one grid in some unpacked square, exchange policy required
    if unpack squares is empty then  ▷ no 2 × 2 unpacked squares, exchange condition not satisfied
        report no valid coordination between r_i, r_j
        return NONE
    for square in unpacked squares do
```

if POINTINSQUARE(*best evade coord*, *square*) **then**
exchange square ← *square*
break
exchange plan ← EXCHANGE2X2(r_i, r_j, r_j, *best evade coord*, *exchange square*) ▷ provided
exchange moves ← EXCHANGETOMOVES(*exchange plan*) ▷ provided
return *exchange moves*

I'm not going to present pseudocode for trivial functions in the local planner, but I'll explain. POINTINSQUARE, return TRUE if the point is inside the square, otherwise FALSE. ISUNPACKEDSQUARE, it basically checks if all conditions mentioned in the prompt are satisfied, and then return TRUE / FALSE.

For map1, the proposed method cannot solve it, this is exactly the same counter-example I provided in part (A). The reason is simple, agents start from four corners and they move to the central tunnel with width only two, meaning that they are going to meet in the tunnel and cannot progress.

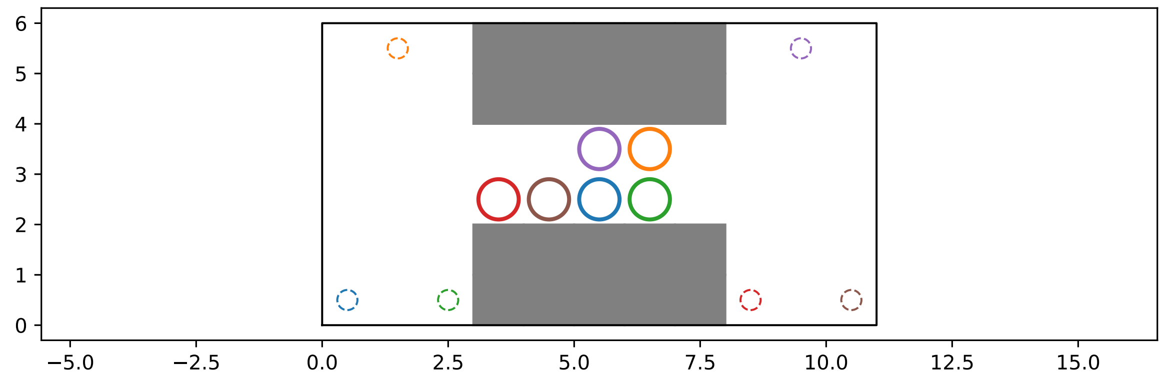

Figure 3: Map1 failure

For map2, it takes 41 steps with replan, output commands are Al Bd Cl Dd Ed Fl Al Bd Cl Dr Er Fl Br Cl Dr Er Fl Br Al Cl Dr Fu Er Fl Fl Cd Ar Cl Br Eu Cl Er Ed Fl Cl Er Fl Cl Er Er Er; and 43 steps without replan and use greedy strategy, output commands are Al Bd Cl Dd Ed Fl Al Bd Cl Dr Er Fl Br Cl Dr Er Fl Br Al Cl Dr Fu Er Fl Fd Cd Ar Cl Br Cu Er Fl Cl Er Fl Cl Er Fl Cl Er Fu Cd Er.

For map3, it takes 80 steps with replan, output commands are Au Bl Cl Du Eu Fu Al Bu Cl Eu Dr Dl Ed Fr Al Bl Cu Er Dr Fr Er Fr Fu Al Fr Bl Cl Dr Er Eu Al Er Bl Cl Dr Cr Fd Cu Du Al Dr Bl Cl Ed Fr Al Bl Cl Dd Er Fr Al Bl Cl Dr Er Fd Al Bl Cl Dr Er Fd Ad Bd Cl Dr Ed Fr Ad Bd Cd Dr Ed Fr Cd Dd Er Cd Dd; and 86 steps without replan and use greedy strategy, output commands are Au Bl Cl Du Eu Fu Al Bu Cl Eu Dr Dd Ed Fr Al Bl Cu Er Du Fr Er Fr Fu Al Fr Bl Cl Dr Er Eu Al Er Bl Cl Dr Cr Fd Cu Du Al Dr Bl Cd Ed Cu Fr Cl Al Bl Cd Dd Cu Er Cl Fr Al Bl Cd Cu Dr Cl Er Fd Al Bl Cd Dr Er Fd Ad Bd Cl Dr Ed Fr Ad Bd Cl Dr Ed Fr Cd Dd Er Cd Dd.

(C) For this part, we also have lots of design choices, through discussion with prof. Kris, I think it is ok to have a planner that freeze all other robots if there is exchange taking place, and the planner finishes exchange simultaneously first, then everything go back to normal setting, where robots move simultaneously across the map. This method is easier to implement, but it is not efficient, because it is possible that other robots can also move when there is exchange taking place, and even more, we can have more than one exchange taking place. As a result, I'm going to provide a planner that can simultaneously move all robots across the map, provided in the meanwhile, we can have exchanges taking place.

Algorithm 3 Simultaneous MAPF Solver

Require: map M, robots $R = \{r_i\}_{i=1}^N$, starts $S = \{s_i\}_{i=1}^N$, goals $G = \{g_i\}_{i=1}^N$, paths $P = \{p_i\}_{i=1}^N$, *replan*

```
Initialize output command string result ← ∅, step count step ← 0
robot in exchange ← ∅
exchange threading ← {∀i, r_i : NONE}                                        ▷ dictionary
robot has queued action ← ∅
queued action ← {∀i, r_i : NON}                                              ▷ dictionary
moved ← TRUE
while ∃r_i ≠ g_i and moved = TRUE do
    moved ← FALSE
    grids registered ← ∅                                   ▷ grids that cannot be entered
    moves executed this step ← ∅          ▷ store moves in this propagation step, for output purpose
    for i ← 1 to N do
        if replan = TRUE then
            for i ← 1 to N do
                p_i ← BFS(r_i, g_i, M)
        if robot in exchange is not empty then
            for robot, policy in exchange threading with policy ≠ NONE do
                add all grids in the 2 × 2 square of this policy to grids registered
        if r_i in robot in exchange then        ▷ robot under exchange, just run precomputed policy
            partner, square, moves, step ← exchange threading[r_i]
            robot to move, action ← moves[step]
            if r_i = robot to move then        ▷ this is r_i's move, otherwise, its partner will progress
                MoveRobot(r_i, action)
                add moves[step] to moves executed this step
                moved ← TRUE
                step ← step + 1
                if step = size of moves then     ▷ no more move in moves, reset exchange threading
                    remove r_i from robot in exchange
                    remove partner from robot in exchange
                    set exchange threading[r_i] to NONE
                    set exchange threading[partner] to NONE
                else                                             ▷ update exchange threading
                    exchange threading[r_i] ← (partner, square, moves, step)
                    exchange threading[partner] ← (r_i, square, moves, step)
        else if r_i in robot has queued action then          ▷ robot has queued action, details later
            action ← queued action[r_i]
            MoveRobot(r_i, action)
            add (r_i, action) to moves executed this step
            moved ← TRUE
            remove r_i from robot has queued action
            set queued action[r_i] to NONE
        else if r_i = g_i then
            skip its round
        else                                                    ▷ need to plan for this robot
            waypoint ← GetClosestPathWaypoint(r_i, p_i, M)
            if r_i = waypoint then
                desired coord ← next point on the path
            else if r_i ≠ waypoint then
                desired coord ← greedily pick the (u, r, l, d)-point closest to path
            occupied ← CheckIfOccupied(desired coord, M)
            if occupied = NONE then
                moves, exchanged, square ← MovePassiveAgent(r_i, occupied, M)
                if moves = NONE then
                    just skip this one, possibly we have following agent can be moved
```

```
            else
                moves ← (r_i, ActionFromCoords(r_i, desired coord))
            if exchanged = TRUE then
                add r_i to robot in exchange                          ▷ register an exchange thread
                exchange threading[r_i] ← (occupied, square, moves, 0)
                add occupied to robot in exchange                     ▷ register an exchange thread
                exchange threading[occupied] ← (r_i, square, moves, 0)
                add all grids in square to grids registered           ▷ register the square
                robot to move, action ← moves[0]
                if r_i = robot to move then
                    MoveRobot(r_i, action)
                    add moves[0] to moves executed this step
                    moved ← TRUE
                    if 1 = size of moves then
                        remove r_i from robot in exchange
                        remove partner from robot in exchange
                        set exchange threading[r_i] to NONE
                        set exchange threading[partner] to NONE
                    else
                        exchange threading[r_i] ← (partner, square, moves, 1)
                        exchange threading[partner] ← (r_i, square, moves, 1)
            else if size of moves > 1 then            ▷ no exchange required, but multiple moves
                we can assert that size of moves is 2                 ▷ checkout MovePassiveAgent
                passive robot, passive action ← moves[0]
                active robot, active action ← moves[1]
                add passive robot to robot has queued action          ▷ will explain
                queued action[passive robot] ← passive action
                current coord ← passive robot
                next coord ← NextCoord(passive robot, passive action)
                add current coord to grids registered
                add next coord to grids registered
            else                                      ▷ no exchange required, and only one move
                robot to move, action ← moves[0]
                current coord ← robot to move
                next coord ← NextCoord(robot to move, action)
                if next coord in grids registered then
                    continue, because this move is not valid at the moment
                MoveRobot(robot to move, action)
                add move[0] to moves executed this step
                moved ← TRUE
                add current coord to grids registered
                add next coord to grids registered
        if moved = FALSE but robot in exchange or robot has queued action is not empty then
            moved ← TRUE                              ▷ because we still can do something
    optional: visualize
    if moves executed this step is not empty then
        step ← step + 1
        result ← result · moveStr(moves) · WHITE_SPACE
if ∀i, r_i = g_i then
    report success
    return result, step
else
    report failed
```

TL;DR: basically keep track of robots that are exchanging, and each time it is their turn, execute one move in the exchange policy.

In the above pseudocode, gray part is adapted from what we have in nonsimultaneous planner, modification is indicated in blue. Here, *exchanged* is a boolean value indicating if we have exchange policy, *square* is the 2×2 square exchange taking place. This said, we need modification to MovePassiveAgent as well, but this is very trivial since we know which if-else branch we are in, we can easily tell the value of *exchanged*, and that's even more trivial for us to tell the exchange square, I'm not going to present the modified pseudocode here. For NextCoord, this is also very straightforward, given current coordinate and action (u, r, b, l), we just return the next coordinate.

Why we need queued action / robot? Because in MovePassiveAgent, one possible outcome is no exchange requires because we can move passive robot one step away any unpacked squares, and then, move the active agent one step (I explained why it is always 1 in our case) to its desired position (the one occupied by passive agent). Now, we know under such situation, the returned plan by local planner is $\big[(\textit{passive robot, passive action}), (\textit{active robot, active action})\big]$ in sequence. Next, why we queue the passive agent's movement? First, we should realize that in such situation, we cannot move active agent until passive agent leaves its current position, second, it takes a little bit time to realize that, we cannot move the passive agent anymore and we should save this step to next its round. Situation (1), passive robot has a smaller index than active robot, in this case, we already have passive robot moved within this round, we cannot move it anymore, so let's save it to next round; situation (2), passive robot has a larger index than active robot, in this case, there is not harm to queue its action, we just plan for it in this step, and execute it when it is passive robot's turn.

The preformance is definitely amazing. For map1, as analyzed before, this is not going to be solved by this proposed method. For map2, it takes 12 steps with replan, output commands are AlBdClDdEdFl AlBdClDrErFl BrClDrErFl BrDrFu AlBrClErFl CdFl ArClFl CuErFl ClEr ClEr ClEr CdEr; and 12 steps without replan, output commands are AlBdClDdEdFl AlBdClDrErFl BrClDrErFl BrDrFu AlBrClErFl CdFd ArClFl CuErFl ClErFl ClErFu ClEr CdEr. For map3, it takes 22 steps with replan, output commands are AuDuEuFu AlBuClEuFr AlBrClDrFr AlBlCuDrEd AuClDrFr AlBlClEr AlBlCuDrFr AlClErFl AlBlCl AlBuClFr AdBlClFr AdBlCdDrErFr AdBlCdDrFd BlCdDrErFd BlDrErFr BdDdErFr BdDdEr BdEd DrEd Du DlEr Dd; and failed without replan, this is greedy strategy failed to figure its way out and put all robots into one place – the tunnel. But it is hard to say why the nonsimultaneous planner worked on map3 without replan, I think one possible reason is this simultaneous planner greedily adds possible robot move to execute in one time step, and this may leads to deaklock or other situation that leads to failure.

(D) First, as what I showed in previous part, replan is definitely a good strategy to improve efficiency, refer to results in previous parts. Second, if replan is adapted, we can avoid running BFS each time by precompute heatmaps for each agent – backward full BFS that produces cost-to-go in each grid, then, during online execution, we just follow the Bellman optimality condition to retrieve the next best step that drives the agent to its goal. Additionally, with experiments, one map3, it is possible that robot A's path is making too much trouble for another robot B which is already at its goal – the A' path generated has B on its way, in this case, unnecessary steps appear. For example, the output commands for map3 with replan is "... DrEd Du DlEr Dd", the ending commands basically makes robot D out of its goal grid, and make robot E to go through D's goal grid, and then D goes back to its goal, this is ridiculous. This said, one heuristic we can have is take robot which is already in its goal state as obstacle in BFS search. This definitely helps, I'm not going to present results here, because all these are hacks to make this proposed method better in a empirical manner.

Through discussion with prof. Kris, I think the best option is to present and analyze some SOTA method in MAPF community. To my best knowledge, it is hard to have a multi-agent planner that is both optimal and complete, but there do exist, LaCAM* (Keisuke Okumura, 2023), this is very complicated, I would not be able to analyze such planner, but I would love to mention this recent amazing planner.

Instead, in this part, I'm going to introduce Collision Based Search (CBS) by Sharon *et al.*, 2015, which is solution complete and optimal, by solution complete, we mean the planner ensures to find solutions for solvable instances but it never identifies unsolvable ones. The reason is CBS is the foundation of many recent MAFP algorithms, e.g., Enhanced CBS (ECBS), Explicit Estimation CBS (EECBS). CBS is not particularly amazing in MAPF, because it cannot handle large number of agents, but such idea / method / framework is definitely worth discussing. The following is largely informed by external resources, but rephrased and summarized with my own words, links are attached.

Problem definition: In MAPF problem, we have graph $G = (V, E)$, a set of k agents with labels $a_1, ..., a_k$. Start positions $s_i \in V$, goal positions $g_i \in V$. At each time step, agent a_i can move to its adjacent position or wait in its current position. The goal is to return a set of actions for each agent, so that it can move to its goal from start without conflicting other agents.

Introduce CBS: The state space of MAPF is exponential in k the number of agents. However, in a single-agent pathfinding problem, the state space is linear in the graph size. CBS solves the MAPF problem by decomposing it into a large number of single-agent pathfinding problems. Each problem is relatively simple to solve, while there may be an exponential number of such single-agent problem. This is the essential idea behind CBS and the following recently developed SOTA MAPF solvers. To proceed, let's introduce some notion throughout following discussion. By *path*, we mean the path for one agent, while we use *solution* to represent k paths for k agents. We define *constraint* for a given agent a_i be (a_i, v, t), which says agent a_i cannot be in position v at time step t. A *consistent path* for agent a_i is a path that satisfies all its constraints, similarly, a *consistent solution* is a solution that made up from k consistent paths. A *conflict*, (a_i, a_j, v, t), means agent a_i and a_j occupy position v at time step t. A solution is *valid* if all its paths have no conflicts. A consistent solution can be *invalid* if, despite the fact that the paths are consistent with their individual agent constraints, there paths still have conflicts. Again, I emphasize that the key idea of CBS is to grow a set of constraints for each agent and find paths that are consistent with the constraints. This is the ultimate contribution of CBS in MAPF community. If these paths have conflicts, and are thus invalid, the conflicts are resolved by adding new constraints. CBS does this by working in two levels. At the high level, conflicts are found and constraints are added, then, the low level updates the agents paths to be consistent with the new constraints.

High level: Search the Constraint Tree (CT): CBS searches a *constraint tree* (CT), which is a binary tree. Each node N in CT contains (1) A set of constraints (can be queried by $N.constraints$). Note that root of CT contains an empty set of constraints. The child of a node in the CT inherits the constraints of the parent and adds one new constraint for one agent. (2) A solution (can be queried by $N.solution$). The k paths are found by low level. (3) The total cost (can be queried by $N.cost$) of the current solution. We denote this cost the f-value of the node. Node N in the CT is a goal node when $N.solution$ is valid. The high level performs a best-first search on the CT where nodes are ordered by their costs. Ties are broken by using a *conflict avoidance table* (CAT), this is a dynamic lookup table under the *Independence Detection* (ID) framework.

Processign a node in the CT: Given $N.constraints$ in the CT, the low level search in invoked. This search returns one shortest path (SP) for each agent a_i that is consistent with all constraints associated with a_i in node N. Once a consistent path has been found for each agent w.r.t. its constraints, these paths are then *validated* w.r.t. other agents. The *validation* performed by simulating the set of k paths. If all agents reach their goal without any conflict, this CT node N is declared as the goal node, and the current solution $N.solution$ is returned. If, however, while performing the *validation* a conflict $C = (a_i, a_j, v, t)$ is found for two or more agents a_i, a_j, the validation halts and the node is declared as a non-goal node.

Resolving a conflict: Given a non-goal CT node N whose solution $N.solution$ includes a conflict $C_n = (a_i, a_j, v, t)$, at least one constraints (a_i, v, t) or (a_j, v, t) should be added to $N.constraints$. To guarantee optimality, both possibilities are examined and N is split into two children. Both children inherit $N.constraints$. The left child resolves the conflict by adding (a_i, v, t) and the right child adds (a_j, v, t).

Algorithm 4 High level of CBS

Require: MAPF instance
  $R.constraints \leftarrow \emptyset$
  $R.solution \leftarrow$ find individual paths using the LowLevelCBS()
  $R.cost \leftarrow$ cost of $R.solution$
  insert R to $OPEN$ ▷ $OPEN$ is the standard queue in search
  while $OPEN$ is not empty **do**
    $P \leftarrow$ best node from $OPEN$ ▷ lowest solution cost
    Validate the paths in P until a conflict occurs
    if P has no conflict **then**
      return $P.solution$ ▷ P is goal
    $C \leftarrow$ fist conflict (a_i, a_j, v, t) in P
    for each agent a_i in C **do** ▷ there are $\{a_i, a_j\}$
      $A \leftarrow$ new node
      $A.constraints \leftarrow P.constraints \cup (a_i, v, t)$ ▷ optimality can be proved
      $A.solution \leftarrow P.solution$
      Update $A.solution$ by invoking LowLevelCBS(a_i)
      $A.cost \leftarrow$ cost of $A.solution$
      Insert A to $OPEN$

Low level: Find Paths for CT Nodes: The low level is given an agent a_i, and a set of associated constraints. It performs a search in the underlying graph to find an optimal path for a_i that satisfy all its constraints. Agent a_i is solved in a *decoupled manner*, i.e., ignoring the other agents. This search is 3-dimensional, as it includes two spatial dimensions, and one time dimension, i.e., (x, y, t). We can use whatever search to generate SP for this single-agent pathfinding problem.

By now, I discussed this solution complete and optimal MAPF framework CBS, this is much better than the proposed method. I'll argue in two perspectives. First, the proposed method cannot handle map1 as I explained in previous parts, but it is obvious there exists solution in that map – the stupidest commands, just move robot A to its goal with all other robots freezed, this is possible because it is so obvious such path exists, the same for all other robots. Recall CBS is solution complete, meaning CBS can find a solution for map1, and map2, map3 as well. Second, CBS produces optimal solution, it is obvious at the moment, the proposed method generates nonoptimal solutions, and even not bounded suboptimal solutions, the resulting commands are attached in previous parts. CBS, however, can solve these three maps optimally.

But I haven't prove that CBS is solution complete and optimal. Let's prove that CBS will returns an optimal solution (optimality) if one exists (solution complete).

Definition 1 For given node N in a CT, let $CV(N)$ be the set of all solutions that are: (1) consistent with the set of constraints of N and (2) are also valid, i.e., no conflict.

If N is not a goal node, then the solution of N will not be part of $CV(N)$ because it is not valid.

Definition 2 For any solution $p \in CV(N)$ we say that node N permits the solution p.

The root of the CT, for example, has an empty set of constraints. Any valid solution satisfies the empty set of constraints. Thus the root node permits all valid solution. The cost of a solution in $CV(N)$ is the sum of the costs of the individual agents. Let $minCost(CV(N))$ be the minimum cost over all solution in $CV(N)$.

Lemma 1 The cost of a node N in the CT is a lower bound on $minCost(CV(N))$. **proof:** $N.cost$ is the optimal cost of a set of paths that satisfy $N.constraints$. This set of paths is not necessarily a valid solution. Thus, $N.cost$ is a lower bound on the cost of any set of paths that make a valid solution for N as no single agent in any solution can achieve its goal faster.

Lemma 2 Let p be a valid solution. At all time steps there exists a CT node N in $OPEN$ that permits p. **proof:** By induction on the expansion cycle: For the base case $OPEN$ only contains the root node, which has no constraints. Consequently, the root node permits all valid solutions and also p. Now, assume this is true for the first i expansion cycles. In cycle $i+1$, assume that node N, which permits p, is expanded and its children N_1', N_2' are generated. Any valid solution in $VS(N)$ must be either in $VS(N_1')$ or $VS(N_2')$, as any valid solution must satisfy at least one of the new constraints.

One step further, we realize that at all times at least one CT node in $OPEN$ permits the optimal solution (as a special case of lemma 2).

Theorem CBS returns the optimal solution. **proof:** Consider the expansion cycle when a goal node G is chosen for expansion by the high level. At that point all valid solutions are permitted by at least one node from $OPEN$ (lemma 2). Let p be a valid solution (with cost $c(p)$) and let $N(p)$ be the node that permits p in $OPEN$, let $c(N)$ be the cost of node N. We then have $c(N(p)) \leq c(p)$ (lemma 1). Since G is a goal node $c(G)$ is a cost of a valid solution. Since the high level search explores solution costs in a best-first manner, we eventually have $c(g) \leq c(N(p)) \leq c(p)$.

For detailed proof and other theoretical analysis on situations that CBS is greatly better then existing MAPF solvers (back in 2015), one should refer to the origin paper, links are all attached before.