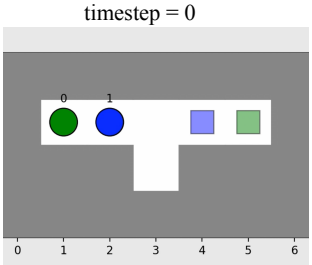
CMPT 417 Individual project report

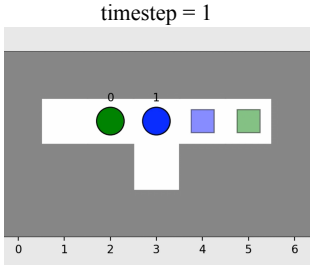
Task 1.1: Searching in the Space Time Domain

1. Included (0, 0) in the function move(loc, dir) for the wait action so that the agent waits in its current cell instead of moving to a neighboring cell when generating child nodes in the function a_star.
2. Included 'timestep' variable in the root and child node, and updated the key value for closed_list's dictionary to (cell, timestep).
3. Observed identical behavior and <u>output</u> as follows.

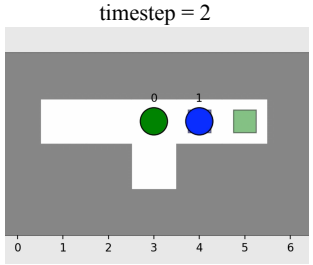
timestep = 0



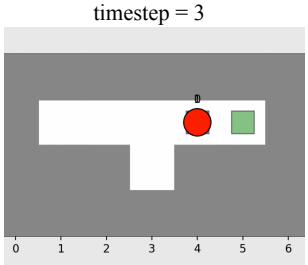
timestep = 1



timestep = 2



timestep = 3



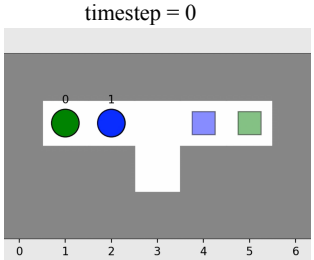
Terminal output:

```
jeongwoonsuh@Jeongwoonui-MacBookPro code_Task1 % python run_experiments.py --instance instances/exp1.txt --solver Independent
***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 4
COLLISION! (agent-agent) (0, 1) at time 4
COLLISION! (agent-agent) (0, 1) at time 4
COLLISION! (agent-agent) (0, 1) at time 4
COLLISION! (agent-agent) (0, 1) at time 4
COLLISION! (agent-agent) (0, 1) at time 4
COLLISION! (agent-agent) (0, 1) at time 4
COLLISION! (agent-agent) (0, 1) at time 4
COLLISION! (agent-agent) (0, 1) at time 4
```

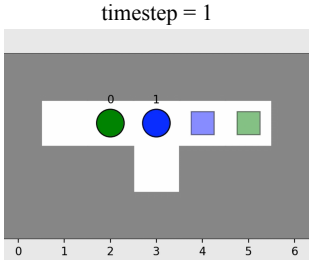
Task 1.2: Handling Vertex Constraints

1. Implemented build_constraint_table function which returns a table that contains the list of constraints such as agent's number, agent's location, and agent's time step which will be used in the a_star function to create a constraint_table.
2. Implemented is_constrained function which checks whether a move from curr_loc to next_loc at time step next_time violates any given constraints. I have implemented for both vertex and edge constraints.
3. Added the part where the function a_star checks the vertex constraint, whether the current child is violating the constraints, if it is, then ignore the action of the current child.
4. Added a given vertex constraint in prioritized.py that prohibits agent_0 from being at it's goal cell (1, 5) at time step 4 for testing.
5. <u>Output</u> as follows.

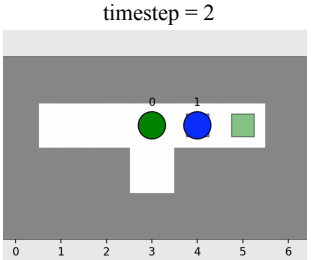
timestep = 0



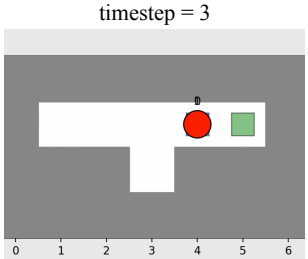
timestep = 1



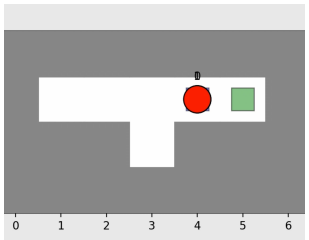
timestep = 2



timestep = 3



timestep = 5



Terminal output:

```
jeongwoonsuh@Jeongwoonui-MacBookPro code_Task1 % python run_experiments.py --instance instances/exp1.txt --solver Prioritized
***Import an instance***
Start locations
@ @ @ @ @ @ @
@ 0 1 . . . @
@ @ @ . @ @ @
@ @ @ @ @ @ @

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

***Run Prioritized***
('Agent[0]', 'path = ', [(1, 1), (1, 2), (1, 3), (1, 4), (1, 4), (1, 5)])
('Agent[1]', 'path = ', [(1, 2), (1, 3), (1, 4)])

 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 4
COLLISION! (agent-agent) (0, 1) at time 4
COLLISION! (agent-agent) (0, 1) at time 4
COLLISION! (agent-agent) (0, 1) at time 4
COLLISION! (agent-agent) (0, 1) at time 4
COLLISION! (agent-agent) (0, 1) at time 4
COLLISION! (agent-agent) (0, 1) at time 4
COLLISION! (agent-agent) (0, 1) at time 4
COLLISION! (agent-agent) (0, 1) at time 4
COLLISION! (agent-agent) (0, 1) at time 4
COLLISION! (agent-agent) (0, 1) at time 5
COLLISION! (agent-agent) (0, 1) at time 5
COLLISION! (agent-agent) (0, 1) at time 5
COLLISION! (agent-agent) (0, 1) at time 5
COLLISION! (agent-agent) (0, 1) at time 5
COLLISION! (agent-agent) (0, 1) at time 5
COLLISION! (agent-agent) (0, 1) at time 5
COLLISION! (agent-agent) (0, 1) at time 5
COLLISION! (agent-agent) (0, 1) at time 5
COLLISION! (agent-agent) (0, 1) at time 4
COLLISION! (agent-agent) (0, 1) at time 4
COLLISION! (agent-agent) (0, 1) at time 4
COLLISION! (agent-agent) (0, 1) at time 4
COLLISION! (agent-agent) (0, 1) at time 4
```

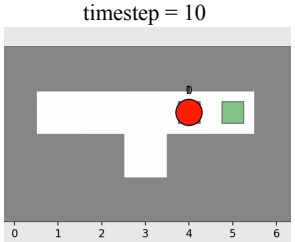
Task 1.3: Adding Edge Constraints

1. The function build_constraint_table is already implemented from Task 1.2
2. The function is_constrained is implemented for checking both vertex and edge constraints
3. In the function a_star, constraint_table is assigned using build_constraint_table to create table followed by given constraints. (same as Task 1.2)
4. Updated the function a_star to check the edge constraint that prohibit a given agent from moving from a given cell to another given cell at a given time step.
5. Added a given edge constraint in prioritized.py that prohibits agent_1 from moving from its start cell (1, 2) to the neighboring cell (1,3) from time step 0 to time step 1.

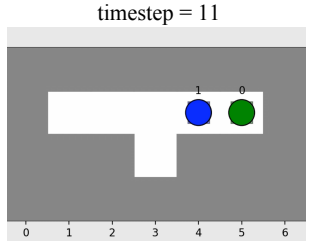
Task 1.4: Handling Goal Constraints

1. Added a given constraint in prioritized.py that prohibits agent_0 from being at its start cell (1, 5) at time step 10 for testing.
2. Have changed the goal test condition in the function a_star by getting the special time step constraint to obtain constrained_max_timestep = 10, so that it can handle the goal constraint by checking whether it reached goal cell after the max timestep.
3. Agent_0 is located at (1, 4) at time step 10 in my solution by using vertex constraint as follows:

timestep = 10



timestep = 11



terminal output:

```
jeongwoonsuh@Jeongwoonui-MacBookPro code_Task1 % python run_experiments.py --instance instances/exp1.txt --solver Prioritized
***Import an instance***
Start locations
@ @ @ @ @ @ @
@ 0 1 . . . @
@ @ @ . @ @ @
@ @ @ @ @ @ @

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

***Run Prioritized***
('Agent[0]', 'path = ', [(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 5), (1, 5), (1, 5), (1, 5), (1, 5), (1, 4), (1, 5)])
('Agent[1]', 'path = ', [(1, 2), (1, 3), (1, 4)])
```

Task 1.5: Designing Constraints

{ ‘agent’ : 0,
  ‘loc’ : [(1, 2), (1, 1)], [(1, 2), (1, 2)]
  ‘timestep’ : 2 }

{ ‘agent’ : 0,
  ‘loc’ : [(1, 3), (1, 2)], [(1, 3), (1, 3)], [(1, 3), (2, 3)]
  ‘timestep’ : 3 }

{ ‘agent’ : 0,
  ‘loc’ : [(1, 4), (1, 3)], [(1, 4), (1, 4)]
  ‘timestep’ : 4 }

{ ‘agent‘ : 1,
  ‘loc’ : [(1, 2), (1, 1)] , [(1, 2), (1, 2)]
  ‘timestep’ : 1 }

{ ‘agent‘ : 1,
  ‘loc’ : [(1, 3), (1, 4)] , [(1, 3), (1, 3)], [(1, 3), (1, 2)]
  ‘timestep’ : 2 }

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

{ ‘agent‘ : 1,
  ‘loc’ : [(1, 3), (1, 4)] , [(1, 3), (1, 3)], [(1, 3), (1, 2)]
  ‘timestep’ : 4 }

Agent 0 : Path = [((1, 1), 0), ((1, 2), 1), ((1, 3), 2), ((1, 4), 3), ((1, 5), 4)]
$\Rightarrow$ sum of costs : 4

Agent 1 : Path = [((1, 2), 0), ((1, 3), 1), ((2, 3), 2), ((1, 3), 3), ((1, 4), 4)]
$\Rightarrow$ sum of costs : 4
$\therefore$ Total sum of costs: 8

Task 2 : Implementing Prioritized Planning

Task 2.1: Adding Vertex Constraints

1. Added the constraint for vertex constraints in the function find_solution in prioritized.py which is two loops, to iterate over the path of the current agent and to add vertex constraints for all future agents.
2. There still exists a collision because both agents move to the cell of the other agent at the same time step.

Task 2.2: Adding Edge Constraints

1. Added the constraint for edge constraints in the function find_solution in prioritized.py by the condition for the path length greater than 0, and assigning the previous location to the path[path -1] and the next time step to path. Then, the constraint for the location is [current location, previous location] to avoid any of the collision.
2. Agents can still move on top of other agents that have already reached their goal locations.

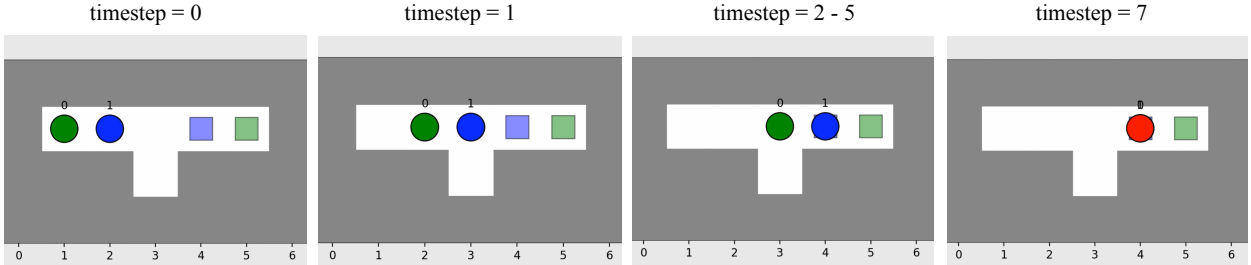
Task 2.3: Adding Additional Constraints

1. Added the additional constraints that apply not only to the time step when agents reach their goal locations but also to all future time step in the function find_solution in prioritized.py by using double for loop, and assigning the last location as path[-1] and the last timestep as the length of the path minus 1, to get the maximum cell.
2. For example, agent_0 reached its goal cell at time step 3, but agent_1 will reach to the goal cell at time step 7, then agent_0 will be waiting. Nonetheless, we could not possibly obtain the time step 7, so we just estimate this value with the prior knowledge, and that's how to obtain the number of a valid cell.

Task 2.4: Addressing Failures

1. Rerun the experiment on instance exp2_3.txt.
2. <u>Output</u> as follows:

timestep = 0 | timestep = 1 | timestep = 2 - 5 | timestep = 7



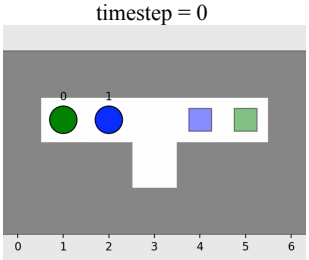
Terminal output:

```
jeongwoonsuh@Jeongwoonui-MacBookPro code_Task2 % python run_experiments.py --instance instances/exp2_3.txt --solver Prioritized
***Import an instance***
Start locations
@ @ @ @ @ @
@ 1 0 . . . @
@ @ @ . @ @
@ @ @ @ @ @

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

***Run Prioritized***
('Agent[0]', 'path = ', [(1, 2), (1, 3), (1, 4)])
('Agent[1]', 'path = ', [(1, 1), (1, 2), (1, 3), (1, 3), (1, 3), (1, 3), (1, 4), (1, 5)])

 Found a solution!

CPU time (s):    0.00
Sum of costs:    9
[[(1, 2), (1, 3), (1, 4)], [(1, 1), (1, 2), (1, 3), (1, 3), (1, 3), (1, 3), (1, 4), (1, 5)]]
***Test paths on a simulation***
COLLISION! (agent-agent) (0, 1) at time 7
COLLISION! (agent-agent) (0, 1) at time 7
COLLISION! (agent-agent) (0, 1) at time 7
COLLISION! (agent-agent) (0, 1) at time 7
COLLISION! (agent-agent) (0, 1) at time 7
COLLISION! (agent-agent) (0, 1) at time 7
COLLISION! (agent-agent) (0, 1) at time 7
COLLISION! (agent-agent) (0, 1) at time 7
COLLISION! (agent-agent) (0, 1) at time 7
COLLISION! (agent-agent) (0, 1) at time 7
COLLISION! (agent-agent) (0, 1) at time 7
```

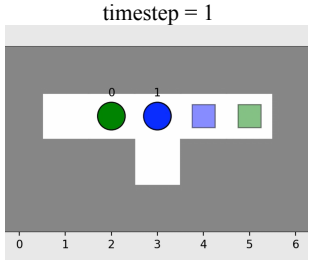
3. After running the experiment on instance exp2_3.txt, my solver did not terminate properly and not reported "no solution", but had collision at a time step 7 with the sum of costs 9.

4. Changed the code to address this issue in the function find_solution in prioritized.py by reordering the agents' priority using the heuristic costs.
5. <u>New output</u> as follows:

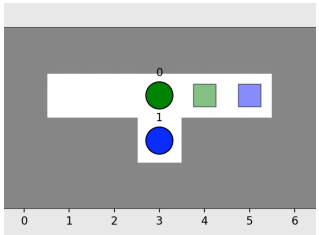
timestep = 0



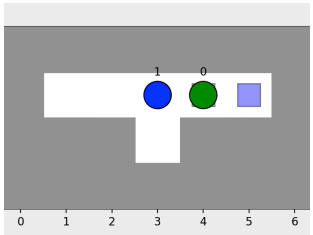
timestep = 1



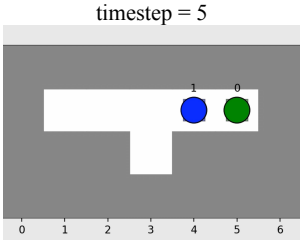
timestep = 2



timestep = 3



timestep = 5



Terminal output:

```
jeongwoonsuh@Jeongwoonui-MacBookPro code_Task2 % python run_experiments.py --instance instances/exp2_3.txt --solver Prioritized
***Import an instance***
Start locations
@ @ @ @ @ @ @
@ 1 0 . . . @
@ @ @ . @ @ @
@ @ @ @ @ @ @

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

***Run Prioritized***
((1, 2), 2)
((1, 3), 1)
((1, 4), 0)
((1, 5), 1)
((2, 3), 2)
((1, 1), 3)
((1, 2), 3)
((1, 3), 2)
((1, 4), 1)
((1, 5), 0)
((2, 3), 3)
((1, 1), 4)
('Agent[1]', 'path = ', [(1, 1), (1, 2), (1, 3), (1, 4), (1, 5)])
('Agent[0]', 'path = ', [(1, 2), (1, 3), (2, 3), (1, 3), (1, 4), (1, 4)])

 Found a solution!

CPU time (s):    0.00
Sum of costs:    9
[[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5)], [(1, 2), (1, 3), (2, 3), (1, 3), (1, 4), (1, 4)]]
***Test paths on a simulation***
jeongwoonsuh@Jeongwoonui-MacBookPro code_Task2 %
```

Task 2.5: Showing that Prioritized Planning is Incomplete and Suboptimal

1. After Task 2.3, I have designed a MAPF instance for which prioritized planning does not find an (optimal or suboptimal) collision-free solution for a given ordering of the agents:

For Agent_0:

{ 'agent' : 0,
  'loc' : [(1, 1)]
  'timestep' : 0 }

{ 'agent' : 0,
  'loc' : [(1, 2)]
  'timestep' : 1 }

{ 'agent' : 0,
  'loc' : [(1, 3)]
  'timestep' : 2 }

{ 'agent' : 0,
  'loc' : [(1, 4)]
  'timestep' : 3 }

{ 'agent' : 0,
  'loc' : [(1, 4)]
  'timestep' : 4 }

For Agent_1

{ 'agent' : 1,
  'loc' : [(1, 2)]
  'timestep' : 0 }

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

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

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

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

Task 3: Implementing Conflict-Based Search (CBS)

Task 3.1: Detecting Collisions

1. Implemented the function detect_collision(path1, path2) in cbs.py to return the list of the union of vertex and edge constraint, by adding the code to detect the vertex collision, to detect the edge collision and then converted to set due to make both constraints to not consider their order, and finally find the union of both. Also, I have used the function get_location(path, t) to obtain the cell of an agent at time step t.
2. Implemented the function detect_collisions(paths) in cbs.py to return a list of the first collisions between all pairs of agents, by creating empty list named ‘collisions’, checking for collision by the function ‘combination’ of 2 paths, and using double for loop to check all of the collisions of all pairs of the agent.
3. Changed the testing code to print the collision for testing in the function find_solution in cbs.py.
4. There does not exists instance exp3_1.txt in the provided instance folder, so I have used the exp2_1.txt, exp2_2.txt, exp2_3.txt to test the output.

Task 3.2: Converting Collisions to Constraints

1. Implemented the function standard_splitting in cbs.py to return a list of two constraints to resolve the given collision for both checking vertex and edge constraint. I have assigned agent_0, agent_1, loc, and timestep as collision, and created a empty table as constraints to return after checking vertex constraints where next location is None, and edge constraint where next location exists.
2. There does not exists instance exp3_1.txt in the provided instance folder, so I have used the exp2_1.txt, exp2_2.txt, exp2_3.txt to test the output.

Task 3.3: Implementing the High Level Search

1. Added code in cbs.py that finalized the high-level search of CBS in the function find_solution:
   A. If open_list is not empty, then
      a. Get the next node from the open_list by using self.pop_node( )
      b. If this node has no collision, then return solution
      c. Otherwise, choose the first collision and convert to a list of constraints using the function standard_splitting from Task 3.2, and add a new child node to the open_list for each constraint.
2. Added the print statements that list the expanded nodes for testing.
3. There does not exists instance exp3_1.txt in the provided instance folder, so I have used the exp2_3.txt to test the output, and since the transcript was too long, I have truncated with the reasonable length.

Terminal output:

```
jeongwoonsuh@Jeongwoonui-MacBookPro code_Task3_4 % python run_experiments.py --instance instances/exp2_3.txt --solver CBS
***Import an instance***
Start locations
@ @ @ @ @ @ @
@ 1 0 . . . @
@ @ @ . @ @ @
@ @ @ @ @ @ @

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

***Run CBS***
Generate node 0
[0, 1, (1, None), 3]
[0, 1, (1, None), 4]
[0, 1, (1, None), 1]
[0, 1, (1, None), 2]
[0, 1, (1, None), 0]
[0, 1, (1, 1), 1]
[0, 1, (1, 1), 4]
[0, 1, (1, 1), 2]
[0, 1, (1, 1), 3]
[{'loc': [1, None], 'agent': 0, 'timestep': 3}, {'loc': [1, None], 'agent': 1, 'timestep': 3}]
[{'loc': [1, None], 'agent': 0, 'timestep': 4}, {'loc': [1, None], 'agent': 1, 'timestep': 4}]
[{'loc': [1, None], 'agent': 0, 'timestep': 1}, {'loc': [1, None], 'agent': 1, 'timestep': 1}]
[{'loc': [1, None], 'agent': 0, 'timestep': 2}, {'loc': [1, None], 'agent': 1, 'timestep': 2}]
[{'loc': [1, None], 'agent': 0, 'timestep': 0}, {'loc': [1, None], 'agent': 1, 'timestep': 0}]
[{'loc': [1, 1], 'agent': 0, 'timestep': 1}, {'loc': [1, 1], 'agent': 1, 'timestep': 1}]
[{'loc': [1, 1], 'agent': 0, 'timestep': 4}, {'loc': [1, 1], 'agent': 1, 'timestep': 4}]
[{'loc': [1, 1], 'agent': 0, 'timestep': 2}, {'loc': [1, 1], 'agent': 1, 'timestep': 2}]
[{'loc': [1, 1], 'agent': 0, 'timestep': 3}, {'loc': [1, 1], 'agent': 1, 'timestep': 3}]
Expand node 0
Generate node 1
Generate node 2
Expand node 2
Generate node 3
Generate node 4
Expand node 4
Generate node 5
Generate node 6
Expand node 6
Generate node 7
Generate node 8
Expand node 8
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
Generate node 17
Generate node 18
Expand node 18
```

Task 3.4: Implementing the High Level Search

1. Tested the implementation by the instances ‘python run_experiments.py --instance “instances/test_*” --solver CBS --batch
2. The batch command created an output file, result.csv, which is empty since it has the key error which I could not solve.

Terminal output:

```
jeongwoonsuh@Jeongwoonui-MacBookPro code_Task3_4 % python run_experiments.py --instance "instances/test_*" --solver CBS --batch
***Import an instance***
Start locations
3 . @ . . . . .
4 0 . @ . . . .
. . @ . . . . .
. . . . . . . .
. . . 2 . . . @
. . . . . @ . .
. . . . . . @ .
. . . . . . 1 .

Goal locations
1 . @ . . . . .
. 2 . @ . . . .
. . @ . . . . .
0 . . . . . . .
. . . . . . . @
. . . . 3 @ 4 .
. . . . . . @ .
. . . . . . . .

***Run CBS***
Traceback (most recent call last):
  File "run_experiments.py", line 97, in <module>
    paths = cbs.find_solution(args.disjoint)
  File "/Users/jeongwoonsuh/Desktop/code_final/code_Task3_4/cbs.py", line 302, in find_solution
    i, root['constraints'])
  File "/Users/jeongwoonsuh/Desktop/code_final/code_Task3_4/single_agent_planner.py", line 129, in a_star
    h_value = h_values[start_loc]
KeyError: (1, 1)
jeongwoonsuh@Jeongwoonui-MacBookPro code_Task3_4 %
```

Task 4: Implementing CBS with Disjoint Splitting

Task 4.1: Supporting Positive Constraints

1. Added the code to single_agent_planner.py to handle positive vertex and edge constraints by adding the key ‘positive’ in the function a_star to root and child.

Task 4.2: Converting Collisions to Constraints

1. Implemented the function disjoint_splitting in cbs.py by returning a list of two constraints to resolve the given collisions using both vertex and edge constraint by choosing one of the two colliding agents randomly using ‘random.randint(0, 1)’. I have considered both case where next location is None, and next location exists for both agent_0 and agent_1.
2. To create a positive constraint, I have set the item positive in Python dictionary of the constraint to True in the function disjoint_splitting in cbs.py.

Task 4.3: Adjusting the High Level Search

1. Updated the code in cbs.py to adjust the high-level search of CBS in the function find_solution to the new constraints. I have computed a list of agent ids of agents that violate a given positive constraint with the helper function that I have also created, paths_violate_constraint.
2. In the function find_solution in cbs.py, the assigned constraints is updated to use the disjoint_splitting from Task 4.2.
3. The code was updated by computing a list of agents’ id of agents that violate a given positive constraint in the function find_solution in cbs.py using the function paths_violate_constraint(constraint, paths).
4. The function paths_violate_constraint(constraint, paths) returns the agents’ id by getting the previous and current location depending on the case where the next agent is available or not (vertex collision or edge collision, respectively).
5. Knowing that CBS with disjoint splitting should expand about 8 instead of 11 nodes,

Terminal output:

```
jeongwoonsuh@Jeongwoonui-MacBookPro code_Task3_4 % python run_experiments.py --instance instances/exp4.txt --solver CBS
***Import an instance***
Start locations
@ @ @ @ @ @ @
@ . 1 . . . @
@ 0 . . . . @
@ . . . . . @
@ . . . . . @
@ @ @ @ @ @ @
@ @ @ @ @ @ @

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

***Run CBS***
Traceback (most recent call last):
  File "run_experiments.py", line 97, in <module>
    paths = cbs.find_solution(args.disjoint)
  File "/Users/jeongwoonsuh/Desktop/code_final/code_Task3_4/cbs.py", line 302, in find_solution
    i, root['constraints'])
  File "/Users/jeongwoonsuh/Desktop/code_final/code_Task3_4/single_agent_planner.py", line 129, in a_star
    h_value = h_values[start_loc]
KeyError: (2, 1)
jeongwoonsuh@Jeongwoonui-MacBookPro code_Task3_4 %
```

I had an key error : (2, 1) to run CBS, thus I could not know how many nodes that my code expanded.