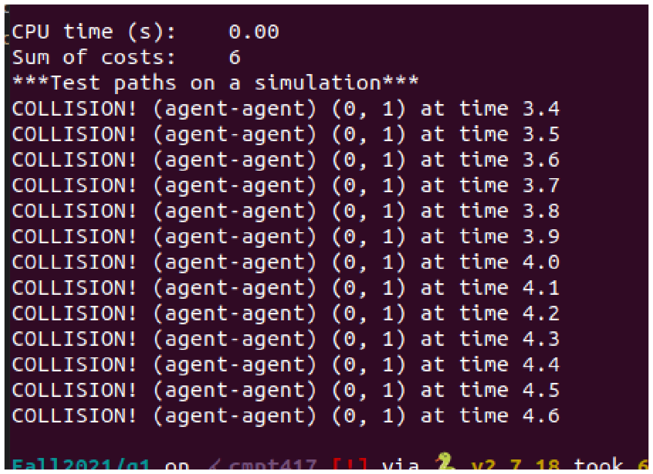
417 Individual Project Report

Yizhong Wang

301354602

Task 1: Implementing Space-Time A*

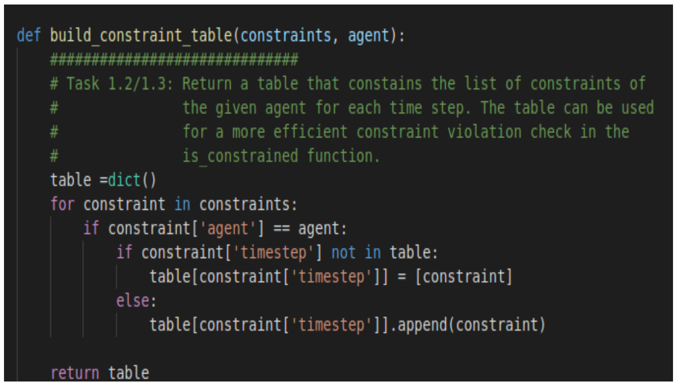
1.1:



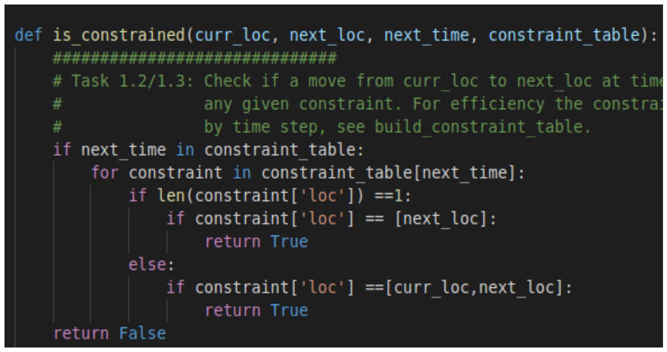
```
CPU time (s):     0.00
Sum of costs:     6
***Test paths on a simulation***
COLLISION! (agent-agent) (0, 1) at time 3.4
COLLISION! (agent-agent) (0, 1) at time 3.5
COLLISION! (agent-agent) (0, 1) at time 3.6
COLLISION! (agent-agent) (0, 1) at time 3.7
COLLISION! (agent-agent) (0, 1) at time 3.8
COLLISION! (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

Fall2021/a1 on ⎇ cmpt417 [!] via 🐍 v2.7.18 took 6
```

As we have only changed the key from space only to time-space for each node and a new direction (stay) for each agent. There is no constraint that agents know how to prevent the collision. Therefore the results are the same as before.

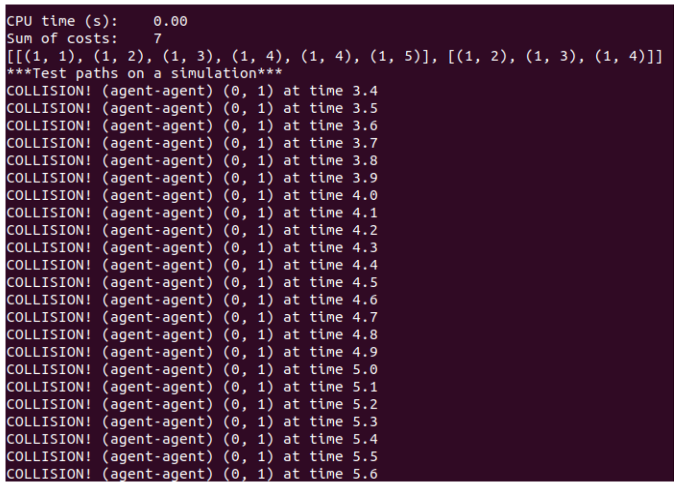
1.2



```python
def build_constraint_table(constraints, agent):
    ##############################
    # Task 1.2/1.3: Return a table that constains the list of constraints of
    #               the given agent for each time step. The table can be used
    #               for a more efficient constraint violation check in the
    #               is_constrained function.
    table =dict()
    for constraint in constraints:
        if constraint['agent'] == agent:
            if constraint['timestep'] not in table:
                table[constraint['timestep']] = [constraint]
            else:
                table[constraint['timestep']].append(constraint)

    return table
```



```python
def is_constrained(curr_loc, next_loc, next_time, constraint_table):
    ##############################
    # Task 1.2/1.3: Check if a move from curr_loc to next_loc at time
    #               any given constraint. For efficiency the constrai
    #               by time step, see build_constraint_table.
    if next_time in constraint_table:
        for constraint in constraint_table[next_time]:
            if len(constraint['loc']) ==1:
                if constraint['loc'] == [next_loc]:
                    return True
            else:
                if constraint['loc'] ==[curr_loc,next_loc]:
                    return True
    return False
```

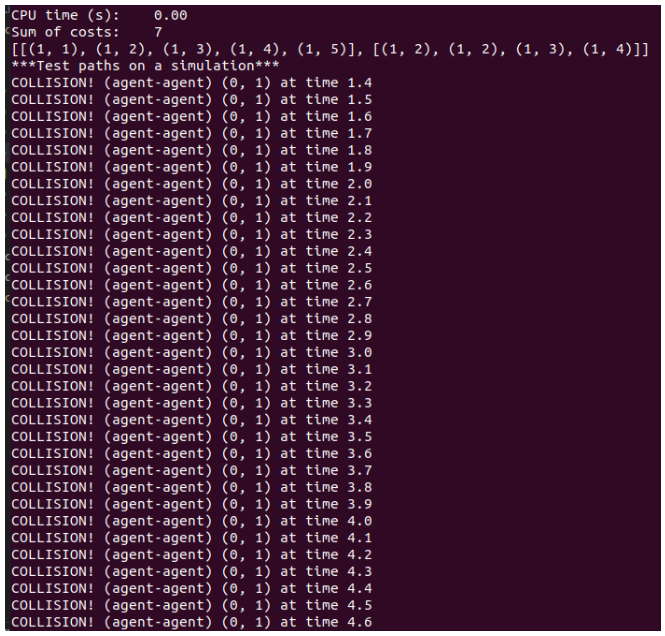


```
CPU time (s):     0.00
Sum of costs:     7
[[(1, 1), (1, 2), (1, 3), (1, 4), (1, 4), (1, 5)], [(1, 2), (1, 3), (1, 4)]]
***Test paths on a simulation***
COLLISION! (agent-agent) (0, 1) at time 3.4
COLLISION! (agent-agent) (0, 1) at time 3.5
COLLISION! (agent-agent) (0, 1) at time 3.6
COLLISION! (agent-agent) (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
```

As for efficiency and table's index is the time step. So that the table can be a dictionary, where the index is the time step.

For each index, it contains a list of constraints whose timesteps are equal to that index (for example, 1: [constraint1, constraint2]). If we can not find an index that is equal to the constraint we want to add to the table, add a new index where the index is the constraint's time step, otherwise, we just append the list. As we want to find whether the path or edge that we want to check is in the table. So, if we can find that list of constraints in the table, we check every constraint in that list. If the location of the constraint meets the next location the agent wants to go, then we do not push that child node to the open list. So in exp 1, as we prevent agent 1 from getting into (1,5) at time step 5, it will stay at (1,4) where agent 1 has already entered for one time step.

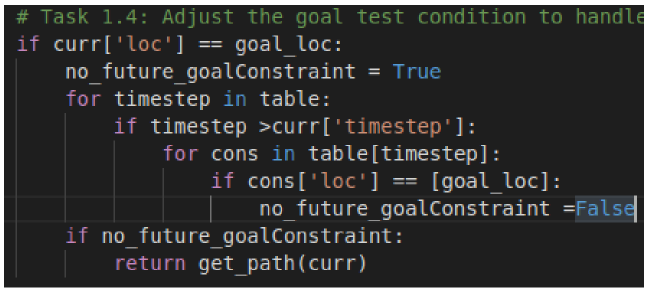
1.3



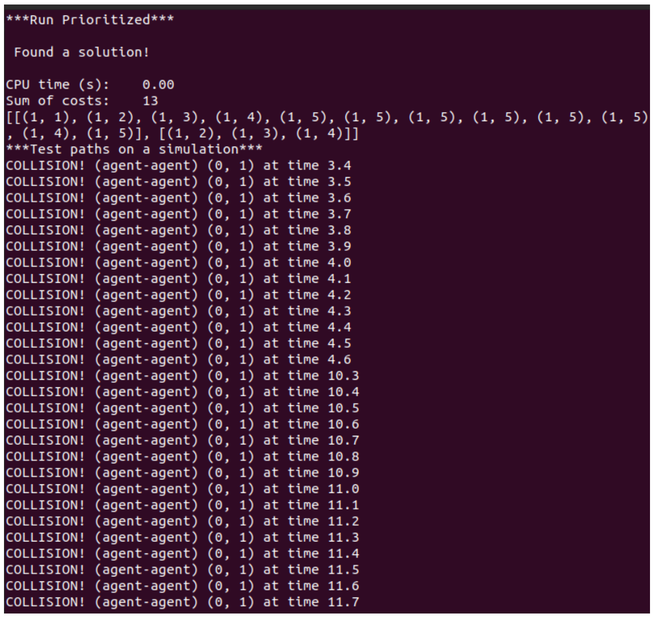
```
CPU time (s):    0.00
Sum of costs:    7
[[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5)], [(1, 2), (1, 2), (1, 3), (1, 4)]]
***Test paths on a simulation***
COLLISION! (agent-agent) (0, 1) at time 1.4
COLLISION! (agent-agent) (0, 1) at time 1.5
COLLISION! (agent-agent) (0, 1) at time 1.6
COLLISION! (agent-agent) (0, 1) at time 1.7
COLLISION! (agent-agent) (0, 1) at time 1.8
COLLISION! (agent-agent) (0, 1) at time 1.9
COLLISION! (agent-agent) (0, 1) at time 2.0
COLLISION! (agent-agent) (0, 1) at time 2.1
COLLISION! (agent-agent) (0, 1) at time 2.2
COLLISION! (agent-agent) (0, 1) at time 2.3
COLLISION! (agent-agent) (0, 1) at time 2.4
COLLISION! (agent-agent) (0, 1) at time 2.5
COLLISION! (agent-agent) (0, 1) at time 2.6
COLLISION! (agent-agent) (0, 1) at time 2.7
COLLISION! (agent-agent) (0, 1) at time 2.8
COLLISION! (agent-agent) (0, 1) at time 2.9
COLLISION! (agent-agent) (0, 1) at time 3.0
COLLISION! (agent-agent) (0, 1) at time 3.1
COLLISION! (agent-agent) (0, 1) at time 3.2
COLLISION! (agent-agent) (0, 1) at time 3.3
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
```

As edge constraint has two locations (the current location and the next location where the agent wants to go). Each edge constraint will be the list that length is 2. The method will be similar to the vertex constraint, where if that constraint length is larger than 1, then it will be an edge constraint. And if the agent's [current location, next location] meets the location of the constraint, we know that agent can not go through that edge, then we purne that direction option in our search at that time step. In the exp1, as we prevent agent 1 going from (1,2) to (1,3) at time step 1, as there is only one exist option for agent 1 which is stay at (1,2), it stays at (1,2) at time step 1, which causes the collision with agent 0 in every step until agent 1 meets its goal.

1.4



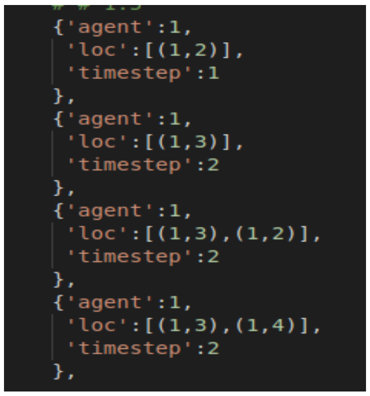
```python
# Task 1.4: Adjust the goal test condition to handle
if curr['loc'] == goal_loc:
    no_future_goalConstraint = True
    for timestep in table:
        if timestep >curr['timestep']:
            for cons in table[timestep]:
                if cons['loc'] == [goal_loc]:
                    no_future_goalConstraint =False
    if no_future_goalConstraint:
        return get_path(curr)
```



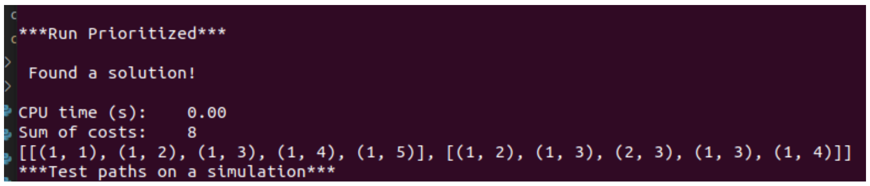
```
***Run Prioritized***

 Found a solution!

CPU time (s):    0.00
Sum of costs:    13
[[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 5), (1, 5), (1, 5), (1, 5), (1, 5)
, (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 10.3
COLLISION! (agent-agent) (0, 1) at time 10.4
COLLISION! (agent-agent) (0, 1) at time 10.5
COLLISION! (agent-agent) (0, 1) at time 10.6
COLLISION! (agent-agent) (0, 1) at time 10.7
COLLISION! (agent-agent) (0, 1) at time 10.8
COLLISION! (agent-agent) (0, 1) at time 10.9
COLLISION! (agent-agent) (0, 1) at time 11.0
COLLISION! (agent-agent) (0, 1) at time 11.1
COLLISION! (agent-agent) (0, 1) at time 11.2
COLLISION! (agent-agent) (0, 1) at time 11.3
COLLISION! (agent-agent) (0, 1) at time 11.4
COLLISION! (agent-agent) (0, 1) at time 11.5
COLLISION! (agent-agent) (0, 1) at time 11.6
COLLISION! (agent-agent) (0, 1) at time 11.7
```

As we want to check whether a future constraint exists after the agent meets its goal. The method is that, check those constraints that time steps are bigger than the time step when the agent meets its goal. So, the code will only compare those constraints in the table where the list's index is greater than the timestep when the agent arrives at the goal location. And if that constraint's location meets the goal location, that means we cannot just return the solution but have to wait to execute the constraint. So as there is a constraint at (1,5) at time step 10 for agent 0, which is after agent 0 arrives at its goal location. It has to wait until step 10 and move to another vertex, in this case the only option is (1,4), which will collide with agent 0, which in the animation, agent 0 waits at (1,5) until time step 10, moves to (1,4) and goes back to (1,5).

1.5



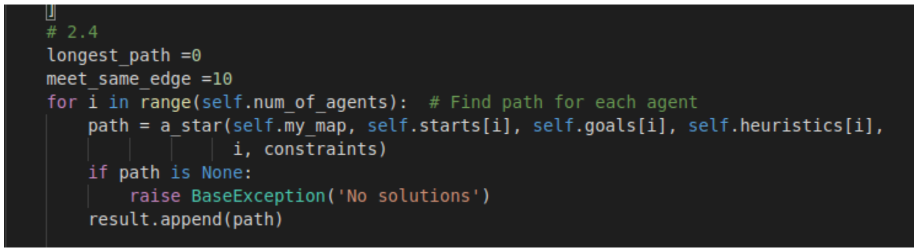
```python
    {'agent':1,
     'loc':[(1,2)],
     'timestep':1
    },
    {'agent':1,
     'loc':[(1,3)],
     'timestep':2
    },
    {'agent':1,
     'loc':[(1,3),(1,2)],
     'timestep':2
    },
    {'agent':1,
     'loc':[(1,3),(1,4)],
     'timestep':2
    },
```



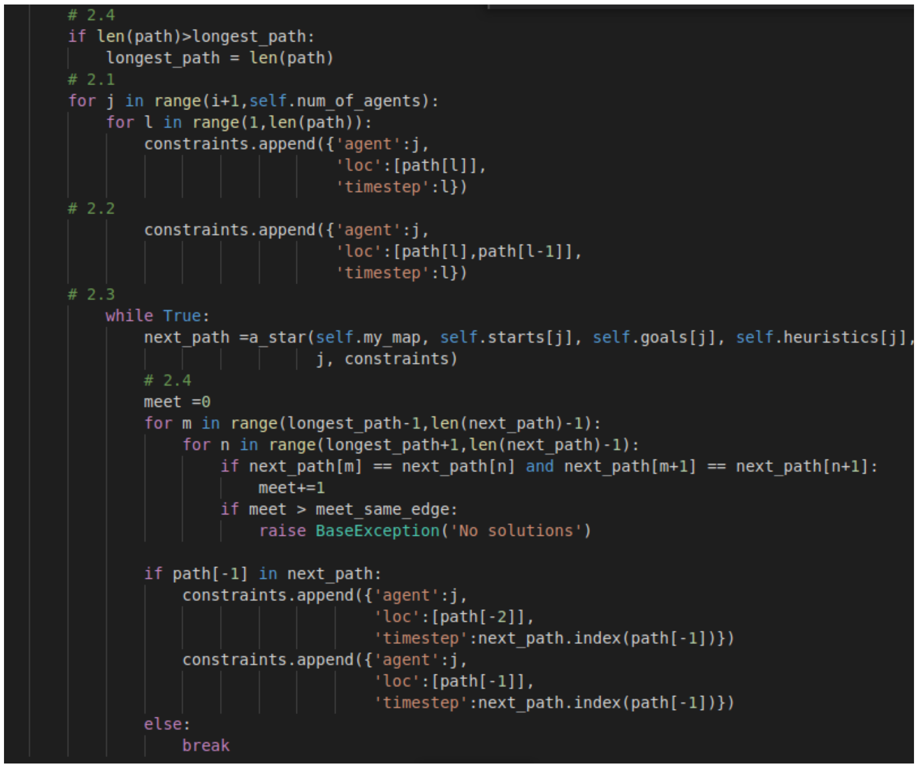
```
***Run Prioritized***

 Found a solution!

CPU time (s):    0.00
Sum of costs:    8
[[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5)], [(1, 2), (1, 3), (2, 3), (1, 3), (1, 4)]]
***Test paths on a simulation***
```

As we want to let agent 1 give the way to agent 0, in order to avoid the collision. That means agent 1 has to go to (2,3) at time step 2. So, the constraints will force agent 1 to go down and get into (2,3). Also, the first constraint in the screenshot is needed as otherwise agent 1 will choose to stay at the (1,2) at time step 1 for the optimal solution, which is a result of colliding with agent 0 from (1,2) to (1,4).

Task 2: Implementing Prioritized Planning



```
    # 2.4
    longest_path =0
    meet_same_edge =10
    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, constraints)
        if path is None:
            raise BaseException('No solutions')
        result.append(path)
```



```
        # 2.4
        if len(path)>longest_path:
            longest_path = len(path)
        # 2.1
        for j in range(i+1,self.num_of_agents):
            for l in range(1,len(path)):
                constraints.append({'agent':j,
                                    'loc':[path[l]],
                                    'timestep':l})
        # 2.2
                constraints.append({'agent':j,
                                    'loc':[path[l],path[l-1]],
                                    'timestep':l})
        # 2.3
            while True:
                next_path =a_star(self.my_map, self.starts[j], self.goals[j], self.heuristics[j],
                                  j, constraints)
                # 2.4
                meet =0
                for m in range(longest_path-1,len(next_path)-1):
                    for n in range(longest_path+1,len(next_path)-1):
                        if next_path[m] == next_path[n] and next_path[m+1] == next_path[n+1]:
                            meet+=1
                        if meet > meet_same_edge:
                            raise BaseException('No solutions')

                if path[-1] in next_path:
                    constraints.append({'agent':j,
                                        'loc':[path[-2]],
                                        'timestep':next_path.index(path[-1])})
                    constraints.append({'agent':j,
                                        'loc':[path[-1]],
                                        'timestep':next_path.index(path[-1])})
                else:
                    break
```

2.1
As each agent who has lower priority can not go to the place where higher priority agents are at each timestep. So When finding the path for an agent, we should add constraints for those vertices that are in the path, for all agents that have lower priority. Which in this case, it still can not solve the collision in exp2_1, as it only prevents agent 1 from staying at (1,4) at timestep 3, but on constraint on other four directions, so agent 1 still can go to (1,3) at timestep 3, which cause a edge collision.

2.2
As we want to prevent an edge collision between two agents, for example, agent 0 will go from a to b, the only thing we need to do is to prevent other agents from going from b to a at that

timestep. So we will only need to at that edge constraint where ‘loc is [b,a] to the constraint table.

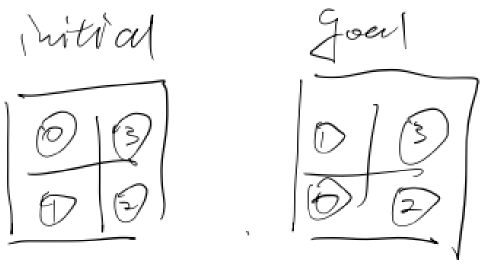
2.3

As when an agent arrives at its goal location, we can see that location as a wall for other lower priority agents. So one approach is to assign the vertex constraint which location is that agent location, as long as those lower priority agents do not arrive at their goal location, and will go to that agent goal location, and vertex constraint where the lower priority agents are at the last timestep, to prevent agents go back and forth between its last location and current location endlessly.

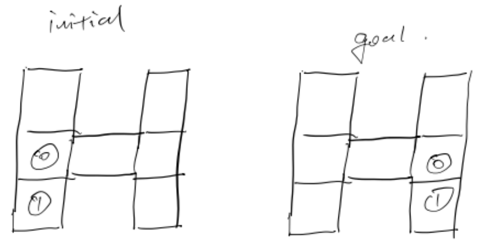
2.4

As we can not know the path that each agent will go unless we complete all the findings, there is no solution for the minimal upper bound that can be assigned before the algorithm starts to find the path for agent 0. As we keep assigning vertex constraints to the lower priority agent in 2.3, we set an upper bound where an agent through an edge for several times (10 in the code, as we can give some tolerance which allows the algorithm to try as best as possible to find the collision free solution). If it can not find that solution, which shows the agent arriving on the same path for a couple of times, then terminates the whole search and returns an error of no solution.

2.5



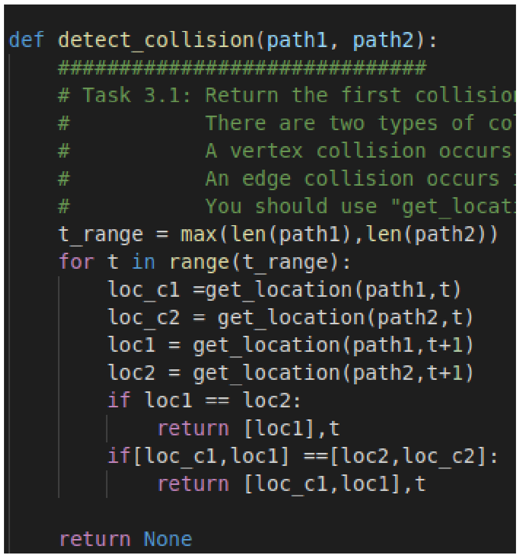
This will be an example of an instance for which prioritized planning does not find a collision-free solution, no matter which ordering of the agents it uses. As for all agents the option is either spin all together in clockwise or counter clockwise, or no agent will move. As the goal map shows that agent 1 and agent 0 will swap their position which no matter how the priority changes, there will be an edge collision.



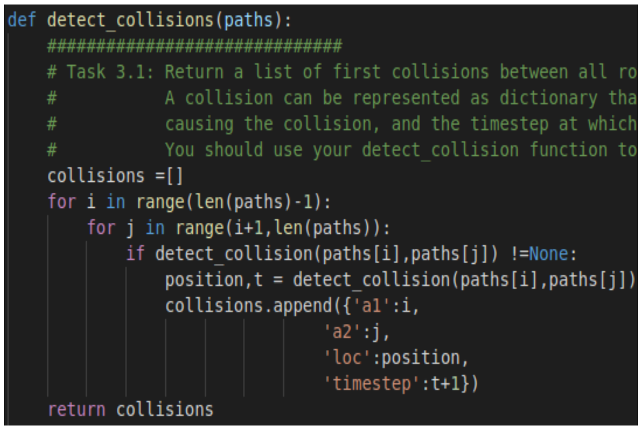
This will be an example of an instance for which prioritized planning does not find a collision-free solution for a given ordering of the agents. As in this ordering where agent 0 goes first, when it gets to the goal location it will block the only way that agent 1 needs to go. But if they swap the order, agent 0 will go to the top right block to give the way to agent 1.

Task 3: Implementing Conflict-Based Search

3.1



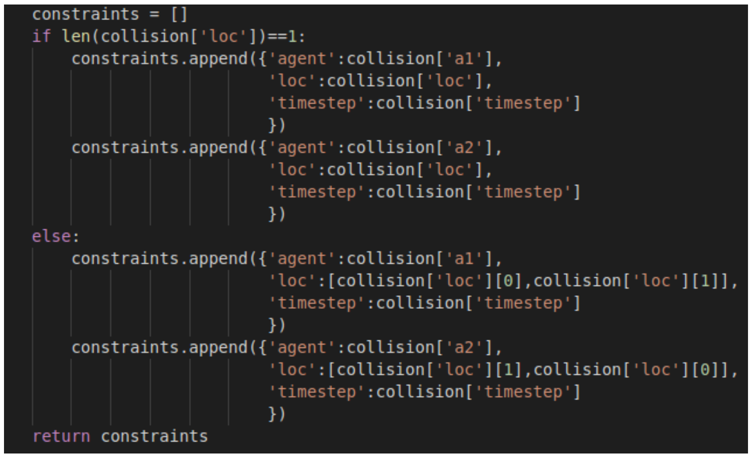
```
def detect_collision(path1, path2):
    ##############################
    # Task 3.1: Return the first collisio
    #           There are two types of co
    #           A vertex collision occurs
    #           An edge collision occurs
    #           You should use "get_locat
    t_range = max(len(path1),len(path2))
    for t in range(t_range):
        loc_c1 =get_location(path1,t)
        loc_c2 = get_location(path2,t)
        loc1 = get_location(path1,t+1)
        loc2 = get_location(path2,t+1)
        if loc1 == loc2:
            return [loc1],t
        if[loc_c1,loc1] ==[loc2,loc_c2]:
            return [loc_c1,loc1],t

    return None
```



```
def detect_collisions(paths):
    ##############################
    # Task 3.1: Return a list of first collisions between all ro
    #           A collision can be represented as dictionary tha
    #           causing the collision, and the timestep at which
    #           You should use your detect_collision function to
    collisions =[]
    for i in range(len(paths)-1):
        for j in range(i+1,len(paths)):
            if detect_collision(paths[i],paths[j]) !=None:
                position,t = detect_collision(paths[i],paths[j])
                collisions.append({'a1':i,
                                   'a2':j,
                                   'loc':position,
                                   'timestep':t+1})
    return collisions
```

To detect collisions we need to compare every node in each pair of paths. In detect_collision function, we will compare each vertex in that timestep. If loc1(next location in path 1at time step t) is same as loc2 (next location in path 2 at time step t), it means it is a vertex collision. And if [loc_c1(the current location in path 1 at time step t), loc_1) is equal to [loc_c2(the current location in path 2 at time step t),loc_2], it means it is an edge collison.
As we need to find all the edge collisions in the map. So we need to compare every path with others. For each path in the list, compare the path where it is behind it (i.e., path1 will compare path 2, path3, …, path n, and path 2 will compare path3, … ,path n) and insert the collision that found in that pair of paths into the collisions list, as path 2 has already been compared to path , so there is no need for go through that pair again when we looking for collision that path 2 has.

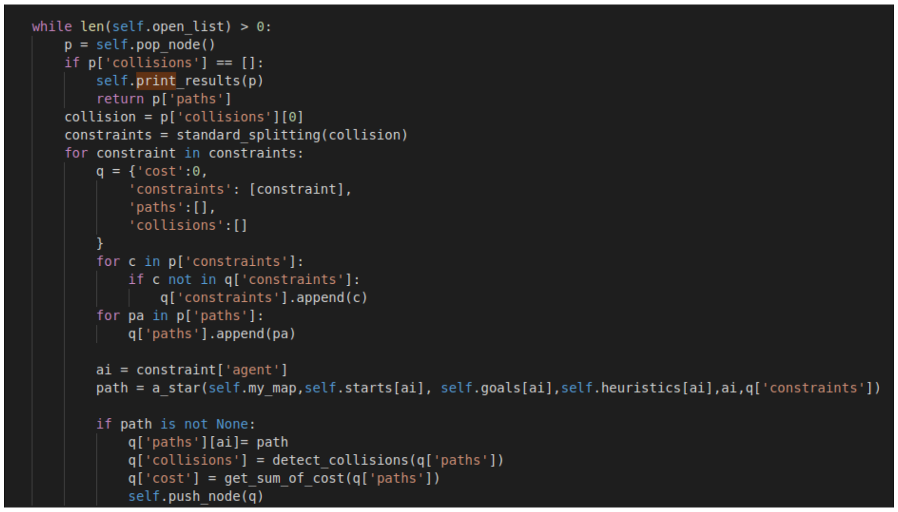
3.2



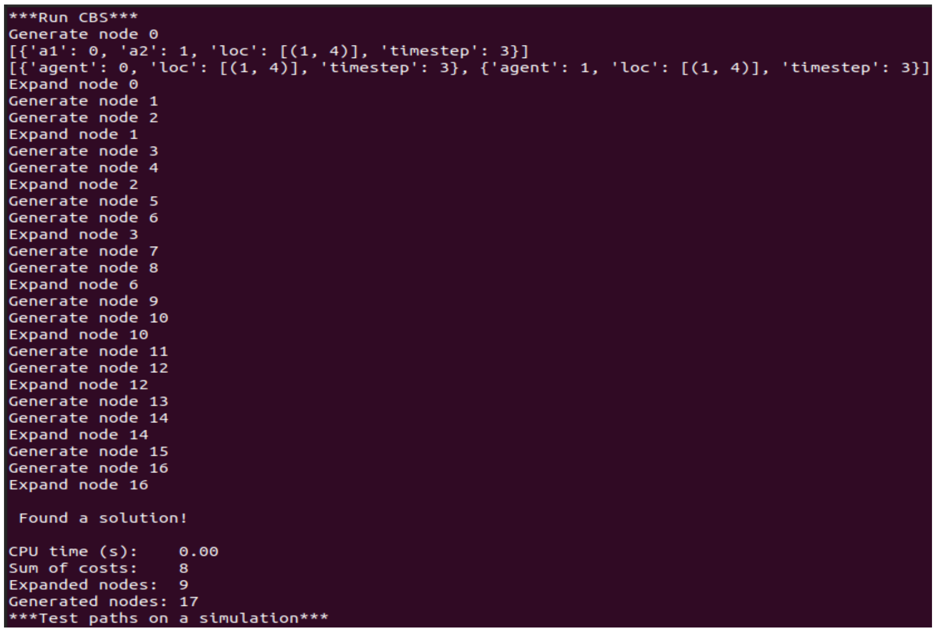
```
constraints = []
if len(collision['loc'])==1:
    constraints.append({'agent':collision['a1'],
                        'loc':collision['loc'],
                        'timestep':collision['timestep']
                        })
    constraints.append({'agent':collision['a2'],
                        'loc':collision['loc'],
                        'timestep':collision['timestep']
                        })
else:
    constraints.append({'agent':collision['a1'],
                        'loc':[collision['loc'][0],collision['loc'][1]],
                        'timestep':collision['timestep']
                        })
    constraints.append({'agent':collision['a2'],
                        'loc':[collision['loc'][1],collision['loc'][0]],
                        'timestep':collision['timestep']
                        })
return constraints
```

Each collision can generate a pair of vertex constraints or a pair of edge constraints, if the length of the location list of the collision is 1, then that means it will generate a set of vertex constraints. Otherwise, the collision is an edge collision, then it will transfer to a set of edge constraints.

3.3



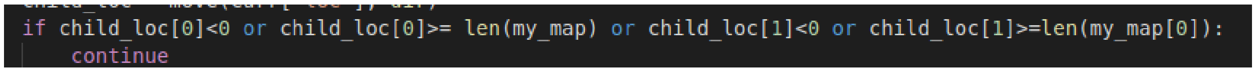
```
while len(self.open_list) > 0:
    p = self.pop_node()
    if p['collisions'] == []:
        self.print_results(p)
        return p['paths']
    collision = p['collisions'][0]
    constraints = standard_splitting(collision)
    for constraint in constraints:
        q = {'cost':0,
            'constraints': [constraint],
            'paths':[],
            'collisions':[]
        }
        for c in p['constraints']:
            if c not in q['constraints']:
                q['constraints'].append(c)
        for pa in p['paths']:
            q['paths'].append(pa)

        ai = constraint['agent']
        path = a_star(self.my_map,self.starts[ai], self.goals[ai],self.heuristics[ai],ai,q['constraints'])

        if path is not None:
            q['paths'][ai]= path
            q['collisions'] = detect_collisions(q['paths'])
            q['cost'] = get_sum_of_cost(q['paths'])
            self.push_node(q)
```



```
***Run CBS***
Generate node 0
[{'a1': 0, 'a2': 1, 'loc': [(1, 4)], 'timestep': 3}]
[{'agent': 0, 'loc': [(1, 4)], 'timestep': 3}, {'agent': 1, 'loc': [(1, 4)], 'timestep': 3}]
Expand node 0
Generate node 1
Generate node 2
Expand node 1
Generate node 3
Generate node 4
Expand node 2
Generate node 5
Generate node 6
Expand node 3
Generate node 7
Generate node 8
Expand node 6
Generate node 9
Generate node 10
Expand node 10
Generate node 11
Generate node 12
Expand node 12
Generate node 13
Generate node 14
Expand node 14
Generate node 15
Generate node 16
Expand node 16

 Found a solution!

CPU time (s):    0.00
Sum of costs:    8
Expanded nodes:  9
Generated nodes: 17
***Test paths on a simulation***
```

As for each collision, we will have two negative constraints for each agent that is in this collision. So for each agent we need to find a new path based on this constraint. As if this collision is solved, we can not guarantee that there will be no collision between any of two agents in the map. So that we need to keep searching for the collision, until we make sure we can not find one, and now the solution is optimal, and collision free.

3.4



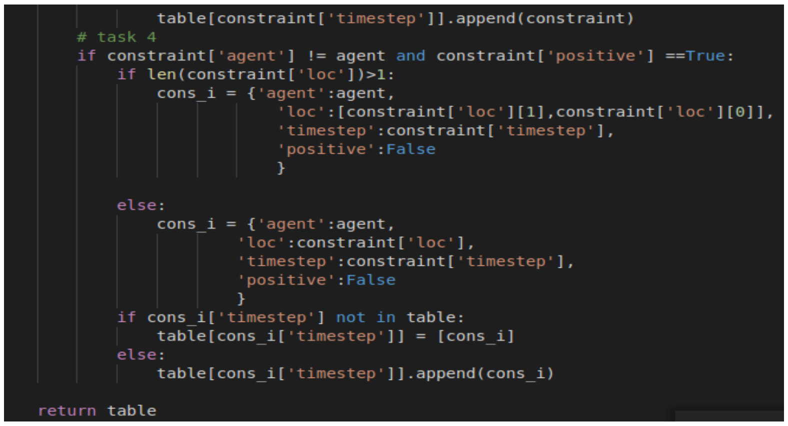
```
if child_loc[0]<0 or child_loc[0]>= len(my_map) or child_loc[1]<0 or child_loc[1]>=len(my_map[0]):
    continue
```

As there might be no walls at the edge of the map like some of the test files are, agents can go out of the map and trigger an error where the vertex will be negative (for example, (1,-1). So we need to apply an edge detection for the agent, which is when it is out of map (x-axis<0 or x-axis

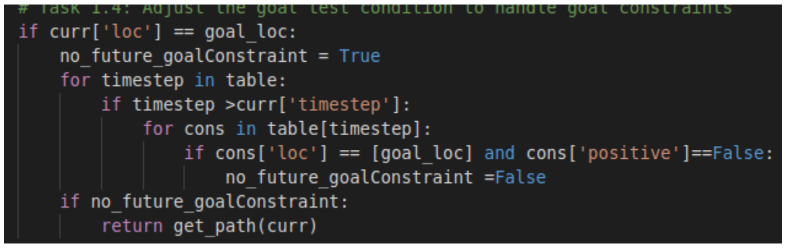
>map’s length, or y-axis <0 or y-axis> map’s width). We should prune that child node, as it is meaningless to keep expanding it.

Task 4: Implementing CBS with Disjoint Splitting

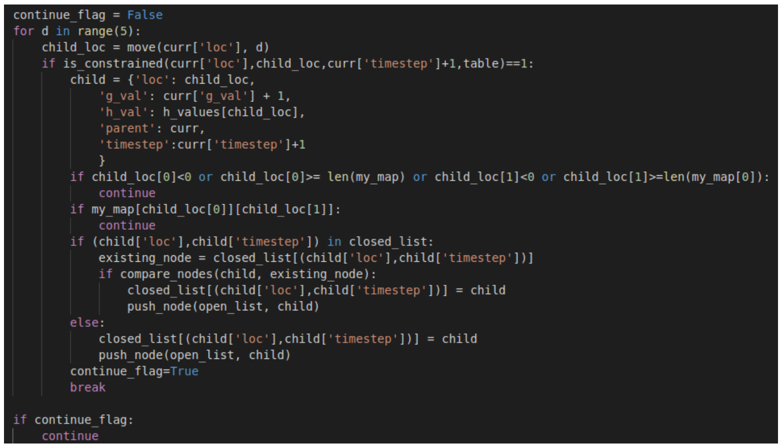
4.1



```
                table[constraint['timestep']].append(constraint)
        # task 4
        if constraint['agent'] != agent and constraint['positive'] ==True:
            if len(constraint['loc'])>1:
                cons_i = {'agent':agent,
                            'loc':[constraint['loc'][1],constraint['loc'][0]],
                            'timestep':constraint['timestep'],
                            'positive':False
                            }

            else:
                cons_i = {'agent':agent,
                        'loc':constraint['loc'],
                        'timestep':constraint['timestep'],
                        'positive':False
                        }
            if cons_i['timestep'] not in table:
                table[cons_i['timestep']] = [cons_i]
            else:
                table[cons_i['timestep']].append(cons_i)

    return table
```



```
# Task 1.4: Adjust the goal test condition to handle goal constraints
if curr['loc'] == goal_loc:
    no_future_goalConstraint = True
    for timestep in table:
        if timestep >curr['timestep']:
            for cons in table[timestep]:
                if cons['loc'] == [goal_loc] and cons['positive']==False:
                    no_future_goalConstraint =False
    if no_future_goalConstraint:
        return get_path(curr)
```



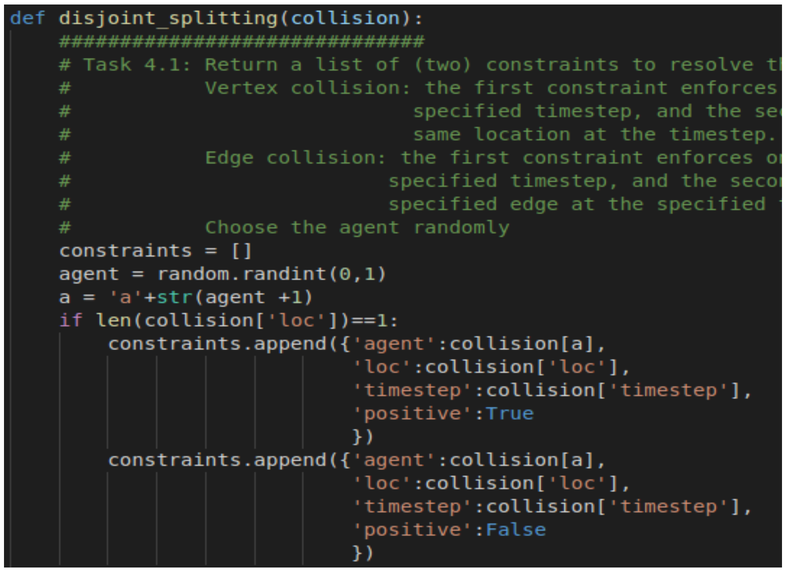
```
continue_flag = False
for d in range(5):
    child_loc = move(curr['loc'], d)
    if is_constrained(curr['loc'],child_loc,curr['timestep']+1,table)==1:
        child = {'loc': child_loc,
            'g_val': curr['g_val'] + 1,
            'h_val': h_values[child_loc],
            'parent': curr,
            'timestep':curr['timestep']+1
            }
        if child_loc[0]<0 or child_loc[0]>= len(my_map) or child_loc[1]<0 or child_loc[1]>=len(my_map[0]):
            continue
        if my_map[child_loc[0]][child_loc[1]]:
            continue
        if (child['loc'],child['timestep']) in closed_list:
            existing_node = closed_list[(child['loc'],child['timestep'])]
            if compare_nodes(child, existing_node):
                closed_list[(child['loc'],child['timestep'])] = child
                push_node(open_list, child)
        else:
            closed_list[(child['loc'],child['timestep'])] = child
            push_node(open_list, child)
        continue_flag=True
        break

if continue_flag:
    continue
```

As a positive constraint is force that agent is that force that agent goes to that specific location only, and prohibits all other agents from going to that location at that time step. When generate constraint table for A* search, we need to generate all negative constraints for each agent if there are positive constraints for other agents.

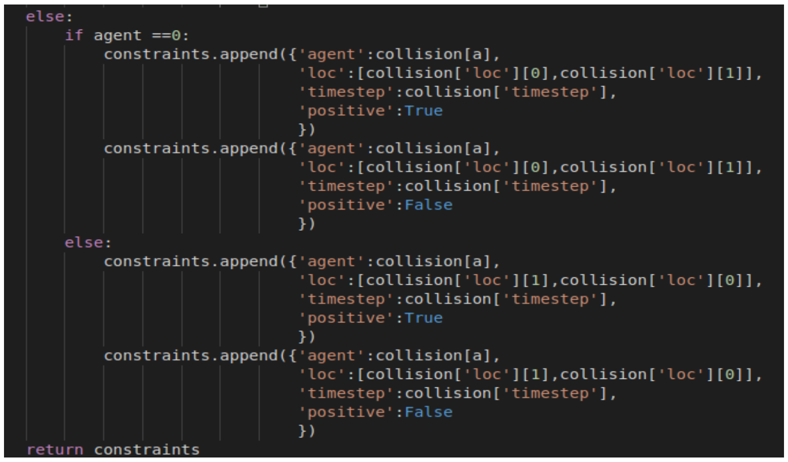
As in section 1.4, we only deal with whether there is a future negative constraint after an agent arrives at its goal location. As now we have a positive constraint, the code needs to only wait to the negative constraint, as if we wait for a positive constraint, which means waiting at the goal location and when at that time force to go to the place the agent has already been at, which means just wasting time.So, the code should only detect if there is future negative constraint or there will be a chance that the solution is not optimal due to the wasting steps.

When expand node in A*, as a positive constraint is forcing that agent to the specific place. Before expanding a child node with normal condition (no constraint) or negative constraint, we need to check if it has positive constraint at that time. If we have then we should only go to that location, and skip four other directions.

4.2



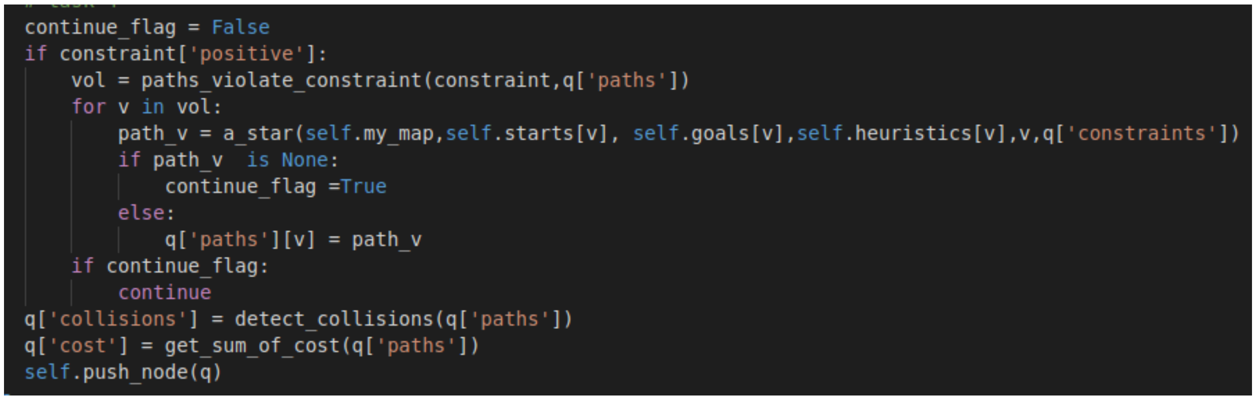
```
def disjoint_splitting(collision):
    ##############################
    # Task 4.1: Return a list of (two) constraints to resolve th
    #           Vertex collision: the first constraint enforces
    #                             specified timestep, and the se
    #                             same location at the timestep.
    #           Edge collision: the first constraint enforces o
    #                           specified timestep, and the secon
    #                           specified edge at the specified
    #           Choose the agent randomly
    constraints = []
    agent = random.randint(0,1)
    a = 'a'+str(agent +1)
    if len(collision['loc'])==1:
        constraints.append({'agent':collision[a],
                            'loc':collision['loc'],
                            'timestep':collision['timestep'],
                            'positive':True
                            })
        constraints.append({'agent':collision[a],
                            'loc':collision['loc'],
                            'timestep':collision['timestep'],
                            'positive':False
                            })
```



```
    else:
        if agent ==0:
            constraints.append({'agent':collision[a],
                                'loc':[collision['loc'][0],collision['loc'][1]],
                                'timestep':collision['timestep'],
                                'positive':True
                                })
            constraints.append({'agent':collision[a],
                                'loc':[collision['loc'][0],collision['loc'][1]],
                                'timestep':collision['timestep'],
                                'positive':False
                                })
        else:
            constraints.append({'agent':collision[a],
                                'loc':[collision['loc'][1],collision['loc'][0]],
                                'timestep':collision['timestep'],
                                'positive':True
                                })
            constraints.append({'agent':collision[a],
                                'loc':[collision['loc'][1],collision['loc'][0]],
                                'timestep':collision['timestep'],
                                'positive':False
                                })
    return constraints
```

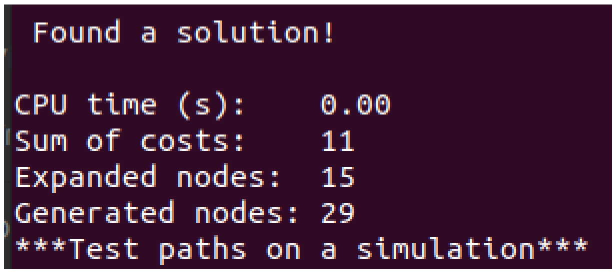
As when we generate a positive constraint, we need also generate a negative one for that agent. So, we randomly choose what agent we want to give the constraint, and see what kind of collision it is (vertex or edge) and transfer the collision to the constraint.

4.3

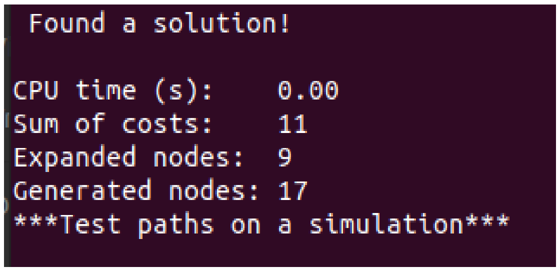


```
continue_flag = False
if constraint['positive']:
    vol = paths_violate_constraint(constraint,q['paths'])
    for v in vol:
        path_v = a_star(self.my_map,self.starts[v], self.goals[v],self.heuristics[v],v,q['constraints'])
        if path_v  is None:
            continue_flag =True
        else:
            q['paths'][v] = path_v
    if continue_flag:
        continue
q['collisions'] = detect_collisions(q['paths'])
q['cost'] = get_sum_of_cost(q['paths'])
self.push_node(q)
```

As we want to find a collision free path for each agent. For all agents that are influenced by that positive constraint, we need to update the path in order to solve collisions. So, firstly we need to find those agents who are influenced by the positive constraint, and then update paths to try to find a collision free solution. And if we can not find a solution for one or more agents that are influenced by the positive constraint, we should prune that child node, as there will be other solutions who have stronger constraints, and those subtrees may have optimal solutions.



```
 Found a solution!

CPU time (s):    0.00
Sum of costs:    11
Expanded nodes:  15
Generated nodes: 29
***Test paths on a simulation***
```



```
 Found a solution!

CPU time (s):    0.00
Sum of costs:    11
Expanded nodes:  9
Generated nodes: 17
***Test paths on a simulation***
```

CBS with disjoint splitting is faster than CBS with standard splitting.As the vertex where may be wasted due to two negative constraints (those constraints may prevent all agents from getting in that vertex at that time, and find a new collision free path which has more cost), but CBS with disjoint splitting force one of these agent get in this vertex, which may leads to found less collision in the future, which will be main reason of CBS with disjoint is faster.
CBS with disjoint splitting randomly chooses which agent to give the constraint, if there is a collision between two agents, how much faster is dependent on which agent is assigned to the constraint, so how less nodes will be generated and expanded is not constant, which means the nodes that CBS with disjoint splitting generated and expanded will less or equal to nodes that CBS with standard splitting generated and expanded.
For example, on the left of two outputs is the CBS with standard splitting, where 29 nodes are generated and 15 are expanded. On the right is CBS with disjoint splitting. CBS with disjoint splitting has 12 nodes less which are generated, and 6 nodes less are expanded.