CMPT 419 REPORT

Bryan Wong
301381982

1.1 Independent scheduler output

```
***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
```

1.2 Prioritized planning output

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

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

***Run Prioritized***
CHILDLOC: (1, 5)

 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
```

1.4 Goal test conditions

The program needs to check if the current time step at the goal node is larger than the earliest goal time step in order to terminate.

1.5 Designing Constraints

Constraints

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

Solution

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

Sum of costs: 8

2.4 Addressing failures

No, my solver didn't terminate properly, instead it kept on running presumably forever. The program needs to calculate an upper bound for the path length such that if the current time step exceeds the upper bound it terminates.

2.5 Showing that prioritized planning is incompleted and suboptimal

I've provided a MAPF instance under custominstances/inst_1.txt such that prioritized planning cannot provide a collision free solution. In addition to that, if you swap the starting positions of two agents you will get an ordering of the agents such that prioritized planning will provide a collision free solution.

custominstances/inst_1.txt

```
5 7
@ @ @ @ @ @ @
@ . @ @ @ @ @
@ . . . . . @
@ . @ @ @ @ @
@ @ @ @ @ @ @
2
2 4 2 2
2 3 2 5
```

3.3 CBS w/ expanded nodes output

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

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

***Run CBS***
Generate node 0
Expand node 0
P cost 6
P constraints []
P paths [[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5)], [(1, 2), (1, 3),
(1, 4)]]
P collisions [{'a1': 0, 'a2': 1, 'loc': [(1, 4)], 'timestep': 3}]
Generate node 1

Generate node 2

Expand node 1
P cost 7
P constraints [{'agent': 0, 'loc': [(1, 4)], 'timestep': 3,
'positive': False}]
P paths [[(1, 1), (1, 2), (1, 3), (1, 3), (1, 4), (1, 5)], [(1, 2),
(1, 3), (1, 4)]]
P collisions [{'a1': 0, 'a2': 1, 'loc': [(1, 4)], 'timestep': 4}]
Generate node 3

Generate node 4

Expand node 2
P cost 8
P constraints [{'agent': 1, 'loc': [(1, 4)], 'timestep': 3,
'positive': False}]
P paths [[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5)], [(1, 2), (1, 3),
(1, 4), (1, 3), (1, 4)]]
P collisions [{'a1': 0, 'a2': 1, 'loc': [(1, 3), (1, 4)],
'timestep': 3}]


.
.
.
```

```
Expand node 14
P cost 8
P constraints [{'agent': 1, 'loc': [(1, 4)], 'timestep': 3,
'positive': False}, {'agent': 1, 'loc': [(1, 4), (1, 3)],
'timestep': 3, 'positive': False}, {'agent': 1, 'loc': [(1, 3)],
'timestep': 2, 'positive': False}, {'agent': 1, 'loc': [(1, 5), (1,
4)], 'timestep': 4, 'positive': False}, {'agent': 1, 'loc': [(1,
3), (1, 2)], 'timestep': 2, 'positive': False}]
P paths [[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5)], [(1, 2), (1, 2),
(1, 2), (1, 3), (1, 4)]]
P collisions [{'a1': 0, 'a2': 1, 'loc': [(1, 2)], 'timestep': 1}]
Generate node 15

Generate node 16

Expand node 16
P cost 8
P constraints [{'agent': 1, 'loc': [(1, 4)], 'timestep': 3,
'positive': False}, {'agent': 1, 'loc': [(1, 4), (1, 3)],
'timestep': 3, 'positive': False}, {'agent': 1, 'loc': [(1, 3)],
'timestep': 2, 'positive': False}, {'agent': 1, 'loc': [(1, 5), (1,
4)], 'timestep': 4, 'positive': False}, {'agent': 1, 'loc': [(1,
3), (1, 2)], 'timestep': 2, 'positive': False}, {'agent': 1, 'loc':
[(1, 2)], 'timestep': 1, 'positive': False}]
P paths [[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5)], [(1, 2), (1, 3),
(2, 3), (1, 3), (1, 4)]]
P collisions []

 Found a solution!

CPU time (s):    0.01
Sum of costs:    8
Expanded nodes:  9
Generated nodes: 17
***Test paths on a simulation***
```

4.3 CBS w/ disjoint splitting performance

CBS with disjoint splitting on the MAPF instance “exp4.txt” would expand anywhere from 7 - 10 nodes.

Prioritized planning implementation

First, we find the shortest path from the starting node to the goal node of the highest priority agent. For each subsequent agent we determine any conflicts with higher priority agents and add constraints such that lower priority agents do not collide with higher priority agents. If no collision free path can be found for any agent then we return no path

In order to avoid other agents that have already reached their goal location. I've decided to define a special constraint where I set the time step of the constraint to -1 in order to indicate that the constraint lasts forever. We also need to deal with constraints at the goal location so that the program doesn't terminate before a constraint has been dealt with. So I calculate the earliest time step the program can exit, which is the largest time step of any constraint of the goal node. Finally, we need to check for the largest time step so that the program will not run forever if no solution exists. I set the largest time step to be the size of the map plus the length of the constraint table

CBS implementation

First, calculate the root node by finding the initial shortest paths for each agent from start node to goal node and find the cost and collisions of each path.
While OPEN is not empty let the parent node be the smallest cost node from OPEN.

- If there are no collisions in the parent node we have found the goal node and we return parent path
- Else, get the list of constraints with standard splitting
    - Standard splitting sets a constraint for agent 1 to not be at the location of the collision and the same constraint for agent 2

Create a child node that inherits the parent node with each new constraint from standard splitting. Find a new path for the constrained agent, if a path exists then detect the collisions of the new path, find the cost of the path and push the child node to OPEN.

CBS w/ disjoint splitting implementation

The implementation of CBS with disjoint splitting is similar to CBS with a few differences. Firstly, instead of standard splitting we use disjoint splitting where we choose a random agent from the collision to be constrained by a positive and negative constraint. I decided to add a new key to the constraint dictionary called "positive" with the value of True or False. The constraint is then handled in a_star() where is_constrained() checks if the next location satisfies the given constraint

which is either positive or negative. If a positive constraint was dealt with, we need to recalculate any agents that now violate the new positive constraint. We find the violating agents with paths_violate_constraint() that returns all agents that collide with the new path. We create a new negative constraint for the violating agent(s) and calculate a new path for each of them. If any of the violating agents or the positive constraint agent cannot find a path, then we do not push the child node to OPEN.