CMPT 417 – INDIVIDUAL LAB

Jayden Cole

MARCH 16, 2021

Contents

Task 1 – Implementing Space-Time A*

Part 1.1 – Adding Time Step to Space-Time A*

Part 1.1 was to add the time domain to make the base implementation of A* into Space-Time A*. Figure 1 is the output of the algorithm for "instances/exp0.txt".

```
***Import an instance***
Start locations
@ @ @ @ @ @ @
@ 0 1 . . . @
@ @ @ . @ @ @
@ @ @ @ @ @ @

Goal locations
@ @ @ @ @ @ @
@ . . . 1 0 @
@ @ @ . @ @ @
@ @ @ @ @ @ @

***Run Independent***

 Found a solution!

CPU time (s):    0.00
Sum of costs:    6
***Test paths on a simulation***
COLLISION! (agent-agent) (0, 1) at time 3.4
COLLISION! (agent-agent) (0, 1) at time 3.5
COLLISION! (agent-agent) (0, 1) at time 3.6
COLLISION! (agent-agent) (0, 1) at time 3.7
COLLISION! (agent-agent) (0, 1) at time 3.8
COLLISION! (agent-agent) (0, 1) at time 3.9
COLLISION! (agent-agent) (0, 1) at time 4.0
COLLISION! (agent-agent) (0, 1) at time 4.1
COLLISION! (agent-agent) (0, 1) at time 4.2
COLLISION! (agent-agent) (0, 1) at time 4.3
COLLISION! (agent-agent) (0, 1) at time 4.4
COLLISION! (agent-agent) (0, 1) at time 4.5
COLLISION! (agent-agent) (0, 1) at time 4.6
```

Figure 1: Output of Space-Time A for "instances/exp0.txt" using the Independent solver.*

Part 1.2 – Handling Vertex Constraints

Part 1.2 was to add negative vertex constraints into the program. A constraint table was built for each agent indicating the time step and locations that were not permitted. If a node was generated that fit a constraint definition, it was subsequently pruned. Below in Figure 2 is the output of the A* Space-

Time implementation running the Priority instance of the solver with constraint { ‘agent’ : 0, ‘loc’ : (1, 5), ‘timestep’: 4}.

```
***Import an instance***
Start locations
@ @ @ @ @ @ @
@ 0 1 . . . @
@ @ @ . @ @ @
@ @ @ @ @ @ @

Goal locations
@ @ @ @ @ @ @
@ . . . 1 0 @
@ @ @ . @ @ @
@ @ @ @ @ @ @

***Run Prioritized***

 Found a solution!

CPU time (s):    0.00
Sum of costs:    7
[[(1, 1), (1, 2), (1, 3), (1, 4), (1, 4),
(1, 5)], [(1, 2), (1, 3), (1, 4)]]
***Test paths on a simulation***
COLLISION! (agent-agent) (0, 1) at time 3.4
COLLISION! (agent-agent) (0, 1) at time 3.5
COLLISION! (agent-agent) (0, 1) at time 3.6
COLLISION! (agent-agent) (0, 1) at time 3.7
COLLISION! (agent-agent) (0, 1) at time 3.8
COLLISION! (agent-agent) (0, 1) at time 3.9
COLLISION! (agent-agent) (0, 1) at time 4.0
COLLISION! (agent-agent) (0, 1) at time 4.1
COLLISION! (agent-agent) (0, 1) at time 4.2
COLLISION! (agent-agent) (0, 1) at time 4.3
COLLISION! (agent-agent) (0, 1) at time 4.4
COLLISION! (agent-agent) (0, 1) at time 4.5
COLLISION! (agent-agent) (0, 1) at time 4.6
COLLISION! (agent-agent) (0, 1) at time 4.7
COLLISION! (agent-agent) (0, 1) at time 4.8
COLLISION! (agent-agent) (0, 1) at time 4.9
COLLISION! (agent-agent) (0, 1) at time 5.0
COLLISION! (agent-agent) (0, 1) at time 5.1
COLLISION! (agent-agent) (0, 1) at time 5.2
COLLISION! (agent-agent) (0, 1) at time 5.3
COLLISION! (agent-agent) (0, 1) at time 5.4
COLLISION! (agent-agent) (0, 1) at time 5.5
COLLISION! (agent-agent) (0, 1) at time 5.6
```

Figure 2: Output for Priority solver on instances/exp1.txt with vertex constraints

Part 1.3 – Handling Negative Edge Constraints

Building off the work done in part 1.2, negative edge constraint calculations were added. No sample output code is requested, however, a test constraining agent 0 from moving from (1, 1) to (1, 2) at timestep 1 had the agent remain in its starting square for one timestep before traversing that edge in timestep 2.

Part 1.4 – Handling Goal Constraints

A goal location node was not checked if there were any constraints for timesteps greater than when the agent reached the goal location node. Therefore, to ensure that a path was complete, a check at a potential goal node would test all future constraints to see if the agent was not permitted to be in the goal location node at some future time. Only if the agent was not constrained from the goal location in the future, would the node be returned as a goal node.

Part 1.5 – Designing Constraints

To find a collision free path with a minimum sum of paths for the Priority solver on experiment 1 (instances/exp1.txt) involved the following constraints:

- {'agent': 1, 'loc': [(1, 2)], 'timestep': 1},
- {'agent': 1, 'loc': [(1, 3)], 'timestep': 2},
- {'agent': 1, 'loc': [(1, 3), (1, 4)], 'timestep': 2},
- {'agent': 1, 'loc': [(1, 3), (1, 2)], 'timestep': 2}

Which found the solution:

[[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5)], [(1, 2), (1, 3), (2, 3), (1, 3), (1, 4)]]

With Sum of Cost: 8

Task 2 – Implementing Prioritize Planning

Part 2.1 – Adding Vertex Constraints

Vertex constraints were added to not allow agents to occupy the same vertex at the same timestep.

Part 2.2 – Adding Edge Constraints

Edge constraints were added so that two agents could not move through each other from one timestep to another.

Part 2.3 – Adding Additional Constraints

Additional constraints were needed for agents that had reached their goal nodes. Before this step, the goal nodes would only constrain the timestep that the agent moved into them. After implementation, goal nodes are infinitely occupied by the first agent to move into them.

Part 2.4 – Addressing Failures

Running the experiment 2-3 revealed errors in the program. The program continued running trying to find a solution to a problem that did not have one. As such, the process ran indefinitely until it ran out of memory. An upper bound on the length of an agent's path was required.

Part 2.5 – Prioritized Planning is Incomplete and Suboptimal

Figure 3 shows an example of where the ordering of the agents matters in finding a solution. In the displayed priority of the agents, the algorithm finds a solution, but reverse the ordering, and no solution can be found (blue agent shown in Figure 3 will block green agent after its path has been calculated). This example can be run using the command:

"python run_experiments.py --instance custominstances/exp2_5a.txt --solver Prioritized"

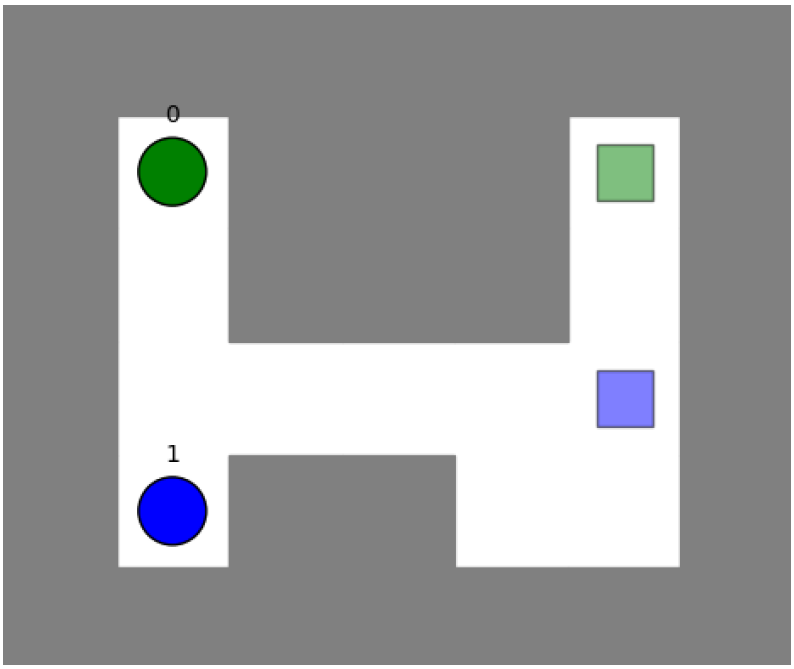

Figure 3: The green agent can reach its destination in this Prioritized Planning problem. However, reversing the priority of the agents results in no solution.

Figure 4 shows a simple instance of a MAPF where there is no solution no matter the priorities of the agents. There is simply no room for the agents to get out of each others way to reach their respective goal locations. The code for a similar example (but rather on a 5 by 7 grid) can be found in the custominstances folder and can be run using the command:

“python run_experiments.py --instance custominstances/exp2_5b.txt --solver Prioritized”

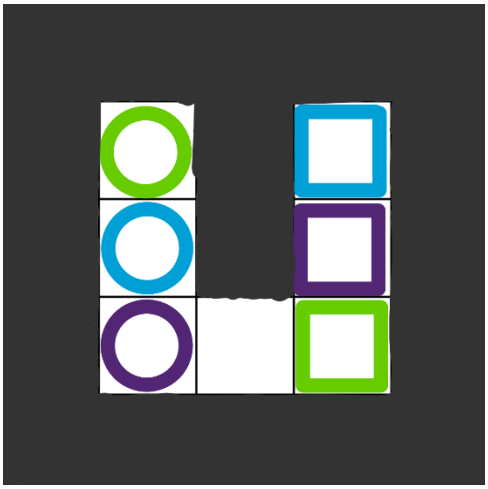

Figure 4: A MAPF instance with no solution no matter the priority of the agents.

Finally Figure 5 shows a MAPF instance again where the ordering of the agent’s priorities matter. The priority shown in the Figure permits for a solution to be found. The yellow agent will remain out of the way of the other two agents until they pass by. However, if the priorities are switched, for example, yellow, green, then blue, there are no solutions. The simulation below can be run using the command:

“python run_experiments.py --instance custominstances/exp2_5c.txt --solver Prioritized”

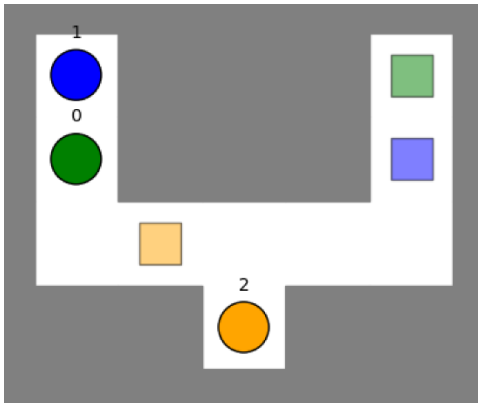

Figure 5: An instance of MAPF that has a solution for one set of priorities but not another. The set with a solution is pictured above.

Task 3 – Implementing Conflict-Based Search

Part 3.4 – Testing the CBS with Standard Splitting Implementation

Below is the output of the program when put to the test on the "instances/test_1.txt" MAPF instance using conflict-based search with standard splitting.

```
***Import an instance***

// Shortened for brevity

***Run CBS***

Generate node 0


Expand node 0

Expanded node cost: 40

Expanded node constraints: []

Expanded node collisions: [{'a1': 4, 'a2': 0, 'timestep': 0, 'loc':
[(1, 0), (1, 1)]}, {'a1': 4, 'a2': 1, 'timestep': 6, 'loc': [(3, 4)]},
{'a1': 4, 'a2': 2, 'timestep': 3, 'loc': [(3, 1)]}, {'a1': 3, 'a2': 1,
'timestep': 6, 'loc': [(3, 3), (3, 4)]}, {'a1': 3, 'a2': 2,
'timestep': 3, 'loc': [(2, 1), (3, 1)]}, {'a1': 2, 'a2': 1,
'timestep': 11, 'loc': [(1, 1)]}]

Generate node 1

Generate node 2


Expand node 2

Expanded node cost: 40

Expanded node constraints: [{'agent': 1, 'loc': [(1, 1)], 'timestep':
11}]

Expanded node collisions: [{'a1': 4, 'a2': 0, 'timestep': 0, 'loc':
[(1, 0), (1, 1)]}, {'a1': 4, 'a2': 1, 'timestep': 6, 'loc': [(3, 4)]},
{'a1': 4, 'a2': 2, 'timestep': 3, 'loc': [(3, 1)]}, {'a1': 3, 'a2': 1,
'timestep': 6, 'loc': [(3, 3), (3, 4)]}, {'a1': 3, 'a2': 2,
'timestep': 3, 'loc': [(2, 1), (3, 1)]}]

Generate node 3

Generate node 4
```

```
Expand node 3 …

// Shortened for brevity

Expand node 42

Expanded node cost: 41

Expanded node constraints: [{'agent': 1, 'loc': [(1, 1)], 'timestep':
11}, {'agent': 3, 'loc': [(2, 1), (3, 1)], 'timestep': 3}, {'agent':
1, 'loc': [(3, 4), (3, 3)], 'timestep': 6}, {'agent': 0, 'loc': [(1,
0)], 'timestep': 1}, {'agent': 0, 'loc': [(2, 0)], 'timestep': 2},
{'agent': 0, 'loc': [(3, 0)], 'timestep': 3}, {'agent': 0, 'loc': [(3,
1)], 'timestep': 3}, {'agent': 4, 'loc': [(3, 1)], 'timestep': 3},
{'agent': 1, 'loc': [(4, 3)], 'timestep': 6}]

Expanded node collisions: []


 Found a solution!


CPU time (s):    0.19

Sum of costs:    41

Expanded nodes:  22

Generated nodes: 43

***Test paths on a simulation***
```

The found solution has a sum of costs that matches the min_sum-of-costs.csv provided in the project. In fact, when run as a batch on all the test files, the program found all minimum sum of cost paths as determined in the min_sum-of-costs.csv file.

Task 4 – CBS with Disjoint Splitting

Part 4.3 – Comparing Standard Splitting to Disjoint Splitting

After implementing CBS with disjoint splitting, the implementation was tested against the 50 given batch test cases. Disjoint splitting was much faster than standard splitting when it came to running those test cases. For the instance exp4.txt however, disjoint splitting and standard splitting expanded the same number of nodes (15). This was not as expected as the stricter constraints of disjoint splitting should have pruned more nodes, making the search more quickly find the goal node. However, these

results are not impossible as the number of expanded nodes can vary based on which agent is chosen to be constrained in disjoint splitting.

A second observation of running the two solvers on exp4 was that standard splitting was approximately 0.04s faster than its counterpart on a Surface Pro 6 with a 1.60GHz CPU and 8 GB of RAM. Over the batch of 50 instances however, the disjoint splitting turned out to be four times faster and expanded about one quarter of the nodes, if the search space was large (>25000 expanded nodes) for the MAPF instance. This would have to do with the stronger constraints implemented by the algorithm, pruning more nodes. A smaller MAPF instance would not have such a great difference in number of nodes pruned, but disjoint splitting never expanded more nodes than standard splitting in testing.