CSCI 360 – Project #3

Programming Part: Multi-Agent Path Finding

In this project, you will learn about Multi-Agent Path Finding (MAPF) and implement parts of three solvers, namely time-space A*, prioritized planning, and Conflict-Based Search (CBS).

Review the material from Sections 3.5 and 3.6 of the textbook, the slides on Heuristic Search and MAPF, and the overview of MAPF at the end of this document.

Task 0: Preparing for the Project

Installing Python 3

We provide python code for evaluation and visualization. It requires a Python 3 installation with the `numpy` and `matplotlib` packages. Using this visualization tool is not required to complete the project but highly recommended.

On Ubuntu Linux, download python by using

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

On Mac OS X, download Anaconda 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 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.

Understanding the Code

Folder task0 contains c++ code for an independent planning solver. Go to folder task0 and compile the c++ code by using

```
cmake .
make
```

Execute the c++ code by using

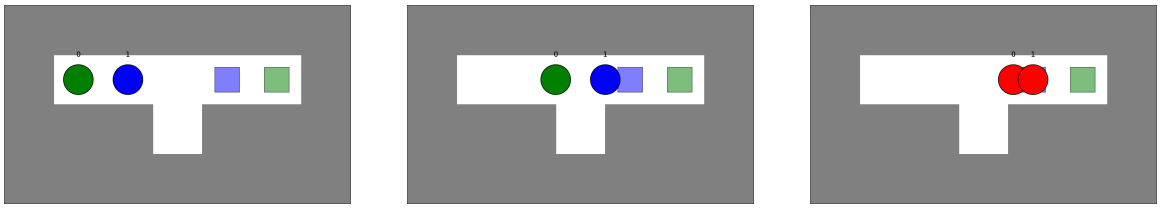
```
./task0 exp0.txt exp0_paths.txt
```

Here, file exp0.txt is the input file that contains the information of the map and the start and goal locations of the agents. File exp0_paths.txt is the output file that contains the paths.

Then, run the visualization tool in python by using

```
python3 ../visualize.py exp0.txt exp0_paths.txt
```

If you are successful, you should see an animation:



The independent planning 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 location 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 class `MAPFInstance` and class `AStarPlanner`. Class `MAPFInstance` stores the map and the start and goal locations of the agents. Class `AStarPlanner` performs an A* search that finds the shortest path for an agent in x-y space.

Task 1: Time-Space A*

In this task, you will change class `AStarPlanner` to perform a time-space A* search that searches in x-y-t space and returns a shortest path that satisfies a given set of constraints.

The files that you have to work on are `AStarPlanner.h` and `AStarPlanner.cpp`. Here are some hints about what you have to change. The changes include but are not limited to:

- struct `AStarNode`: Add a new member variable `timestep`.
- function `find_path`:
  - The function is defined in `AStarPlanner.h` as

```
Path find_path(int agent_id,
               const list<Constraint>& constraints = {});
```

It has a parameter `constraints`, that represents a list of constraints that the output path has to obey. The type `Constraint` is defined in `AStarPlanner.h`:

```
typedef tuple<int, int, int, int, ConstraintType>
  Constraint;
```

- Please **carefully read how the vertex and edge constraints are defined**. If the constraint is a vertex constraint, then it is in the format $\langle a_i, x, -1, t, VERTEX \rangle$, which prohibits agent a_i from being at location x at timestep t. If the constraint is an edge constraint, then it is in the format $\langle a_i, x, y, t, EDGE \rangle$, which prohibits agent a_i from moving from location x to location y from timestep $t-1$ to timestep t. You can distinguish a vertex constraint from an edge constraint by checking the ConstraintType.
- Change the key of the unordered_map `all_nodes` from the location of the state (whose type is `int`) to a location-timestep pair (whose type is `pair<int, int>`) to consider both locations and timesteps when checking for duplicates. The data structure of the new unordered_map `all_nodes` is already provided as a comment in `AStarPlanner.cpp`. Of course, you also have to change the duplicate detection procedure accordingly.
- When generating child nodes, add one more child node where, instead of moving to a neighboring location, the agent takes a wait action.
- Calculate the timesteps of child nodes. The timestep of a child node should be the timestep of its parent node plus one.
- After generating a new node, check whether the new node satisfies the constraints in `constraints` and prune it if it does not.
- Pay attention to how agents should behave with different constraints (for example, whether the constrained timestep is before or after an agent reaches the goal location and whether the constrained location is the start or goal location).

You are not required to follow the hints strictly. Go to folder task1, compile, and run `task1.cpp` on the MAPF instance in file `exp1.txt`. You can also edit `task1.cpp` to add more constraints to agents. Don't forget to use the python tool to help you visualize the results and debug your code.

Here are some scenarios that we suggest you to try to help you ensure that the time-space A* solver runs properly:

1. Run your code with a constraint that prohibits agent 0 from being at its goal location at timestep 4 and another constraint that prohibits agent 1 from moving from its start location to the neighboring location from timestep 0 to timestep 1. Where is agent 0 at timestep 4, and where is agent 1 at timestep 1 on your output paths? (The output paths could have collisions.)

2. Run your code with a constraint that prohibits agent 0 from being at its goal location at timestep 10. Where is agent 0 at timestep 10 on your output paths? (The output paths could have collisions.)

3. Design a set of constraints by hand that allows your algorithm to find collision-free paths with a minimal sum of path lengths. Run your code with the set of constraints. What is the sum of path lengths?

The command to compile and run the code is similar:

```
cmake .
make
./task1 exp1.txt exp1_paths.txt
python3 ../visualize.py exp1.txt exp1_paths.txt
```

Grading of Task 1

Your time-space A* solver has to include the following definitions:

```
class AStarPlanner {
public:
    AStarPlanner(const MAPFInstance& ins);
    Path find_path(int agent_id, const list<Constraint>& constraints = {});
    ...
}
```

You are free to change the source code as long as the testing program can invoke `AStarPlanner` through the definitions above. You are allowed to do things like:

1. Add more class members to the `AStarPlanner` class.
2. Add additional optional parameters to the `find_path` function as long as they are after the `constraints` parameter.

Your `find_path` function should return **a shortest path** for agent `agent_id` that satisfies all input constraints in `constraints` if such a path exists or return **an empty vector** otherwise.

We will evaluate your time-space A* solver with **10 test cases** to test its correctness. A **runtime limit of 5 seconds** (which should be sufficient as our time-space A* solver is able to finish each MAPF instance within 1 seconds) is set for each MAPF instance. Your time-space A* solver needs to output a correct solution within the time limit.

Task 2: Prioritized Planning

In this task, you will implement a prioritized planning solver based on your time-space A* solver. The prioritized planning solver finds paths for all agents, one after the other, that do not collide with the environment or the already planned paths of other agents. To ensure that the path of an agent does not collide with the already planned paths of other agents, you have to transform the already planned paths into constraints and use them as the input of your time-space A* solver. Don't forget to use the python tool to help you visualize the results and debug your code.

1. Add vertex and edge constraints. Go to folder task2 and add code to `task2.cpp` that transforms the already planned paths into constraints. Give agent 0 the highest priority. Then, test your prioritized planning solver on MAPF instance `exp2_1.txt`.

2. Add additional constraints. After Step 1, 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 timestep when agents reach their goal locations but also to all future time steps.

3. Test your prioritized planning solver on MAPF instance `exp2_3.txt` where agent 0 has the highest priority. Did your prioritized planning solver terminate properly and report "no solutions"?

The command to compile and run the code is similar:

```
cmake .
make
./task2 exp2_1.txt exp2_1_paths.txt
python3 ../visualize.py exp2_1.txt exp2_1_paths.txt
```

Grading of Task 2

Your prioritized planning solver has to include the following definitions:

```
class PrioritizedSearch {
public:
  vector<Path> find_solution();
  PrioritizedSearch(const MAPFInstance& ins): a_star(ins) {}
    ...
}
```

You are free to change the code as long as the testing code can use your code through the above class and function definitions.

Your function `find_solution()` has to return a vector of conflict-free paths for all agents (or **all paths that your prioritized planning solver has found so far** when it cannot find a path for an agent any longer). Your prioritized planning solver has to plan for agents in the same order as they are listed in `ins`.

We will evaluate your prioritized planning solver with **20 test cases** to test its correctness. Your prioritized planning solver should output the paths within a runtime limit of 10 seconds (which should be sufficient as our prioritized planning solver is able to finish each MAPF instance within 2 seconds) for each MAPF instance.

Task 3: Conflict-Based Search (CBS)

In this task, you will implement Conflict-Based Search (CBS), a popular optimal solver. It is slower than prioritized planning but complete and optimal. You will implement the high-level search of CBS and reuse the previous implementation in `SingleAgentPlanner` as the low-level search of CBS. Study the code in `CBS.h` and `CBS.cpp` in folder `task3` carefully. Write code in function `find_solution` that performs the high-level search of CBS. The pseudo code of CBS is provided in the appendix. We have already provided the implementation for generating the root node (Lines 1-4), and you have to fill in the code in the while loop (Lines 6-20). Don't forget to use the python tool to help you visualize the results and debug your code. To test your implementation, we provide 50 MAPF instances in folder test and a file sum-of-costs.xlsx that contains the optimal sum of path lengths for each MAPF instance. We suggest that you output the sum of path lengths of your CBS solver and compare them with the given results to check whether your CBS solver runs properly.

The command to compile and run the code is similar:

```
cmake .
make
./task3 ../test/test_1.txt paths.txt
python3 ../visualize.py ../test/test_1.txt paths.txt
```

Grading of Task 3

Your CBS solver has to include the following definitions:

```
class CBS {
public:
  vector<Path> find_solution();
  explicit CBS(const MAPFInstance& ins): a_star(ins) {}
    ...
}
```

You are free to change the code as long as the testing code can use your code through the above class and function definitions.

Your function `find_solution()` has to return a vector of conflict-free paths for all agents or an **empty vector** if it cannot find a solution.

We will evaluate your CBS solver with **20 test cases** to test its correctness and optimality. Your CBS solver should output an optimal solution within a runtime limit of 10 seconds (which should be sufficient as our CBS solver is able to finish each MAPF instance within 2 seconds) for each MAPF instance.

Submission and grading

You are not allowed to use any external libraries (except for STL) for your code. We have tested and verified that the provided code (and a solution to the project that

extends the provided code) compiles and runs on Ubuntu 16.04, Windows 10 Pro, and Mac OS 11. You are free to develop your code on any platform that you choose, although we have not tested if our code works on other operating systems (and we might not be able to help you to get your code working on your preferred platform). We will test your code on a Linux machine that supports **c++11**.

You have to submit these **6** files:

1. AStarPlanner.cpp and .h
2. PrioritizedSearch.cpp and .h
3. CBS.cpp and .h

Our testing code will (only) use the class functions (including constructors) and class members that are used in task{i=0 . . . 3}.cpp. In other words, when you change function or class definitions, please make sure that the new definitions are still compatible with task{i=0 . . . 3}.cpp. **Please read this document carefully and make sure that you understand what the expected output for each function is.**

Theoretical Part: Multi-Agent Path Findingt

In the first two questions, you will work on examples that demonstrate properties of prioritized planning, namely that it is incomplete and suboptimal. In the last question, you will work on an example that demonstrates properties of CBS. Here, a collision-free solution is a set of collision-free paths, and an optimal collision-free solution is a set of collision-free paths that minimize the sum of path lengths. The ordering of the agents specifies their priorities, from highest to lowest.

Solve the following tasks either on paper or with your implementations.

1. 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. Explain why your MAPF instance meets the requirement.
2. Does CBS always terminate in finite time when the input MAPF instance is unsolvable? If not, provide an example MAPF instance. If so, provide a proof.
3. [Bonus question[1]] 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. Explain why your MAPF instance meets the requirement.

[1]The bonus is worth 10% of the full points of this project. If the total points you receive exceed the full points, the extra points won't be counted into your grade.

Submission

You have to submit your solutions through the blackboard system by Thursday, November 18th, 11:59 pm (which is the time when the grace period starts). For the programming part, submit your **six** source code and header files. For the theoretical part, submit a PDF file named "part2.pdf". Upload the files individually and not as an archive. If you have any questions about the project, you can post them on Piazza or ask a TA directly during office hours. If you have a bug in your code that you have been unable to find, you can ask a CP for help.

Appendix: Multi-Agent Path Finding

Introduction

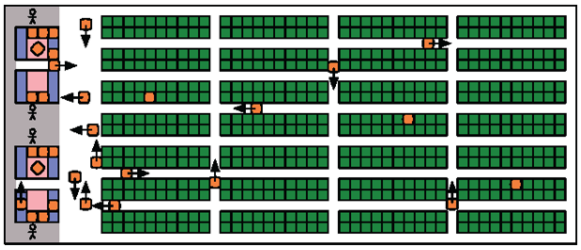


Figure 1: A small Amazon order-fulfillment center [5].

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 [5]. 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 timestep 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 [6].

MAPF Example

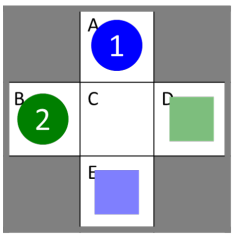


Figure 2: An example MAPF instance. Circles represent start cells. Squares represent goal cells.

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.

Planning in Joint Location Space

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 has 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.

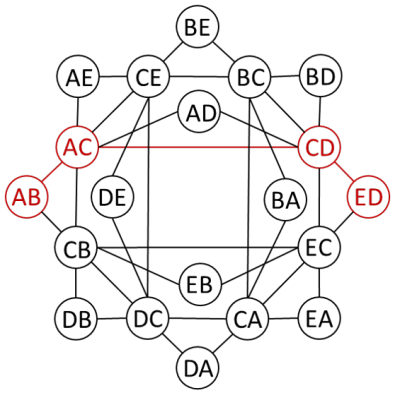


Figure 3: The joint location space for the example in Figure 2, which contains 20 vertices and 36 edges. The two letters in each circle represent the cells of agents 1 and 2. The red circles and lines represent an optimal solution, namely path [A, A, C, E] (of length 3) for agent 1 and path [B, C, D]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

Prioritized Planning

Prioritized planning [1] 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 [2].

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, E] (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).

Conflict-Based Search

Conflict-Based Search (CBS) [3, 4] 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.

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). Algorithm 1 shows the pseudo code of the high-level search.

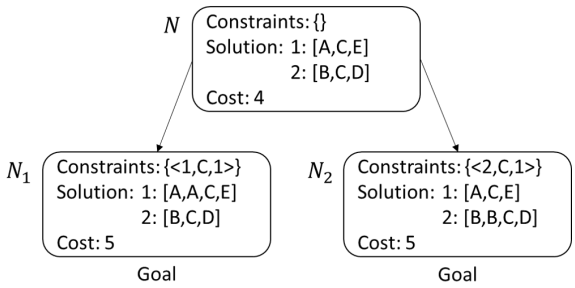


Figure 4: Constraint tree for the example MAPF instance.

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 a version of A*, the details of which are given in [4]). 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.

Consider our example MAPF instance. Figure 4 shows the corresponding constraint tree. Its root node N contains the empty set of constraints, and the low level

Algorithm 1: High-level search of CBS.

```
Data: Representation of the environment, start cells, and goal cells
Result: optimal collision-free solution
R.constraints ← ∅
R.paths ← find independent paths for all agents using a_star()
R.cost ← get_sum_of_cost(R.paths)
insert R into OPEN
while OPEN is not empty do
    P ← node from OPEN with the smallest cost
    collision ← find_collision(P.paths)
    if collision does not exists then
        return P.paths // P is a goal node
    constraints ← extract from collision
    for constraint in constraints do
        Q ← new node
        Q.constraints ← P.constraints ∪ {constraint}
        Q.paths ← P.paths
        a_i ← the agent in constraint
        path ← a_star(a_i, Q.constraints)
        if path is not empty then
            Replace the path of agent a_i in Q.paths by path
            Q.cost ← get_sum_of_cost(Q.paths)
            Insert Q into OPEN
return "No solutions"
```

of CBS finds the shortest path [A, C, E] (of length 2) for agent 1 and the shortest path $\langle B, C, D\rangle$ (of length 2) for agent 2. Thus, the cost of node N is $2 + 2 = 4$. The solution of node N has a 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.

Additional Information

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

References

[1] M. Erdmann and T. Lozano-P'erez. On multiple moving objects. *Algorithmica*, 2:477–521, 1987.

[2] H. Ma, D. Harabor, P. Stuckey, J. Li, and S. Koenig. Searching with consistent prioritization for multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 7643–7650, 2019.

[3] G. Sharon, R. Stern, A. Felner, and N. Sturtevant. Conflict-based search for optimal multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 563–569, 2012.

[4] G. Sharon, R. Stern, A. Felner, and N. Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40–66, 2015.

[5] P. Wurman, R. D'Andrea, and M. Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Magazine*, 29(1):9–20, 2008.

[6] J. Yu and S. LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1444–1449, 2013.