CMPT 417 Report

Just a side note that some future code is included in screenshots since I did not know I had to take them until I was nearly done the project.

1. Implementing Space-Time A*

1.1. Searching in the Space-Time Domain

```
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. Handling Vertex Constraints

```
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

1.3. Adding Edge Constraints

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

1.4. Handling Goal Constraints

I modified the goal test by looking for the maximum timestep in the dictionary of constraints for the given agent, and only allowing the algorithm to return if we are at a timestep larger than this.

1.5. Designing Constraints

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

```python
def build_constraint_table(constraints, agent):
    ##############################
    # Task 1.2/1.3: Return a table that contains 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.
    constraint_table = dict()
    for constraint in constraints:
        if constraint['agent'] != agent:
            continue
        if constraint['timestep'] >= 0:
            if constraint['timestep'] in constraint_table:
                constraint_table[constraint['timestep']].append(constraint)
            else:
                constraint_table[constraint['timestep']] = [constraint]
    return constraint_table
```

```python
def is_constrained(curr_loc, next_loc, next_time, constraint_table, future_constraint_table):
    ##############################
    # Task 1.2/1.3: Check if a move from curr_loc to next_loc at time step next_time violates
    #               any given constraint. For efficiency the constraints are indexed in a constraint_table
    #               by time step, see build_constraint_table.
    # Add the below code in next

    for timestep in future_constraint_table:
        if abs(timestep) > next_time:
            break
        if next_loc == future_constraint_table[timestep][0]['loc'][0]:
            return True
    if next_time not in constraint_table:
        return False
    curr_constraints = constraint_table[next_time]
    for constraint in curr_constraints:
        if len(constraint['loc']) == 1 and next_loc == constraint['loc'][0]:
            if constraint['positive']:
                # print('positive')
                return False
            return True
        elif len(constraint['loc']) == 2 and curr_loc == constraint['loc'][0] and next_loc == constraint['loc'][1]:
            if constraint['positive']:
                # print('positive')
                return False
            return True
    return False
```

```python
def a_star(my_map, start_loc, goal_loc, h_values, agent, constraints):
    """ my_map      - binary obstacle map
        start_loc   - start position
        goal_loc    - goal position
        agent       - the agent that is being re-planned
        constraints - constraints defining where robot should or cannot go at each timestep
    """
    ##############################
    # Task 1.1: Extend the A* search to search in the space-time domain
    #           rather than space domain, only.
    # print('a* constraints', constraints)
    # print('agent', agent, 'goal', goal_loc)
    map_size = 0
    for line in my_map:
        for cell in line:
            if not cell:
                map_size = map_size + 1
    open_list = []
    closed_list = dict()
    h_value = h_values[start_loc]
    constraint_table = build_constraint_table(constraints, agent)
    future_constraint_table = build_future_constraint_table(constraints, agent)
    earliest_goal_timestep = max(constraint_table, key=int, default=0)

    root = {'loc': start_loc, 'g_val': 0, 'h_val': h_value, 'parent': None, 'timestep': 0}
    push_node(open_list, root)
    closed_list[(root['loc'], root['timestep'])] = root
```

```
    closed_list[(root['loc'], root['timestep'])] = root
    while len(open_list) > 0:
        curr = pop_node(open_list)
        #############################
        # Task 1.4: Adjust the goal test condition to handle goal constraints
        if curr['loc'] == goal_loc and curr['timestep'] >= earliest_goal_timestep:
            return get_path(curr)
        for dir in range(5):
            child_loc = move(curr['loc'], dir)
            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
            child = {'loc': child_loc,
                     'g_val': curr['g_val'] + 1,
                     'h_val': h_values[child_loc],
                     'parent': curr,
                     'timestep': curr['timestep'] + 1}
            # if len(future_constraint_table.keys()) > 0:
            #     if child['timestep'] > -min(future_constraint_table.keys()) + map_size:
            if child['timestep'] > earliest_goal_timestep + map_size:
                    return None
            if is_constrained(curr['loc'], child['loc'], child['timestep'], constraint_table, future_constraint_table):
                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
                    return None
            if is_constrained(curr['loc'], child['loc'], child['timestep'], constraint_table, future_constraint_table):
                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)
    return None  # Failed to find solutions
```

```python
def find_solution(self):
    """ Finds paths for all agents from their start locations to their goal locations."""

    start_time = timer.time()
    result = []
    constraints = []
    # constraints.append({'agent': 1, 'loc': [(1,3), (2,3)], 'timestep': 3, 'positive': False})
    # constraints.append({'agent': 1, 'loc': [(1,5)], 'timestep': 4, 'positive': False}) # Test
    # constraints.append({'agent': 2, 'loc': [(3,4)], 'timestep': 5, 'positive': False}) # Test
    # constraints.append({'agent': 1, 'loc': [(1, 2), (1, 3)], 'timestep': 1, 'positive': False}) # Test
    # constraints.append({'agent': 0, 'loc': [(1, 5)], 'timestep': 10, 'positive': False}) # Test

    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)

        ##############################
        # Task 2: Add constraints here
        #         Useful variables:
        #            * path contains the solution path of the current (i'th) agent, e.g., [(1,1),(1,2),(1,3)]
        #            * self.num_of_agents has the number of total agents
        #            * constraints: array of constraints to consider for future A* searches
```

2. Implementing Prioritized Planning

2.1. Adding Vertex Constraints

2.2. Adding Edge Constraints

2.3. Adding Additional Constraints

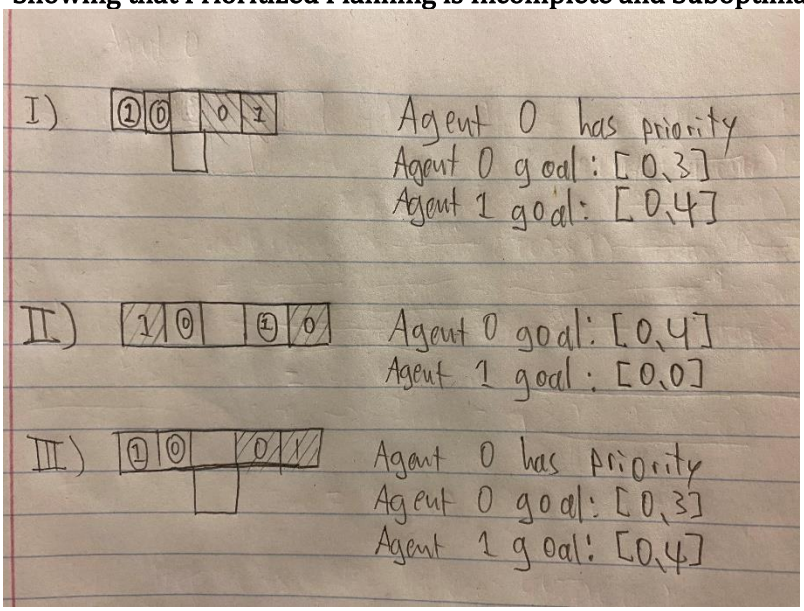
2.4. Addressing Failures

In Exp2_3, agent 1 has higher priority than agent 0, which means that his path is planned first. Agent 1's path is first made according to A*, with no constraints, but when agent 1 reaches his goal, he completely blocks agent 0's path to his respective goal. Agent 1 has already reached his goal and his search has terminated, so he cannot move, meaning that Agent 0 is stuck in an infinite loop. I changed the code to have an upper bound on the current agent equal to the timestep which the previous agent was at when he reached his goal plus the size of the map, since it would not make sense for the current agent to traverse a length greater than the size of the map after surpassing the timestep of the previous agent.

```
    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)

        ##############################
        # Task 2: Add constraints here
        #         Useful variables:
        #            * path contains the solution path of the current (i'th) agent, e.g., [(1,1),(1,2),(1,3)]
        #            * self.num_of_agents has the number of total agents
        #            * constraints: array of constraints to consider for future A* searches

        if i+1 < self.num_of_agents:
            for j, loc in enumerate(path):
                for k in range(self.num_of_agents - (i+1)):
                    constraints.append({'agent': self.num_of_agents - (k+1), 'loc': [path[j]], 'timestep': j, 'positive': False})
                    constraints.append({'agent': self.num_of_agents - (k+1), 'loc': [path[j], path[j-1]], 'timestep': j, 'positive': False})
                    if j == len(path) - 1:
                        constraints.append({'agent': self.num_of_agents - (k + 1), 'loc': [path[j]], 'timestep': -j, 'positive': False})
            # print('conz', constraints)

        ##############################

    self.CPU_time = timer.time() - start_time

    print("\n Found a solution! \n")
```

```
def build_future_constraint_table(constraints, agent):
    future_constraint_table = dict()
    for constraint in constraints:
        if constraint['agent'] != agent:
            continue
        if constraint['timestep'] < 0:
            if constraint['timestep'] in future_constraint_table:
                future_constraint_table[constraint['timestep']].append(constraint)
            else:
                future_constraint_table[constraint['timestep']] = [constraint]
    return future_constraint_table
```

2.5. Showing that Prioritized Planning is Incomplete and Suboptimal



3. Implementing Conflict-Based Search (CBS)

3.1. Detecting Collisions

3.2. Converting Collisions to Constraints

3.3. Implementing the High-Level Search

Generate node 0
Expand node 0
expanded node: {'cost': 6, 'constraints': [], 'paths': [[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5)], [(1, 2), (1, 3), (1, 4)]], 'collisions': [{'a1': 0, 'a2': 1, 'loc': [(1, 4)], 'timestep': 3}]}
Generate node 1
Generate node 2
Expand node 1
expanded node: {'cost': 7, 'constraints': [{'agent': 0, 'loc': [(1, 4)], 'timestep': 3}], 'paths': [[(1, 1), (1, 2), (1, 3), (1, 3), (1, 4), (1, 5)], [(1, 2), (1, 3), (1, 4)]], 'collisions': [{'a1': 0, 'a2': 1, 'loc': [(1, 4)], 'timestep': 4}]}
Generate node 3
Generate node 4
Expand node 2
expanded node: {'cost': 8, 'constraints': [{'agent': 1, 'loc': [(1, 4)], 'timestep': 3}], 'paths': [[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5)], [(1, 2), (1, 3), (1, 4), (1, 3), (1, 4)]], 'collisions': [{'a1': 0, 'a2': 1, 'loc': [(1, 3), (1, 4)], 'timestep': 3}]}
Generate node 5
Generate node 6
Expand node 3
expanded node: {'cost': 8, 'constraints': [{'agent': 0, 'loc': [(1, 4)], 'timestep': 3}, {'agent': 0, 'loc': [(1, 4)], 'timestep': 4}], 'paths': [[(1, 1), (1, 2), (1, 3), (1, 3), (1, 3), (1, 4), (1, 5)], [(1, 2), (1, 3), (1, 4)]], 'collisions': [{'a1': 0, 'a2': 1, 'loc': [(1, 4)], 'timestep': 5}]}
Generate node 7
Generate node 8
Expand node 6

expanded node: {'cost': 8, 'constraints': [{'agent': 1, 'loc': [(1, 4)], 'timestep': 3}, {'agent': 1, 'loc': [(1, 4), (1, 3)], 'timestep': 3}], 'paths': [[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5)], [(1, 2), (1, 3), (1, 3), (1, 3), (1, 4)]], 'collisions': [{'a1': 0, 'a2': 1, 'loc': [(1, 3)], 'timestep': 2}]}
Generate node 9
Generate node 10
Expand node 10
expanded node: {'cost': 8, 'constraints': [{'agent': 1, 'loc': [(1, 4)], 'timestep': 3}, {'agent': 1, 'loc': [(1, 4), (1, 3)], 'timestep': 3}, {'agent': 1, 'loc': [(1, 3)], 'timestep': 2}
], 'paths': [[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5)], [(1, 2), (1, 3), (1, 4), (1, 5), (1, 4)]], 'collisions': [{'a1': 0, 'a2': 1, 'loc': [(1, 4), (1, 5)], 'timestep': 4}]}
Generate node 11
Generate node 12
Expand node 5
expanded node: {'cost': 9, 'constraints': [{'agent': 1, 'loc': [(1, 4)], 'timestep': 3}, {'agent': 0, 'loc': [(1, 3), (1, 4)], 'timestep': 3}], 'paths': [[(1, 1), (1, 2), (1, 3), (1, 3), (1, 4), (1, 5)], [(1, 2), (1, 3), (1, 4), (1, 3), (1, 4)]], 'collisions': [{'a1': 0, 'a2': 1, 'loc': [(1, 3)], 'timestep': 3}]}
Generate node 13
Generate node 14
Expand node 7
expanded node: {'cost': 9, 'constraints': [{'agent': 0, 'loc': [(1, 4)], 'timestep': 3}, {'agent': 0, 'loc': [(1, 4)], 'timestep': 4}, {'agent': 0, 'loc': [(1, 4)], 'timestep': 5}], 'path
s': [[(1, 1), (1, 2), (1, 3), (1, 3), (1, 3), (1, 3), (1, 4), (1, 5)], [(1, 2), (1, 3), (1, 4)]], 'collisions': [{'a1': 0, 'a2': 1, 'loc': [(1, 4)], 'timestep': 6}]}
Generate node 15
Generate node 16
Expand node 9
expanded node: {'cost': 9, 'constraints': [{'agent': 1, 'loc': [(1, 4)], 'timestep': 3}, {'agent': 1, 'loc': [(1, 4), (1, 3)], 'timestep': 3}, {'agent': 0, 'loc': [(1, 3)], 'timestep': 2}
], 'paths': [[(1, 1), (1, 2), (1, 2), (1, 3), (1, 4), (1, 5)], [(1, 2), (1, 3), (1, 3), (1, 3), (1, 4)]], 'collisions': [{'a1': 0, 'a2': 1, 'loc': [(1, 3)], 'timestep': 3}]}
Generate node 17
Generate node 18
Expand node 11
expanded node: {'cost': 9, 'constraints': [{'agent': 1, 'loc': [(1, 4)], 'timestep': 3}, {'agent': 1, 'loc': [(1, 4), (1, 3)], 'timestep': 3}, {'agent': 1, 'loc': [(1, 3)], 'timestep': 2}
, {'agent': 0, 'loc': [(1, 4), (1, 5)], 'timestep': 4}], 'paths': [[(1, 1), (1, 2), (1, 3), (1, 4), (1, 4), (1, 5)], [(1, 2), (1, 3), (1, 4), (1, 5), (1, 4)]], 'collisions': [{'a1': 0, 'a
2': 1, 'loc': [(1, 4)], 'timestep': 4}]}
Generate node 19
Generate node 20
Expand node 12
expanded node: {'cost': 9, 'constraints': [{'agent': 1, 'loc': [(1, 4)], 'timestep': 3}, {'agent': 1, 'loc': [(1, 4), (1, 3)], 'timestep': 3}, {'agent': 1, 'loc': [(1, 3)], 'timestep': 2}
, {'agent': 1, 'loc': [(1, 5), (1, 4)], 'timestep': 4}], 'paths': [[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5)], [(1, 2), (1, 3), (1, 2), (1, 3), (1, 4), (1, 4)]], 'collisions': [{'a1': 0, 'a
2': 1, 'loc': [(1, 2), (1, 3)], 'timestep': 2}]}
Generate node 21
Generate node 22

Expand node 14
expanded node: {'cost': 9, 'constraints': [{'agent': 1, 'loc': [(1, 4)], 'timestep': 3}, {'agent': 0, 'loc': [(1, 3), (1, 4)], 'timestep': 3}, {'agent': 1, 'loc': [(1, 3)], 'timestep': 3}
], 'paths': [[(1, 1), (1, 2), (1, 3), (1, 3), (1, 4), (1, 5)], [(1, 2), (1, 3), (1, 4), (1, 5), (1, 4)]], 'collisions': [{'a1': 0, 'a2': 1, 'loc': [(1, 4)], 'timestep': 4}]}
Generate node 23
Generate node 24
Expand node 18
expanded node: {'cost': 9, 'constraints': [{'agent': 1, 'loc': [(1, 4)], 'timestep': 3}, {'agent': 1, 'loc': [(1, 4), (1, 3)], 'timestep': 3}, {'agent': 0, 'loc': [(1, 3)], 'timestep': 2}
, {'agent': 1, 'loc': [(1, 3)], 'timestep': 3}], 'paths': [[(1, 1), (1, 2), (1, 2), (1, 3), (1, 4), (1, 5)], [(1, 2), (1, 3), (1, 4), (1, 5), (1, 4)]], 'collisions': [{'a1': 0, 'a2': 1, '
loc': [(1, 4)], 'timestep': 4}]}
Generate node 25
Generate node 26
Expand node 22
expanded node: {'cost': 9, 'constraints': [{'agent': 1, 'loc': [(1, 4)], 'timestep': 3}, {'agent': 1, 'loc': [(1, 4), (1, 3)], 'timestep': 3}, {'agent': 1, 'loc': [(1, 3)], 'timestep': 2}
, {'agent': 1, 'loc': [(1, 5), (1, 4)], 'timestep': 4}, {'agent': 1, 'loc': [(1, 3), (1, 2)], 'timestep': 2}], 'paths': [[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5)], [(1, 2), (1, 2), (1, 2),
 (1, 3), (1, 4), (1, 4)]], 'collisions': [{'a1': 0, 'a2': 1, 'loc': [(1, 2)], 'timestep': 1}]}
Generate node 27
Generate node 28
Expand node 28
expanded node: {'cost': 9, 'constraints': [{'agent': 1, 'loc': [(1, 4)], 'timestep': 3}, {'agent': 1, 'loc': [(1, 4), (1, 3)], 'timestep': 3}, {'agent': 1, 'loc': [(1, 3)], 'timestep': 2}
, {'agent': 1, 'loc': [(1, 5), (1, 4)], 'timestep': 4}, {'agent': 1, 'loc': [(1, 3), (1, 2)], 'timestep': 2}, {'agent': 1, 'loc': [(1, 2)], 'timestep': 1}], 'paths': [[(1, 1), (1, 2), (1,
 3), (1, 4), (1, 5)], [(1, 2), (1, 3), (2, 3), (1, 3), (1, 4), (1, 4)]], 'collisions': [None]}

3.4. Testing your Implementation

```python
def detect_collision(path1, path2, a1, a2):
    ##############################
    # Task 3.1: Return the first collision that occurs between two robot paths (or None if there is no collision)
    #           There are two types of collisions: vertex collision and edge collision.
    #           A vertex collision occurs if both robots occupy the same location at the same timestep
    #           An edge collision occurs if the robots swap their location at the same timestep.
    #           You should use "get_location(path, t)" to get the location of a robot at time t.
    # min_path = path1 if len(path1) < len(path2) else path2
    max_len = max(len(path1), len(path2))
    for i in range(max_len):
        loc_a1_t1 = get_location(path1, i)
        loc_a2_t1 = get_location(path2, i)
        if loc_a1_t1 == loc_a2_t1:
            return {'a1': a1, 'a2': a2, 'loc': [loc_a1_t1], 'timestep': i}
        if i < max_len - 1:
            loc_a1_t2 = get_location(path1, i + 1)
            loc_a2_t2 = get_location(path2, i + 1)
            if loc_a1_t2 == loc_a2_t1 and loc_a2_t2 == loc_a1_t1:
                return {'a1': a1, 'a2': a2, 'loc': [loc_a1_t1, loc_a1_t2], 'timestep': i + 1}
```

```python
def detect_collisions(paths):
    ##############################
    # Task 3.1: Return a list of first collisions between all robot pairs.
    #           A collision can be represented as dictionary that contains the id of the two robots, the vertex or edge
    #           causing the collision, and the timestep at which the collision occurred.
    #           You should use your detect_collision function to find a collision between two robots.
    collisions = []
    num_agents = len(paths)
    for i in range(num_agents):
        for j in range(i + 1, num_agents):
            collisions.append(detect_collision(paths[i], paths[j], i, j))
    return collisions
    pass
def standard_splitting(collision):
    ##############################
    # Task 3.2: Return a list of (two) constraints to resolve the given collision
    #           Vertex collision: the first constraint prevents the first agent to be at the specified location at the
    #                             specified timestep, and the second constraint prevents the second agent to be at the
    #                             specified location at the specified timestep.
    #           Edge collision: the first constraint prevents the first agent to traverse the specified edge at the
    #                           specified timestep, and the second constraint prevents the second agent to traverse the
    #                           specified edge at the specified timestep
    if collision is None:
        return
    return [{'agent': collision['a1'], 'loc': collision['loc'], 'timestep': collision['timestep'], 'positive': False},
            {'agent': collision['a2'], 'loc': collision['loc'][::-1], 'timestep': collision['timestep'],
             'positive': False}]

# Generate the root node
# constraints   - list of constraints
# paths         - list of paths, one for each agent
#               [[(x11, y11), (x12, y12), ...], [(x21, y21), (x22, y22), ...], ...]
# collisions     - list of collisions in paths
root = {'cost': 0,
        'constraints': [],
        'paths': [],
        'collisions': []}
for i in range(self.num_of_agents):  # Find initial path for each agent
    path = a_star(self.my_map, self.starts[i], self.goals[i], self.heuristics[i],
                  i, root['constraints'])
    if path is None:
        raise BaseException('No solutions')
    root['paths'].append(path)
root['cost'] = get_sum_of_cost(root['paths'])
root['collisions'] = detect_collisions(root['paths'])
root['collisions'] = [collision for collision in root['collisions'] if collision != None]
self.push_node(root)

# Task 3.1: Testing
print('root', root['collisions'])
```

```
while len(self.open_list) > 0:
    curr = self.pop_node()
    if not len(curr['collisions']):
        return curr['paths']
    collision = curr['collisions'][0]
    constraints = standard_splitting(collision)
    for constraint in constraints:
        # print('constraint', constraint)
        child = {'cost': 0, 'constraints': [], 'paths': [], 'collisions': []}
        if curr['constraints']:
            child['constraints'] = list(curr['constraints'])
        child['constraints'].append(constraint)
        child['paths'] = list(curr['paths'])
        agent = constraint['agent']
        path = a_star(self.my_map, self.starts[agent], self.goals[agent], self.heuristics[agent],
                      agent, child['constraints'])
        if path is not None:
            child['paths'][agent] = path
            violation = False
            if constraint['positive']:
                violation_ids = paths_violate_constraint(constraint, child['paths'])
                for id in violation_ids:
                    # print('violate', id, path)
                    new_constraint = {'agent': id, 'loc': constraint['loc'], 'timestep': constraint['timestep'],
                                      'positive': False}
                    child['constraints'].append(new_constraint)
                    a_path = a_star(self.my_map, self.starts[id], self.goals[id], self.heuristics[id],
                           id, child['constraints'])
                    if a_path is None:
                        child['constraints'].append(new_constraint)
                        a_path = a_star(self.my_map, self.starts[id], self.goals[id], self.heuristics[id],
                               id, child['constraints'])
                        if a_path is None:
                            # print('getOut')
                            violation = True
                            break
                        else:
                            child['paths'][id] = a_path
                            # print('new path', child['paths'][id])
                if not violation:
                    child['collisions'] = detect_collisions(child['paths'])
                    child['collisions'] = [collision for collision in child['collisions'] if collision != None]
                    child['cost'] = get_sum_of_cost(child['paths'])
                    # print('pushing node', child)
                    self.push_node(child)
    self.print_results(root)
    return root['paths']

def print_results(self, node):
    print("\n Found a solution! \n")
    CPU_time = timer.time() - self.start_time
    print("CPU time (s):    {:.2f}".format(CPU_time))
    print("Sum of costs:    {}".format(get_sum_of_cost(node['paths'])))
    print("Expanded nodes:  {}".format(self.num_of_expanded))
    print("Generated nodes: {}".format(self.num_of_generated))
```

4. Implementing CBS with Disjoint Splitting

4.1. Supporting Positive Constraints

4.2. Converting Collisions to Constraints

4.3. Adjusting the High-Level Search

My code expands 15 nodes for both implementations.

```python
def disjoint_splitting(collision):
    ############################
    # Task 4.1: Return a list of (two) constraints to resolve the given collision
    #           Vertex collision: the first constraint enforces one agent to be at the specified location at the
    #                             specified timestep, and the second constraint prevents the same agent to be at the
    #                             same location at the timestep.
    #           Edge collision: the first constraint enforces one agent to traverse the specified edge at the
    #                           specified timestep, and the second constraint prevents the same agent to traverse the
    #                           specified edge at the specified timestep
    #           Choose the agent randomly
    if collision is None:
        return
    # pos_neg = True if random.randint(0, 1) else False
    # return [{'agent': collision['a1'], 'loc': collision['loc'], 'timestep': collision['timestep'], 'positive': pos_neg},
    #         {'agent': collision['a2'], 'loc': collision['loc'][::-1], 'timestep': collision['timestep'], 'positive': not pos_neg}]

    agent = 'a1' if random.randint(0, 1) else 'a2'
    if len(collision['loc']) == 1:
        return [{'agent': collision[agent], 'loc': collision['loc'], 'timestep': collision['timestep'], 'positive': True},
                {'agent': collision[agent], 'loc': collision['loc'], 'timestep': collision['timestep'], 'positive': False}]
    else:
        if agent == 'a1':
            return [{'agent': collision[agent], 'loc': collision['loc'], 'timestep': collision['timestep'], 'positive': True},
                    {'agent': collision[agent], 'loc': collision['loc'], 'timestep': collision['timestep'], 'positive': False}]
        else:
            return [{'agent': collision[agent], 'loc': collision['loc'][::-1], 'timestep': collision['timestep'], 'positive': True},
                    {'agent': collision[agent], 'loc': collision['loc'][::-1], 'timestep': collision['timestep'], 'positive': False}]
```

```
while len(self.open_list) > 0:
    curr = self.pop_node()
    if not len(curr['collisions']):
        return curr['paths']
    collision = curr['collisions'][0]
    constraints = standard_splitting(collision)
    for constraint in constraints:
        # print('constraint', constraint)
        child = {'cost': 0, 'constraints': [], 'paths': [], 'collisions': []}
        if curr['constraints']:
            child['constraints'] = list(curr['constraints'])
        child['constraints'].append(constraint)
        child['paths'] = list(curr['paths'])
        agent = constraint['agent']
        path = a_star(self.my_map, self.starts[agent], self.goals[agent], self.heuristics[agent],
                      agent, child['constraints'])
        if path is not None:
            child['paths'][agent] = path
            violation = False
            if constraint['positive']:
                violation_ids = paths_violate_constraint(constraint, child['paths'])
                for id in violation_ids:
                    # print('violate', id, path)
                    new_constraint = {'agent': id, 'loc': constraint['loc'], 'timestep': constraint['timestep'],
                                      'positive': False}
                    child['constraints'].append(new_constraint)
                    a_path = a_star(self.my_map, self.starts[id], self.goals[id], self.heuristics[id],
                           id, child['constraints'])
            child['constraints'].append(new_constraint)
            a_path = a_star(self.my_map, self.starts[id], self.goals[id], self.heuristics[id],
                   id, child['constraints'])
            if a_path is None:
                # print('getOut')
                violation = True
                break
            else:
                child['paths'][id] = a_path
                # print('new path', child['paths'][id])
    if not violation:
        child['collisions'] = detect_collisions(child['paths'])
        child['collisions'] = [collision for collision in child['collisions'] if collision != None]
        child['cost'] = get_sum_of_cost(child['paths'])
        # print('pushing node', child)
        self.push_node(child)
```