Individual Programming Project (25pt+2 × 0.5pt bonus)*

CMPT 417/827, Spring 2020

A Project on Multi-Agent Path Finding (MAPF)

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

Submission Requirements

Your submission should be a zip file which contains the following:

1. Your modified copy of the provided codebase;
2. A pdf file named as `<YOUR-SFU-STUDENT-NUMBER>.pdf` where-in you describe your implementation decisions, and answer the questions asked of you within the assignment.

It is unnecessary to include the instances folder within your submission, as we will test your implementations against our original copy of the instances.

Any test instances which you design on your own should be included in a folder titled "`custominstances/`" and explained within your report.

Coding Style

Points may be deducted if your implementations are written with an incomprehensible coding style.

Academic Honesty

Please remember that you are expected to complete this project individually, and to refrain from sharing your solutions with other students. Plagiarism, either as a perpetrator or a facilitator, will be dealt with in accordance with the Simon Fraser University academic honesty policy. Maintaining high standards of academic integrity is essential to ensure the lasting value of a university degree.

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:

*This project is based on Model AI Assignments 2020: A Project on Multi-Agent Path Finding (MAPF) by Wolfgang Hönig, Jiaoyang Li, Sven Koenig at the University of Southern California.

```
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
```

It may be necessary to use "pythonw" instead depending on the particular version of Mac OS X.

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 Learning about MAPF

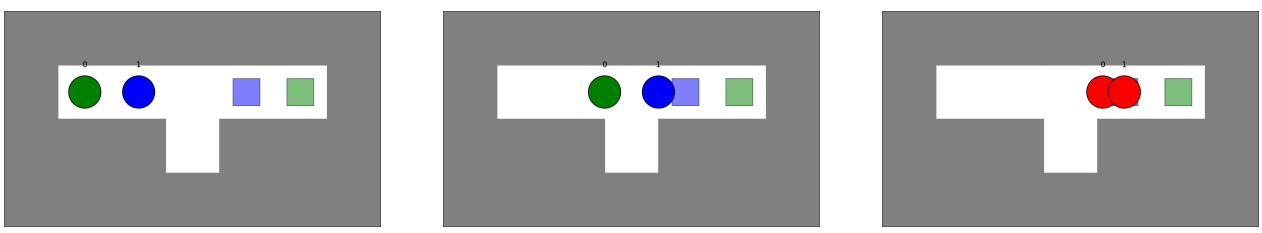
Read the provided textbook-style overview of MAPF.

0.4 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:

Computing heuristics for the low-level search

```
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))
```

list of lists specifying obstacle positions

list of goal location.

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:

representation of the environment

```
    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 Space-Time A* 6/20

You now change the single agent solver to perform a space-time A* search that 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.

~~1.1 Searching in the Space-Time Domain (1pt)~~

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 neighbouring cell.

You can test your code by using:

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

and should observe identical behaviour. Include the output in your report.

1.2 Handling Vertex Constraints (1pt)

We first consider (negative) 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 (negative) vertex constraint that prohibits agent 2 from occupying cell $(3, 4)$ at time step 5:

```
{'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). Include the output in your report.

1.3 Adding Edge Constraints (1pt)

We now consider (negative) 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 (negative) 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 neighbouring cell $(1, 3)$ from time step 0 to time step 1.

~~1.4 Handling Goal Constraints (1.5pt)~~

Run your code with a constraint that prohibits agent 0 from being at its goal cell $(1, 5)$ at time step 10. Where is agent 0 at time step 10 in your solution? To make the algorithm work properly, you might have to change the goal test condition. In your report, explain what changes you made to the goal test condition. (The solution of both agents could have collisions.)

1.5 Designing Constraints (1.5pt) ⇒ Report

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. Within your report, document this set of constraints, the solution, and the sum of path lengths.

2 Task 2: Implementing Prioritized Planning (6.5+0.5)/20

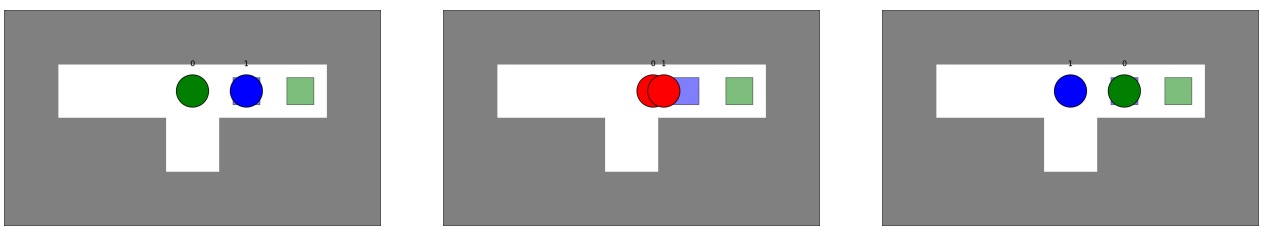
The independent MAPF solver finds paths for all agents, simultaneously or one after the other, that do not collide with the environment but are allowed to collide with the paths of the other agents. The prioritized MAPF solver finds paths for all agents, one after the other, that do not collide with 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 (negative) constraints compiled from their paths.

~~2.1 Adding Vertex Constraints (1pt)~~

Add code to `prioritized.py` that adds all necessary vertex constraints. You need two loops, namely one to iterate over the path of the current agent and one to add vertex constraints for all future agents (since constraints apply only to the specified agent). You can test your code by using:

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

Now, the blue agent 2 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 (negative) edge constraints as well.

2.2 Adding Edge Constraints (1pt)

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

2.3 Adding Additional Constraints (1.5pt)

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.

~~2.4 Addressing Failures (2pt)~~

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`. Did your solver terminate properly and report "no solutions"? If not, describe what happened in your report and change your code to address the issue. Hint: You can address this issue 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.

2.5 Showing that Prioritized Planning is Incomplete and Suboptimal (1pt+0.5pt)

Solve one or more of the following tasks either on paper or with the implementation of the prioritized MAPF solver after you have added the additional constraints from Section 2.3:

- 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.
- (Bonus: 0.5pt) 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.

Document these within your report.

3 Task 3: Implementing Conflict-Based Search (CBS) 6.5/20

Conflict-Based Search (CBS) is slower than prioritized planning but complete and optimal.

~~3.1 Detecting Collisions (1.5pt)~~

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 (1pt)~~

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 (negative) 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 `standard_splitting` 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 (3.5pt)~~

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, and include the transcript of one run within your report. If your transcript is too long, you may truncate it to a more reasonable length.

3.4 Testing your Implementation (0.5pt)

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

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

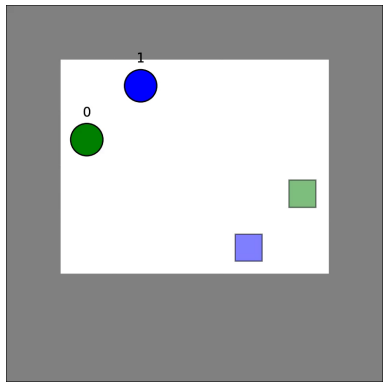
(This may take a while depending on your computer.) The batch command creates an output file `results.csv`, which you can compare to the one provided in `instances/min-sum-of-cost.csv`.

4 Task 4: Implementing CBS with Disjoint Splitting 6/20

CBS expands many nodes for some MAPF instances as the MAPF instance `exp4.txt` shows, for which CBS expands about 11 nodes (although the exact number can vary):

Algorithm 1: High-level search of CBS.

```
Input: 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.collisions ← detect_collisions(R.paths)
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
    if P.collisions = ∅ then
        return P.paths // P is a goal node
    collision ← one collision in P.collisions
    constraints ← standard_splitting(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.collisions ← detect_collisions(Q.paths)
            Q.cost ← get_sum_of_cost(Q.paths)
            Insert Q into OPEN
return'No solutions'
```



You can check this by using:

```
python run_experiments.py --instance instances/exp4.txt --solver CBS
```

If CBS chooses to resolve a vertex collision where agents a and b are both in cell x at time step t, then it generates two negative vertex constraints, namely the negative vertex constraint $\langle a, x, t\rangle$ (that prohibits agent a from being in cell x at time step t) and the negative vertex constraint $\langle b, x, t\rangle$ (that prohibits agent b from being in cell x at time step t). If 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 it generates two negative edge constraints, namely the negative edge constraint $\langle a, x, y, t\rangle$ (that prohibits agent a from moving from cell x to cell y at time step t) and the negative edge constraint $\langle b, y, x, t\rangle$ (that prohibits agent b from moving from cell y to cell x at time step t).

CBS with disjoint splitting changes the second constraint in both cases. If CBS with disjoint splitting chooses to resolve a vertex collision, it changes the second negative vertex constraint that prohibits agent b from being in cell x at time step t to a positive vertex constraint that requires agent a to be in cell x at time step t. On the other hand, if CBS with disjoint splitting chooses to resolve an edge collision, it changes the second negative edge constraint that prohibits agent b from moving from cell y to cell x at time step t to a positive edge constraint that requires agent a to move from cell x to cell y at time step t. In other words, CBS with disjoint splitting changes the second negative constraint (that prohibits the second agent from executing the colliding action) to a positive constraint for the first agent (that requires the first agent to execute the colliding action) in both cases. It could also change the first negative constraint to a positive constraint for the second agent and thus can choose one of the colliding agents freely, for example, randomly.

The reason for this change is that the second constraints are now stronger (meaning more constraining). For example, if an agent a is required to be in cell x at time step t, then all other agents (including agent b) are automatically prohibited from being in cell x at time step t since they would otherwise collide with agent a. Thus, the second positive vertex constraint of CBS with disjoint splitting implicitly includes the second negative vertex constraint of CBS. Thus, CBS with disjoint splitting can be expected to run faster than CBS but remains complete and optimal. More information on CBS with disjoint splitting can be found in [1].

4.1 Supporting Positive Constraints (1pt)

Add code to `single_agent_planner.py` to handle positive vertex and edge constraints (in addition to the current handling of negative vertex and edge constraints). For each constraint dictionary, you should add a new key `positive` with a binary value that indicates whether the constraint is positive.

~~4.2 Converting Collisions to Constraints (1pt)~~

Add code to `cbs.py` that implements the function `disjoint_splitting`. To create a positive constraint, set the item `positive` in the Python dictionary of the constraint to `True`. You can use `random.randint(0,1)` to choose one of the two colliding agents randomly.

~~4.3 Adjusting the High-Level Search (4pt)~~

Update the code in `cbs.py` to adjust the high-level search of CBS in the function `find_solution` to the new constraints. For each child node, the low level of CBS without disjoint splitting finds a new shortest path for the agent with the newly imposed negative constraint. The paths of the other agents do not need to be updated because they still satisfy the constraints of those agents. However, the low level of CBS with disjoint splitting might have to find new shortest paths not only for the agent with a newly imposed positive constraint but for other agents as well since a positive constraint for an agent implies negative constraints for all other agents. You can compute a list of agent ids of agents that violate a given positive constraint with the helper function `paths_violate_constraint`. CBS with disjoint splitting should not add a child node if no path exists for one or more of these agents. Not adding such child nodes is more important for CBS with disjoint splitting than for CBS because there are many more such nodes for CBS with disjoint splitting due to its stronger constraints.

Test your code as before. CBS with disjoint splitting should expand about 8 instead of 11 nodes (although the exact number can vary). Report how many nodes yours expanded.

If possible

5 Task 5: Benchmarking MAPF Solvers (Bonus) +0.5/20

Benchmark your three MAPF solvers on more exciting MAPF instances and compare their performance. You can find benchmark instances at http://mapf.info/index.php/Main/Benchmarks. Describe your findings within your report, preferably with a nice graph.

References

[1] J. Li, D. Harabor, P. Stuckey, A. Felner, H. Ma, and S. Koenig. Disjoint splitting for multi-agent path finding with conflict-based search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 279–283, 2019.