16-891 Multi Robot Planning and Coordination

Instructor : Jiaoyang Li   TAs : Indraneel Patil, Chao Cao

Spring 2023 HW1 : Multi-Agent Path Finding (MAPF)

In this assignment, you will learn about Multi-Agent Path Finding (MAPF) and implement a single-agent solver, namely space-time A*, and parts of four MAPF solvers, namely Joint-State A*, Prioritized Planning, Conflict-Based Search (CBS), and Priority-Based Search (PBS). Skeleton code is given to make the assignment easier and more structured. You are highly encouraged to go through the appendices at the end of the assignment before attempting the assignment.

Acknowledgement

Parts of this assignment has been borrowed from the Summer School on Cognitive Robotics held in 2019 organised by Wolfgang Hönig, Jiaoyang Li, Sven Koenig

Honour Code

As a student you may discuss with your peers but at no point should you seek assistance on your code from them or from the internet. Your code submission should be yours and yours only. This assignment has a lot of learning and intuition waiting to be acquired if only you followed this Honour code and remained loyal to your learning!

Grading Rubrics

This assignment has 15 points in total: 10 for the coding part and 5 for the write-up. For the coding part, the point distribution and the number of test cases on Gradescope are given in the following table.

Algorithm	Points	Test Cases
Joint-State A*	2	5
Prioritized Planning	2	5
Conflict-Based Search	3	20
Priority-Based Search	3	20

To get all points for each path-finding solver, your implementations must pass all corresponding test cases on Gradescope. For Joint-State A* and Conflict-Based Search, your solvers are expected to find optimal paths with the minimum sum of costs. For Prioritized Planning and Priority-Based Search, your solvers are expected to find paths that are not worse than the optimal ones plus some cost margins (10 for Prioritized Planning and 30 for Priority-Based Search). Some test cases (`instances/test_*.txt`) are released with the code template for validating your implementation before submission. The 5 points of the write-up correspond to your written response to Task 1 (1 point), Task 2 (2 points), and Task 5 (2 points), evaluated on the correctness and clarity of the response.

Submission Details

Code submission: Compress your `single_agent_planner.py`, `joint_state.py`, `prioritized.py`, `cbs.py` and `pbs.py` into a zip file and submit it on Gradescope.
Write-up submission: Submit your write-up as a pdf file on Gradescope.

0 Task 0: Preparing for the Project

0.1 Installing Python 3

This project requires a Python 3 installation with the `numpy` and `matplotlib` packages. On Ubuntu Linux, download python by using:

```
sudo apt install python3 python3-numpy python3-matplotlib
```

On Mac OS X, download Anaconda 2019.03 with Python 3.7 from https://www.anaconda.com/distribution/#download-section and follow the installer. You can verify your installation by using:

```
python3 --version
```

On Windows, download Anaconda 2019.03 with Python 3.7 from https://www.anaconda.com/distribution/#download-section.

On Ubuntu Linux and Mac OS X, use `python3` to run python. On Windows, use `python` instead.

You can use a plain text editor for the project. If you would like to use an IDE, we recommend that you download PyCharm from https://www.jetbrains.com/pycharm/. The free community edition suffices fully, but you can get the professional edition for free as well, see https://www.jetbrains.com/student/ for details.

0.2 Installing the MAPF Software

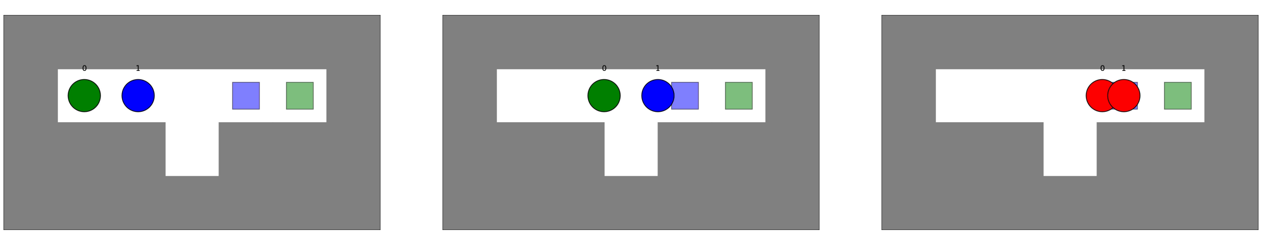
Download the archive with the provided MAPF software and extract it on your computer.

0.3 Understanding Independent Planning

Execute the independent MAPF solver by using:

```
python run_experiments.py --instance instances/exp0.txt --solver Independent
```

If you are successful, you should see an animation:



The independent MAPF solver plans for all agents independently. Their paths do not collide with the environment but are allowed to collide with the paths of the other agents. Thus, there is a collision when the blue agent 1 stays at its goal cell while the green agent 0 moves on top of it. In your animation, both agents turn red when this happens, and a warning is printed on the terminal notifying you about the details of the collision.

Try to understand the independent MAPF solver in `independent.py`. The first part defines the class `IndependentSolver` and its constructor:

```
class IndependentSolver(object):
    def __init__(self, my_map, starts, goals):
        # some parts are omitted here for brevity
        # compute heuristic values for the A* search
        self.heuristics = []
        for goal in self.goals:
            self.heuristics.append(compute_heuristics(my_map, goal))
```

The function `compute_heuristics` receives as input the representation of the environment and the goal cell of the agent and computes a look-up table with heuristic values (or, synonymously, h-values) for the A* search that finds a path for the agent, by executing a Dijkstra search starting at the goal cell.

The second part performs one A* search per agent:

```
    def find_solution(self):
         for i in range(self.num_of_agents):  # Find path for each agent
            path = a_star(self.my_map, self.starts[i], self.goals[i],
            ↪  self.heuristics[i], i, [])
            if path is None:
                raise BaseException('No solutions')
            result.append(path)
        return result
```

The function `a_star` receives as input the representation of the environment, the start cell of the agent, the goal cell of the agent, the heuristic values computed in the constructor, the unique agent id of the agent, and a list of constraints and performs an A* search to find a path for the agent. The independent MAPF solver does not use constraints.

1 Task 1: Implementing Joint-State A* and Space-Time A*

Section 1.1 covers Joint-State A* whereas section 1.2 to 1.4 are for Space-Time A*. Have fun!

1.1 Implementing Joint-State A*

A popular method for planning for small teams of agents in a centralized setting is joint state space planning where we plan for M agents in a state-space that represents joint configurations of agents. The approach is simple: Construct and search a graph where each state encodes the positions of all the agents and each action encodes all possible movements. You will now finish the `joint_state_a_star` function inside `joint_state.py`. Some prompts are given in the code to help your implementation.

You can test your Joint-State A* implementation by using:

```
python run_experiments.py --instance instances/exp0.txt --solver JointState
```

Answer the following question in your report: Assuming a 4-neighbor grid with a fleet of N agents what is the maximum branching factor?

You can also try running Joint-State A* with more complex test cases such as `instances/test_*.txt`. If the solver takes too long to find solutions, you can try to reduce the number of agents.

1.2 Searching in the Space-Time Domain

You may observe that Joint-State A* slows down drastically as the number of agents increases. It's time to upgrade our algorithms! To prepare for that, you now change the single-agent solver to perform a space-time A* search, which searches in cell-time space and returns a shortest path that satisfies a given set of constraints. Such constraints are essential for MAPF solvers such as prioritized planning and CBS.

The existing A* search in the function `a_star` in `single_agent_planner.py` only searches over cells. Since we want to support temporal constraints, we also need to search over time steps. Use the following steps to change the search dimension:

1. Your variables `root` and `child` are dictionaries with various key/value pairs such as the g-value, h-value, and cell. Add a new key/value pair for the time step. The time step of the root node is zero. The time step of each node is one larger than the one of its parent node.
2. The variable `closed_list` contains the processed (that is, expanded) nodes. Currently, this is a dictionary indexed by cells. Use tuples of (cell, time step) instead.
3. When generating child nodes, do not forget to add a child node where the agent waits in its current cell instead of moving to a neighboring cell.

You can test your code by using:

```
python run_experiments.py --instance instances/exp1.txt --solver Independent
```

and should observe identical behavior.

1.3 Handling Vertex Constraints

We first consider vertex constraints, that prohibit a given agent from being in a given cell at a given time step. Each constraint is a Python dictionary. The following code creates a vertex constraint that prohibits agent 2 from occupying cell $(3, 4)$ at time step 5:

```python
{'agent': 2,
 'loc': [(3,4)],
 'timestep': 5}
```

In order to add support for constraints, change the code to check whether the new node satisfies the constraints passed to the `a_star` function and prune it if it does not.

An efficient way to check for constraint violations is to create, in a pre-processing step, a constraint table, which indexes the constraints by their time steps. At runtime, a lookup in the table is used to verify whether a constraint is violated. Example function headers for the functions `build_constraint_table` and `is_constrained` are already provided. You can call `build_constraint_table` before generating the root node in the `a_star` function.

You can test your code by adding a constraint in `prioritized.py` that prohibits agent 0 from being at its goal cell $(1, 5)$ at time step 4 and then using:

```
python run_experiments.py --instance instances/exp1.txt --solver Prioritized
```

Agent 0 should wait for one time step (but when and where it waits depends on the tie-breaking).

1.4 Adding Edge Constraints

We now consider edge constraints, that prohibit a given agent from moving from a given cell to another given cell at a given time step.

The following code creates a edge constraint that prohibits agent 2 from moving from cell $(1, 1)$ to cell $(1, 2)$ from time step 4 to time step 5:

```
{'agent': 2,
 'loc': [(1,1), (1,2)],
 'timestep': 5}
```

Implement constraint handling for edge constraints in the function `is_constrained`.

You can test your code by adding a constraint in `prioritized.py` that prohibits agent 1 from moving from its start cell $(1, 2)$ to the neighboring cell $(1, 3)$ from time step 0 to time step 1.

Answer the following question in your report: Where is agent 0 at timestep 4 and where is agent 1 at timestep 1 on your output paths?

2 Task 2: Implementing Prioritized Planning

Prioritized planning finds paths for all agents, one after the other, that do not collide with the environment or the already planned paths of the other agents. To ensure that the path of an agent does not collide with the already planned paths of the other agents, the function `a_star` receives as input a list of constraints compiled from their paths. Prioritized planning is faster than CBS but it is incomplete and suboptimal.

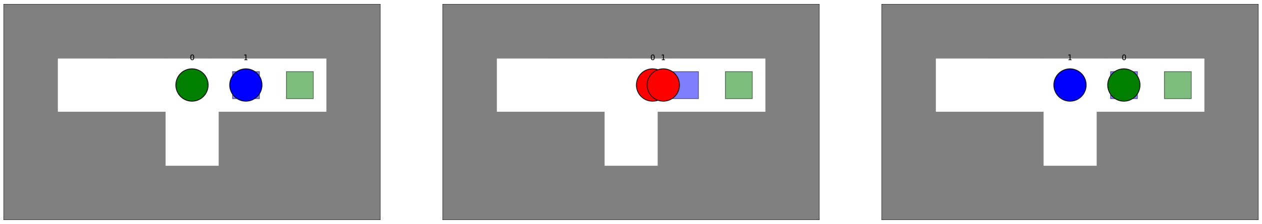
2.1 Adding Vertex Constraints

Add code to `prioritized.py` that adds all necessary vertex constraints. Transform the already planned paths of higher-priority agents into constraints.

You can test your code by using:

```
python run_experiments.py --instance instances/exp2_1.txt --solver Prioritized
```

Now, the blue agent does not stay at its goal cell when it reaches that cell for the first time:



Unfortunately, there is still a collision because both agents move to the cell of the other agent at the same time step. We thus need to add edge constraints as well.

2.2 Adding Edge Constraints

Add code to `prioritized.py` that adds all necessary edge constraints, and test your code as before. There are no more collisions.

Answer the following question in your report: In your report, describe how you added vertex and edge constraints.

2.3 Adding Additional Constraints

2.3.1 Case 1: Higher-Priority Agents Passing One's Goal Position in the "Future"

At this point, the A* search terminates when a node containing the goal position is expanded for the first time. However, there are cases where after an agent reaches its goal position, it would still need to move away from the goal position temporarily to give ways for other higher-priority agents to pass in the "future". For example, let's assume agent 0 reaches its goal position at timestep 2. And at timestep 10, agent 1 with a higher priority needs to pass through agent 0's goal position en route to its own destination. Therefore, agent 0 cannot be at the goal position at timestep 10. To handle such scenarios, the A* search for agent 0 cannot terminate right away at timestep 2, and a vertex constraint needs to be added for agent 0 at timestep 10. Modify your code to handle such scenarios, where the reaching-goal condition check considers future constraints imposed by other agents with a higher priority.

2.3.2 Case 2: Higher-Priority Agents Staying at Their Goal Positions

Your code does not prevent all collisions yet since agents can still move on top of other agents that have already reached their goal locations. You can verify this issue by using the MAPF instance `exp2_2.txt` and assuming that agent 0 has the highest priority. You can address this issue by adding code that adds additional constraints that apply not only to the time step when agents reach their goal locations but also to all future time steps. One way to achieve that is by limiting the time horizon of the search. The shortest path of an agent cannot be infinitely long. So you can calculate an upper bound on the path length for an agent based on the path lengths of all agents with higher priorities and the size of the environment. Then you can simply add vertex constraints at a higher-priority agent's goal position from the moment it reaches its goal position throughout the entire time horizon. Another way to address this is by switching back to the vanilla A* (without the time dimension and the "wait" action) after all other higher-priority agents reach their goals.

2.3.3 Case 3: Addressing Failures

The A* search might not be able to find a collision-free solution in some cases. Your implementation needs to handle such failure cases properly by terminating the A* search at the right time. In the MAPF instance exp2_3.txt, the priorities between agents 0 and 1 and are switched compared to exp2_2.txt. Rerun the experiment on instance exp2_3.txt. Make sure your solver terminates properly and reports "no solutions".

Answer the following question in your report: Document the changes you made in your code to handle the above cases. Also, state and justify the upper bound you used in your code.

2.4 Showing that Prioritized Planning is Incomplete and Suboptimal

Answer the following question in your report:

- Design a MAPF instance for which prioritized planning does not find an (optimal or suboptimal) collision-free solution for a given ordering of the agents.
- Design a MAPF instance for which prioritized planning does not find an (optimal or suboptimal) collision-free solution, no matter which ordering of the agents it uses.
- Design a MAPF instance for which prioritized planning does not find an (optimal or suboptimal) collision-free solution for a given ordering of the agents even if an ordering of the agents exists for which prioritized planning finds an optimal collision-free solution.
- Design a MAPF instance for which prioritized planning does not find an optimal collision-free solution, no matter which ordering of the agents it uses, even if a collision-free solution exists.

3 Task 3: Implementing Conflict-Based Search (CBS)

The independent MAPF solver finds paths for all agents, simultaneously or one after the other, that do not collide with the environment but are allowed to collide with the paths of the other agents. Conflict-Based Search (CBS) maintains a constraint tree to deal with these collisions between agents.

3.1 Detecting Collisions

Write code that detects collisions (or, synonymously, conflicts) among agents, namely vertex collisions where two agents are in the same cell at the same time step and edge collisions where two agents move to the cell of the other agent at the same time step.

Add code to `cbs.py` that implements the two functions `detect_collision` and `detect_collisions`. You should use `get_location(path,t)` to obtain the cell of an agent at time step t. You can test your code by using:

```
python run_experiments.py --instance instances/exp3_1.txt --solver CBS
```

You receive output similar to

```
[{'a1': 0, 'a2': 1, 'loc': [(1, 4)], 'timestep': 3}]
```

3.2 Converting Collisions to Constraints

The high level of CBS searches the constraint tree. Once it has chosen a node of the constraint tree for expansion and picked a collision of the paths of two agents in that node, it transforms this collision into two new constraints, one for each new child node of the chosen node. The first constraint prohibits the first agent from executing the colliding action, and the second constraint prohibits the second agent from executing the colliding action. For the vertex collision between agents 1 and 2 in cell $(1, 4)$ at time step 3, the set of new vertex constraints is:

```
[{'agent': 0, 'loc': [(1, 4)], 'timestep': 3},
 {'agent': 1, 'loc': [(1, 4)], 'timestep': 3}]
```

Add code to `cbs.py` that implements the function `resolve_collision` and test your code as above. Hint: You need to reverse the direction of the edge for the second agent to obtain the second edge constraint for an edge collision.

3.3 Implementing the High-Level Search

Algorithm 1: High-level search of CBS.

Input: Representation of the environment, start cells, and goal cells
Result: optimal collision-free solution

$R.constraints \leftarrow \emptyset$
$R.paths \leftarrow$ find independent paths for all agents using a_star()
$R.collisions \leftarrow$ detect_collisions($R.paths$)
$R.cost \leftarrow$ get_sum_of_cost($R.paths$)
insert R into OPEN
while $OPEN \neq \emptyset$ **do**
    $P \leftarrow$ node from OPEN with the smallest cost
    OPEN $\leftarrow$ OPEN $\setminus \{P\}$
    if $P.collisions = \emptyset$ **then**
        return $P.paths$ // P is a goal node
    $collision \leftarrow$ one collision in $P.collisions$
    $constraints \leftarrow$ resolve_collision($collision$)
    for $constraint$ in $constraints$ **do**
        $Q \leftarrow$ new node
        $Q.constraints \leftarrow P.constraints \cup \{constraint\}$
        $Q.paths \leftarrow P.paths$
        $a_i \leftarrow$ the agent in $constraint$
        $path \leftarrow$ a_star(a_i, $Q.constraints$)
        if $path \neq \emptyset$ **then**
            Replace the path of agent a_i in $Q.paths$ by $path$
            $Q.collisions \leftarrow$ detect_collisions($Q.paths$)
            $Q.cost \leftarrow$ get_sum_of_cost($Q.paths$)
            Insert Q into OPEN
return'No solutions'

Algorithm 1 shows the pseudo-code of the high-level search of CBS. Add code to `cbs.py` that finalizes the high-level search of CBS in the function `find_solution`, where we have already provided the implementation of lines 1 to 5. To manage the OPEN list, you can use the helper functions `push_node` and `pop_node`. Add print statements that list the expanded nodes (for debugging), and test your code as before.

3.4 Testing Your Implementation

You can test your implementation by running it on our test instances:

```
python run_experiments.py --instance "instances/test_*" --solver CBS --batch
```

(This may take a while depending on your computer.) The batch command creates an output file `results.csv`, which you can compare to the one provided in `instances/min-sum-of-cost.csv`.

4 Task 4: Implementing Priority-Based Search (PBS)

Priority-Based Search (PBS) generalizes the notion of prioritized planning with a fixed total priority ordering on the agents to planning with all possible total priority orderings. It performs a depth-first search on the high level to dynamically construct a priority ordering and thus builds a priority tree (PT). PBS computes near-optimal solutions and is much more efficient than CBS.

4.1 Generating Priority Pairs

When a collision occurs between two agents, PBS explores both scenarios where one is prioritized over another and vice versa. In particular, when PBS expands a PT node N to resolve a collision, PBS generates two child PT nodes N_1 nd N_2 that correspond to the ordered priority pairs $j \prec i$ and $i \prec j$. Add code to `pbs.py` that implements the function `generate_priority_pairs`.

4.2 Implementing the High-level Search

Algorithm 2 shows the pseudo code of the high-level search of PBS. Add code to `pbs.py` that finalizes the high-level search of PBS in the function `find_solution` and the function `update_plan`. You may want to make use of the code from Task 2.3 and the following utility functions:

- `get_lower_priority_agents`/`get_higher_priority_agents` give you the agents with lower/higher priority compared to a given agent in a topological ordering.
- `collide_with_higher_priority_agents` determines whether a given agent's path collides with the paths of other agents that have a higher priority in a topological ordering.

4.3 Testing Your Implementation

You can test your implementation by running it on our test instances:

```
python run_experiments.py --instance "instances/test_*" --solver PBS --batch
```

(This may take a while depending on your computer.) The batch command creates an output file `results.csv`, which you can compare to the one provided in `instances/min-sum-of-cost.csv`.

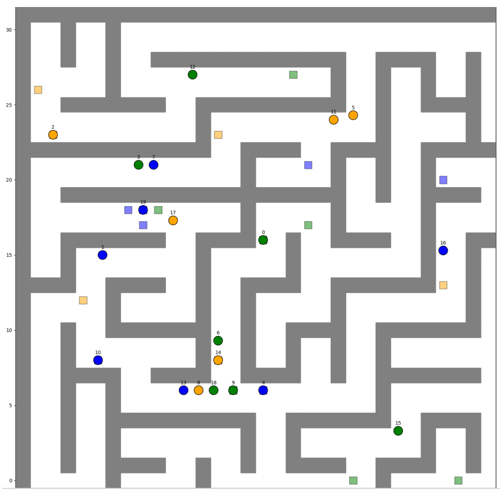
Algorithm 2: High-level search of PBS.

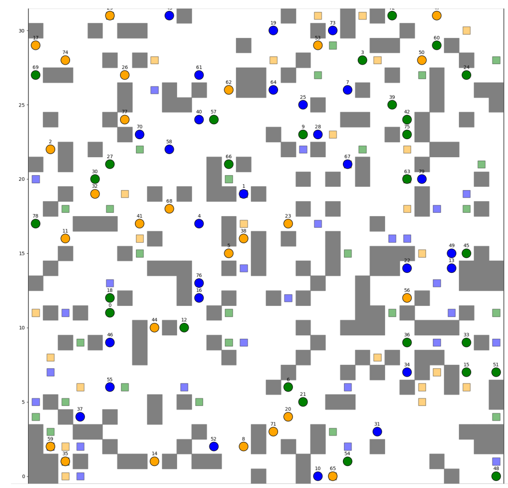
```
Input: Representation of the environment, start cells, and goal cells, ≺_0 (= ∅ by default)
R.priority_pairs ← ≺_0
R.paths ← find independent paths for all agents using a_star()
R.collisions ← detect_collisions(R.paths)
R.cost ← get_sum_of_cost(R.paths)
STACK ← {R}
while STACK ≠ ∅ do
    P ← top node in STACK
    STACK ← STACK \{P}
    if P.collisions = ∅ then
        return P.paths
    collision ← first vertex or edge collision < a_i, a_j, ··· > in P.collisions
    for a_i involved in collision do
        Q ← new node
        Q.paths ← P.paths
        Q.priority_pairs ← P.priority_pairs.append((j, i))
        success ← update_plan(Q,a_i)
        if success then
            Q.cost ← get_sum_of_cost(Q.paths)
            Q.collisions ← detect_collisions(Q.paths)
    Insert new nodes Q into STACK in non-increasing order of Q.cost.
return'No solutions'
Function update_plan(N, a_i):
    LIST ← topological sorting on partially ordered set ({i} ∪ {j|i ≺_N j}) // i and all js
      that are having lower priorities than i specified by N.priority_pairs
    for j ∈ LIST do
        if j = i or a_j collides with a_k in N.paths, where k ≺_N j then
            Update a_j's path in N.paths by invoking a low-level search for a_j, where the paths
              of a_k (k ≺_N j) are converted into constraints for a_j
            if no path returned by the low-level search then
                return false
    return true
```

5 Task 5: Challenge Your MAPF Solvers

Time for challenges! Run your Prioritized Planning (Task 2), CBS (Task 3), and PBS (Task 4) solvers on the following MAPF instances (`maze_map.txt` and `random_map.txt`), compare their performance.

Answer the following question in your report: Document your observation on their CPU time (seconds), the sum of costs, nodes generated and expanded. The template code includes a utility function `print_results` that gives you the above stats for a given solver. You may want to set a time limit for the solvers to run, as the two cases might be too hard for some solvers to find a solution in a reasonable amount of time. In that case, document the time limit set for your solvers, the sum of costs, and nodes generated/expanded under the time limit.





Appendices

A Introduction

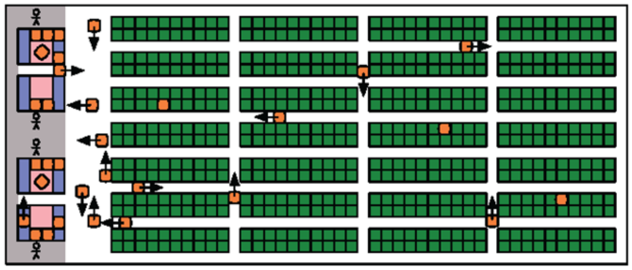


Figure 1: A small Amazon order-fulfillment center .

Multi-agent path finding (MAPF) is important for many applications, including automated warehousing. For example, Amazon order-fulfillment centers (Figure 1) have inventory stations around the perimeter of the warehouse (shown only on the left side in the figure) and storage locations in its center. Each storage location can store one inventory pod. Each inventory pod holds one or more kinds of goods. A large number of warehouse robots operate autonomously in the warehouse. Each warehouse robot is able to pick up, carry, and put down one inventory pod at a time. The warehouse robots move inventory pods from their storage locations to the inventory stations where the needed goods are removed from the inventory pods (to be boxed and eventually shipped to customers) and then back to the same or different empty storage locations to return the inventory pods. Amazon puts stickers onto the floors of their order-fulfillment centers to delineate a grid and allow for robust robot navigation. However, path planning for the robots is tricky since most warehouse space is used for storage locations, resulting in narrow corridors where robots that carry inventory pods cannot pass each other. Just-in-time manufacturing is an extension of automated warehousing for which no commercial installations exist so far. For just-in-time manufacturing, there are manufacturing machines around the perimeter of the warehouse rather than inventory stations. Robots go back and forth between the warehouse and the manufacturing machines, transporting raw material in one direction and manufactured products in the other direction. Just-in-time manufacturing increases the importance of delivering all needed raw material almost simultaneously to the manufacturing machines.

The MAPF problem is a simplified version of these and many other multi-robot or multi-agent path-planning problems and can be described as follows: On math paper, some cells are blocked. The blocked cells and the current cells of n agents are known. A different unblocked cell is assigned to each agent as its goal cell. The problem is to move the agents from their current cells to their respective goal cells in discrete time steps and let them wait there. The optimization objective is to minimize the sum of the travel times of the agents until they reach their goal cells (and can stay there forever). At each time step, each agent can *wait* at its current cell or *move* from its current cell to an unblocked neighboring cell in one of the four main compass directions. A *path* for an agent is a sequence of move and wait actions that lead the agent from its start cell to its goal cell or, equivalently, the sequence of its cells at each time step (starting with time step 0) when it executes these actions. The length of the path is the travel time of the agent until it reaches its goal cell (and stays there forever afterward). A *solution* is a set of n paths, one for each agent. Its cost is the sum of the lengths of all paths. Agents are not allowed to collide with the environment or each other. Two agents collide if and only if they are both in the same cell at the same time step (called a *vertex collision* or, synonymously a vertex conflict) or both move to the current cell of the other agent at the same time step (called an *edge collision* or, synonymously, an edge conflict). (An agent is allowed to move from its current cell x to the current cell y of another agent at the same time step when the other agent moves from cell y to a cell different from cells x and y.) Finding optimal collision-free solutions is NP-hard.

B MAPF Example

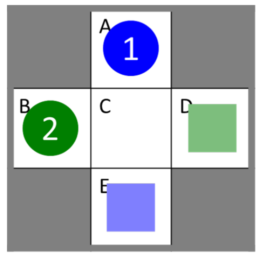


Figure 2: Our example MAPF instance. Circles represent start cells. Squares represent goal cells.

Figure 2 shows an example MAPF instance with two agents, where agent 1 has to navigate from its current cell A to its goal cell E and agent 2 has to navigate from its current cell B to its goal cell D. One optimal collision-free solution (of cost 5) consists of path [A, C, E] (of length 2) for agent 1 and path [B, B, C, D] (of length 3) for agent 2. The other optimal collision-free solution (also of cost 5) consists of path [A, A, C, E] (of length 3) for agent 1 and path [B, C, D] (of length 2) for agent 2.

C Planning in Joint Location Space

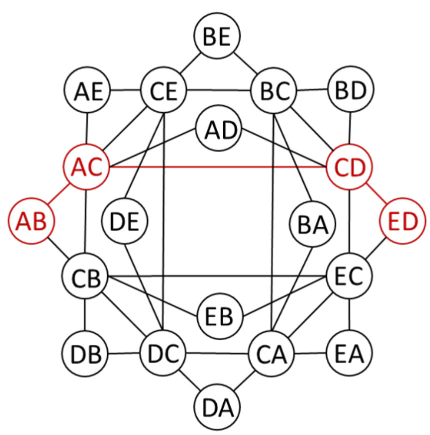


Figure 3: The joint location space for our example MAPF instance, which contains 20 vertices and 36 edges. The two letters in each circular vertex represent the cells of agents 1 and 2. The red circles and lines represent an optimal solution, namely path [A, A, C, E] (of length 3) for agent 1 and path [B, C, D] (of length 2) for agent 2.

In principle, one can find an optimal collision-free solution for a MAPF instance by planning for all agents simultaneously in joint location space by finding a shortest path on a graph whose vertices correspond to tuples of cells, namely one for each agent. Figure 3 shows this graph for our example MAPF instance. However, the number of vertices of the graph grows exponentially with the number of agents, which makes this search algorithm too slow in practice. Therefore, one needs to develop search algorithms that exploit the problem structure better to gain efficiency. We now discuss two such search algorithms, namely prioritized planning and conflict-based search.

D Prioritized Planning

Prioritized planning orders the agents completely by assigning each agent a different priority. It then plans paths for the agents, one after the other, in order of decreasing priority. It finds a path for each agent that does not collide with the environment or the (already planned) paths of all higher-priority agents (which can be done fast). Prioritized planning is fast but suboptimal (meaning that it does not always find an optimal collision-free solution) and even incomplete (meaning that it does not always find a collision-free solution even if one exists). If it finds a solution, then the solution is collision-free but the cost of the solution depends heavily on the priorities of the agents. More information on prioritized planning can be found in.

Consider our example MAPF instance, and assume that agent 1 is assigned a higher priority than agent 2. Then, prioritized planning first finds the shortest path [A, C, E] (of length 2) for agent 1 and afterward the shortest path [B, B, C, D] (of length 3) for agent 2 that does not collide with the path of agent 1 (resulting in a collision-free solution of cost 5).

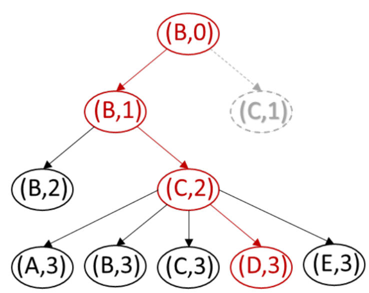


Figure 4: The search tree of space-time A* for agent 2 for our example MAPF instance if agent 1 has higher priority than agent 2 and the path of agent 1 is [A, C, E]. The pair in each oval node represents a cell and time step. The f-value of a node is the sum of its g-value (equal to the time step) and its h-value (equal to the distance of its cell to the goal cell D in the environment). Node (C,1) is pruned because agent 1 occupies cell C at time step 1 and agent 2 has to prevent a vertex collision with agent 1. The red nodes and edges represent an optimal solution for agent 2, namely path [B, B, C, D].

Prioritized planning uses *space-time A** to plan paths for each agent. Space-time A* is a modified A* algorithm that searches in the cell-time space, where the vertices are pairs (x, t) of cells x and time steps t. Vertex (x, t) has a directed edge to vertex $(x, t+1)$ if and only if the agent can wait at cell x at time step t. Vertex (x, t) has a directed edge to vertex $(y, t+1)$ if and only if the agent can move from cell x to cell $y \neq x$ from time step t to time step $t+1$. Figure 4 shows the search tree of space-time A* for agent 2 for our example MAPF instance.

E Conflict-Based Search

Conflict-Based Search (CBS) first plans shortest paths for all agents independently (which can be done fast). These paths do not collide with the environment but are allowed to collide with the paths of the other agents. If this results in a collision-free solution, then it has found an optimal collision-free solution. Otherwise, it chooses a collision between two agents (for example, agents a and b are both in cell x at time step t) and considers recursively two cases, namely one with the constraint that prohibits agent a from being in cell x at time step t and one with the constraint that prohibits agent b from being in cell x at time step t. The hope is that CBS finds a collision-free solution before it has imposed all possible constraints. CBS is slower than prioritized planning but complete and optimal. CBS is a two-level search algorithm. We now describe its operation in detail.

The high level of CBS searches the binary *constraint tree*. Each node N of the constraint tree contains: **(1)** a set of constraints imposed on the agents, where a constraint imposed on agent a is either a *vertex constraint* $\langle a, x, t \rangle$, meaning that agent a is prohibited from being in cell x at time step t, or an *edge constraint* $\langle a, x, y, t \rangle$, meaning that agent a is prohibited from moving from cell x to cell y at time step t; **(2)** a solution that satisfies all constraints but is not necessarily collision-free; and **(3)** the cost of the solution. The root node of the constraint tree contains an empty set of constraints and a solution that consists of n shortest paths. The high level performs a best-first search on the constraint tree, always choosing a fringe node of the constraint tree with the smallest cost to expand next. It breaks ties in favor of a node whose paths have fewer collisions with each other.

Once CBS has chosen node N for expansion, it checks whether the solution of node N is collision-free. If so, then node N is a goal node and CBS returns its solution. Otherwise, CBS chooses one of the collisions and resolves it by *splitting* node N. Assume that CBS chooses to resolve a vertex collision where agents a and b are both in cell x at time step t. In any collision-free solution, at most one of the agents can be in cell x at time step t. Therefore, at least one of the constraints $\langle a, x, t \rangle$ (that prohibits agent a from being in cell x at time step t) or $\langle b, x, t \rangle$ (that prohibits agent b from being in cell x at time step t) must be satisfied. Consequently, CBS splits node N by generating two child nodes of node N, each with a set of constraints that adds one of these two constraints to the constraint set of node N. Now assume that CBS chooses to resolve an edge collision where agent a moves from cell x to cell y and agent b moves from cell y to cell x at time step t. Then, the two additional constraints are the two edge constraints $\langle a, x, y, t \rangle$ (that prohibits agent a from moving from cell x to cell y at time step t) and $\langle b, y, x, t \rangle$ (that prohibits agent b from moving from cell y to cell x at time step t).

For each child node, the low level of CBS finds a new shortest path for the agent with the newly imposed constraint, which can be done fast with space-time A*. A vertex constraint, for example, just prunes a particular node in the search tree. This path does not collide with the environment and has to obey all constraints imposed on the agent in the child node but is allowed to collide with the paths of the other agents.

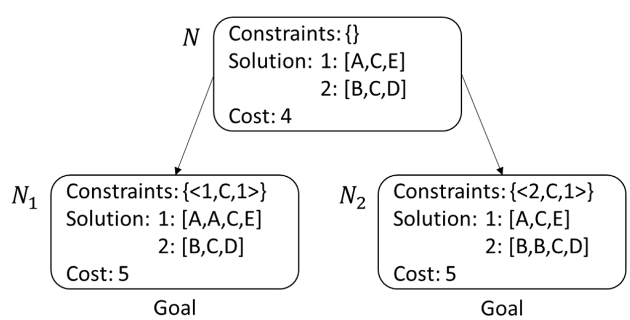


Figure 5: Constraint tree for our example MAPF instance.

Consider our example MAPF instance. Figure 5 shows the corresponding constraint tree. Its root node N contains the empty set of constraints, and the low level of CBS finds the shortest path [A, C, E] (of length 2) for agent 1 and the shortest path [B, C, D] (of length 2) for agent 2. Thus, the cost of node N is $2 + 2 = 4$. The solution of node N has a vertex collision where agents 1 and 2 are both in cell C at time step 1. Consequently, CBS splits node N. The new left child node N_1 of node

N adds the constraint $\langle 1, C, 1\rangle$. The low level of CBS finds the new shortest path [A, A, C, E] of length 3 (that includes a wait action) for agent 1 in node N_1, while the shortest path of agent 2 is identical to the one of node N since no new constraints have been imposed on agent 2. Thus, the cost of node N_1 is $3 + 2 = 5$. Similarly, the new right child node N_2 of node N adds the constraint $\langle 2, C, 1\rangle$. The low level of CBS finds the new shortest path [B, B, C, D] of length 3 (that includes a wait action) for agent 2 in node N_2, while the shortest path of agent 1 is identical to the one of node N since no new constraints have been imposed on agent 1. Thus, the cost of node N_2 is $2 + 3 = 5$. The best-first search on the high level of CBS now chooses a fringe node of the constraint tree with the smallest cost to expand next. Assume that it breaks the tie between nodes N_1 and N_2 in favor of node N_1. Since the solution of node N_1 is collision-free, it is a goal node and CBS returns its collision-free solution (of cost 5) that consists of path [A, A, C, E] (of length 3) for agent 1 and path [B, C, D] (of length 2) for agent 2.

F Priority-Based Search

As you may have observed prioritized MAPF solvers are very fast but their performance heavily depends on the priority order set between the agents. Setting the priority order is a non trivial problem and a bad priority ordering can result in solutions of bad quality or even fail to find solutions for solvable MAPF instances.

Priority-Based Search (PBS) generalises notion of prioritized planning with a fixed total priority ordering on the agents to planning with all possible total priority orderings. PBS explores the space of all total priority orderings lazily using a systematic depth-first search. PBS also computes near-optimal solutions and is much more efficient than CBS.

PBS is a two-level algorithm for prioritized planning. It performs a depth-first search on the high level to dynamically construct a priority ordering and thus builds a priority tree (PT). Conceptually, when a collision occurs between two agents, PBS explores both options by generating two child nodes where one agent is prioritized over another and vice versa. While traversing such a tree, it backtracks and explores other branches if there is no solution in the current branch. It thus effectively incrementally constructs a single partial priority ordering until it finds no collisions.

So algorithmically how does PBS handle collisions? Similar to CBS, PBS builds a tree but instead of a constraint tree it builds a priority tree. PBS introduces a new ordered pair to the priority ordering of the child PT node whenever it splits a parent PT node. Therefore, the number of splits in any branch of the PT is $O(M^2)$ (the number of all possible ordered pairs)

High-level search On the high level, PBS starts with a root PT node that contains an empty priority ordering (or a given one). Then using the collision handling policy above it will search through the space of priority orderings until it finds a priority ordering which produces no collisions

Low-level search On the low level, PBS uses a special low-level search to find an individually optimal path for each agent that does not collide with the path of any agents with higher priorities. This low-level search is similar to just running a prioritized space-time A* search with the priority order given by the high-level search.

G Additional Information

Additional information on the MAPF problem and solution approaches can be found at http://mapf.info, a website that contains tutorials, publications, data sets, and additional software for MAPF.