MULTI-AGENT PATH-FINDING

BHAKTI BHANUSHALI

Task 1

1.1 Searching the time-space domain

The modified A* algorithm tracks both location and time to support constraints that change over time, allowing the agent to avoid restricted areas or wait as needed.

Each node now includes a **time** attribute. This addition tracks when the agent reaches each cell, enabling time-based constraint checks.

When adding child nodes, I included all possible directions: going 1 step up, down, left and right and no move. This wait action allows the agent to pause if advancing would violate constraints.

The `closed_list` tracks visited `(location, time)` pairs, allowing the agent to revisit locations at different times without redundant expansions.

```
          child = {'loc': child_loc,

                        'g_val': curr['g_val'] + 1,

                        'h_val': h_values[child_loc],

                        'parent': curr,

                        'time': curr['time'] + 1 }
```

1.2 and 1.3 Handling vertex and edge constraints

To efficiently handle constraints in our A* search, I use a **constraint table** to pre-process and store prohibited actions for each time step. This table is a dictionary with separate entries for **vertex** and **edge** constraints:

1. **Constraint Table Structure**:
   - **Primary Keys**:
     - **`vertex`**: Holds constraints that prevent an agent from occupying specific cells at specific time steps.
     - **edge**: Holds constraints that prevent an agent from moving between specific cells at specific time steps.

```
constraint_table = {
            'edge': {0: []},
            'vertex': {1:[]},

      }
}
```

2. Each of these entries is further indexed by **time steps**, allowing quick lookups of prohibited actions based on the agent's location and movement at each step.
3. **Using the Table**:
   - Before each move, `is_constrained` checks if the current location (`curr_loc`), intended move (`next_loc`), and time step (`next_time`) match any constraint in the table.
   - **Vertex Constraint Check**: If `next_loc` at `next_time` is in the `vertex` constraints, `is_constrained` returns `True`, blocking the move.
   - **Edge Constraint Check**: Similarly, if the move from `curr_loc` to `next_loc` at `next_time` matches an edge constraint, the move is prohibited.

1.4 Goal Test Condition Adjustment:

- Typically, reaching the `goal_loc` would end the search. However, to account for **goal constraints** that prohibit the agent from occupying the goal at certain time steps, we refined this condition.
- **New Condition**: After checking if `curr['loc']` matches `goal_loc`, an additional loop verifies that no constraints prohibit the agent from occupying the goal location at or after the current time step.

Implementation Details:

- A constraint such as `{'loc': [(1,5)], 'timestep': 10}` prevents agent 0 from being at `(1,5)` at time step 10.
- In the modified condition, we check the `constraint_table["vertex"]` for any prohibited goal occupancy beyond the current time step. If such a constraint exists, `goal_found` is set to `False`, forcing the agent to wait or take an alternative path before reaching the goal.

Effect on Solution:

- With this change, the agent may reach its goal only when constraints allow it to remain there without violating any rules. If constraints prevent it from occupying the goal at a specific time (like time step 10), the agent waits or avoids the goal until the constraint is no longer in effect.

1.5 Designed Constraints

1. **Agent 1 Constraints**:
   - `{'agent': 1, 'loc': [(1,4)], 'timestep': 2}`: Prevents Agent 1 from being at `(1,4)` at time step 2, prohibiting agent 1 from being at its goal at time 2 and not colliding with gent 0.
   - `{'agent': 1, 'loc': [(1,2)], 'timestep': 2}`: These constraints prevent Agent 1 from moving back to the previous cell and colliding with agent 0

2. Sum of costs: 8

Task 2

2.1 Adding Vertex Constraints

- **Implementation**:
  - For each cell (`position`) in the path of the current agent, I add a **vertex constraint** for future agents, using `{'loc': [position], 'timestep': index+1}`. This prevents future agents from occupying that cell at the specified time step.
  - The loop iterates over the current agent's path and creates these constraints based on the cell locations and their corresponding time steps.

2.2 Adding Edge Constraints

- **Implementation**:
  - For each cell transition (from `path[index]` to `position`), you add an **edge constraint** to block future agents from moving between these two cells at the same time. This constraint is represented as `{'loc': [position, path[index]], 'timestep': index+}`.
  - This ensures that agents do not swap positions, avoiding conflicts.

2.3 Final Goal Constraint

- **Goal Condition Adjustments**:
    a. In multi-agent pathfinding (MAPF), agents may face additional constraints at their goal positions, often when other agents are restricted from occupying the same cell at specific times.
    b. `"end"` constraints help ensure that an agent does not reach its goal too early or remain there when another agent has exclusive access at a specified time.
- **Making the 'end' Constraint:**
    a. The last cell of the prioritised agent is extracted and the constraint is made with 'end': True so that future agents do not go over the agent and cause collision.
    b. This is represented as `{'loc': [path[-1]], 'timestep': len(path), 'end': True }`.
- **Applying the "end" Constraint**:
    a. When constructing the constraint table, constraints with `"end": True` indicate that an agent's goal location must not be occupied by others. This constraint ensures that agents respect final position constraints related to timing and access by other agents.
    b. The code evaluates the `next_time` (representing the time when the next move is attempted) to check if it meets or exceeds any time constraint specified in `"end"`.
- **Integration in `is_constrained`**

In `is_constrained`, the `"end"` constraint is checked as follows:

```
for time in constraint_table['end'].keys():

    if(time <= next_time and constraint_table['end'][time] ==
[next_loc]):

        return True
```

- **Logic**: This line iterates over each time step in `constraint_table['end']`. For each time step:
    - For each constraint, `time <= next_time` checks if the time step of the constraint is less than or equal to `next_time`, ensuring the agent is blocked from returning to the goal from that time step onward.
    - If `constraint_table['end'][time] == [next_loc]`, it confirms that the `next_loc` matches the restricted cell (the goal location), preventing the agent from re-entering it.
    - agent will not be allowed to complete its path at that location and time.
    - If both conditions are met, `is_constrained` returns `True`, indicating a violation of the goal constraint, which prevents the move.

Task 2.4 Issue: Infinite Loop Due to Lack of Path Length Bound

When testing the MAPF instance `exp2_3.txt`, the algorithm encountered an infinite loop. This occurred because an agent with lower priority could not find a feasible path that didn't interfere with higher-priority agents. Without an upper bound on the path length, the lower-priority agent repeatedly attempted to find a solution, cycling back and forth without progress.

To prevent this, I introduced an upper bound for each agent's path length. This bound ensures that agents will terminate the search if no solution is possible within a reasonable number of steps. The upper bound is calculated as follows:

```
upperbound = len(self.my_map) * len(self.my_map[0])
```

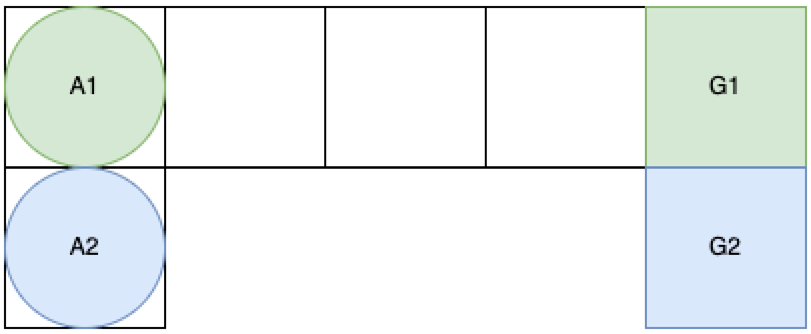
```
upperbound += len(path)
```

Explanation of the Upper Bound Calculation

1. **Map Size Factor**: `len(self.my_map) * len(self.my_map[0])` represents the total number of cells in the environment, providing a baseline maximum path length.
2. **Higher-Priority Agent Path Length**: By adding the path length of higher-priority agents (`len(path)`), the bound accounts for potential delays caused by these agents occupying crucial cells.

Task 2.5 Making Instances

1. Here Prioritised planning fails to find a collision free solution.

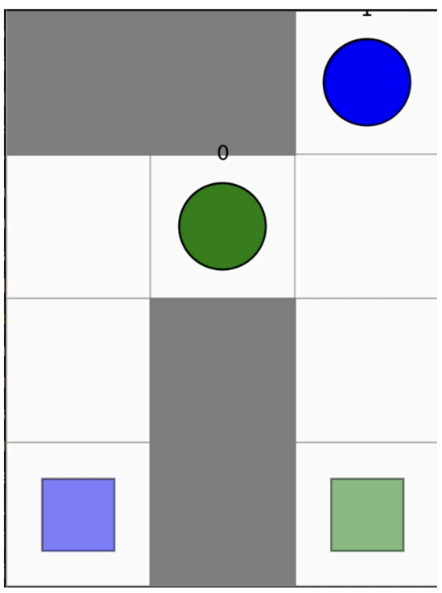


This graph is in `/custominstances2/noSoln.txt`
To test it on prioritised run:

```
python run_experiments.py --instance custominstances2/noSoln.txt --solver Prioritized
```

2. Did not attempt
3. (Bonus) If agent 1 is given priority no solution can be found but if we reverse the priorities and give 0 priority then it finds a solution
   Blue - agent 1, Green -Agent 0
   This graph is in `/custominstances2/3Sol.txt`. To test it on prioritised run:

```
python run_experiments.py --instance custominstances2/3Sol.txt --solver Prioritized
```



Task 3.1 Detecting Collisions in CBS

1. **Vertex Collision**:
    - A vertex collision occurs when both agents occupy the same cell at the same time step.
    - In the code, this is detected by checking if `get_location(shorter_list, time) == pos`, where pos is the position of the agent in `with the longer path`, If they match, a vertex collision is detected, and the function returns a dictionary detailing the collision (`'vertex': True`).
2. **Edge Collision**:
    - An edge collision occurs when two agents swap positions in consecutive time steps (i.e., each agent moves to the other's current location simultaneously).
    - This is checked by verifying if `get_location(shorter_list, time+1) == pos` and `get_location(shorter_list, time) == get_location(longer_list, time+1)`. If true, an edge collision is detected, and the function returns a dictionary detailing the collision with (`'vertex': False`).

```
    for time, pos in enumerate(longer_list):
        #vertex collision
        if(get_location(shorter_list,time) == pos):
                return {'a1': agent1, 'a2': agent2, 'loc': [pos],
'timestep':time, 'vertex': True}
        #edge collision
        if(get_location(shorter_list,time+1) == pos and
get_location(shorter_list,time) == get_location(longer_list,time+1)):
            # print("found edge collision",
pos,get_location(path2,time+1), get_location(path1,time-1), time, i )
            return {'a1': agent1, 'a2': agent2, 'loc':
[pos,get_location(longer_list,time+1)], 'timestep':time+1, 'vertex':
False}
```

Explanation of Implementation

- **Longest Path Handling**: The function compares `path1` and `path2` lengths, setting `longer_list` to the longer path and `shorter_list` to the shorter path. This ensures that `time` steps can iterate fully over the longer path.
- **Collision Output**: The function returns the details of the first detected collision and:
    - `a1` and `a2`: IDs of the agents involved.
    - `loc`: The locations involved in the collision.

- `timestep`: The time step of the collision.
- `vertex`: A Boolean indicating whether it's a vertex (`True`) or edge collision

Task 3.2 Converting Collisions into Constraints

The `standard_splitting` function takes a detected collision between two agents and generates constraints to prevent that collision in future search paths. The constraints are split based on the type of collision: **vertex** or **edge**.

1. **Vertex Collision**:
   - When two agents occupy the same cell at the same time step, it's a vertex collision.
   - The function creates two constraints:
     - **Constraint 1**: Prevents the first agent from occupying the colliding cell at the specified time.
     - **Constraint 2**: Prevents the second agent from occupying the same cell at the same time.
   - These constraints ensure that the agents cannot both occupy the same cell simultaneously.

```
    cons1 = {"agent": collision['a1'], 'loc': collision['loc'],
'timestep': collision['timestep'],"vertex":True,"end":False}
        cons2 = {"agent": collision['a2'], 'loc': collision['loc'],
'timestep': collision['timestep'],"vertex":True,"end":False}
        res.append(cons1)
        res.append(cons2)
```

2. **Edge Collision**:

   - When two agents swap cells at the same time, it's an edge collision.
   - The function creates two edge constraints to prevent this swap:
     - **Constraint 1**: Prevents the first agent from moving along the colliding edge (from its current location to the other agent's location).
     - **Constraint 2**: Prevents the second agent from moving along the reverse of the colliding edge (from its location to the first agent's location).
   - This avoids simultaneous cell swapping, which could cause further conflicts.

```
        cons1 = {"agent": collision['a1'], 'loc': collision['loc'],
'timestep': collision['timestep'],"vertex":False,"end":False}
        cons2 = {"agent": collision['a2'], 'loc':
[collision['loc'][1],collision['loc'][0]], 'timestep':
collision['timestep'],"vertex":False,"end":False}
        res.append(cons1)
        res.append(cons2)
```

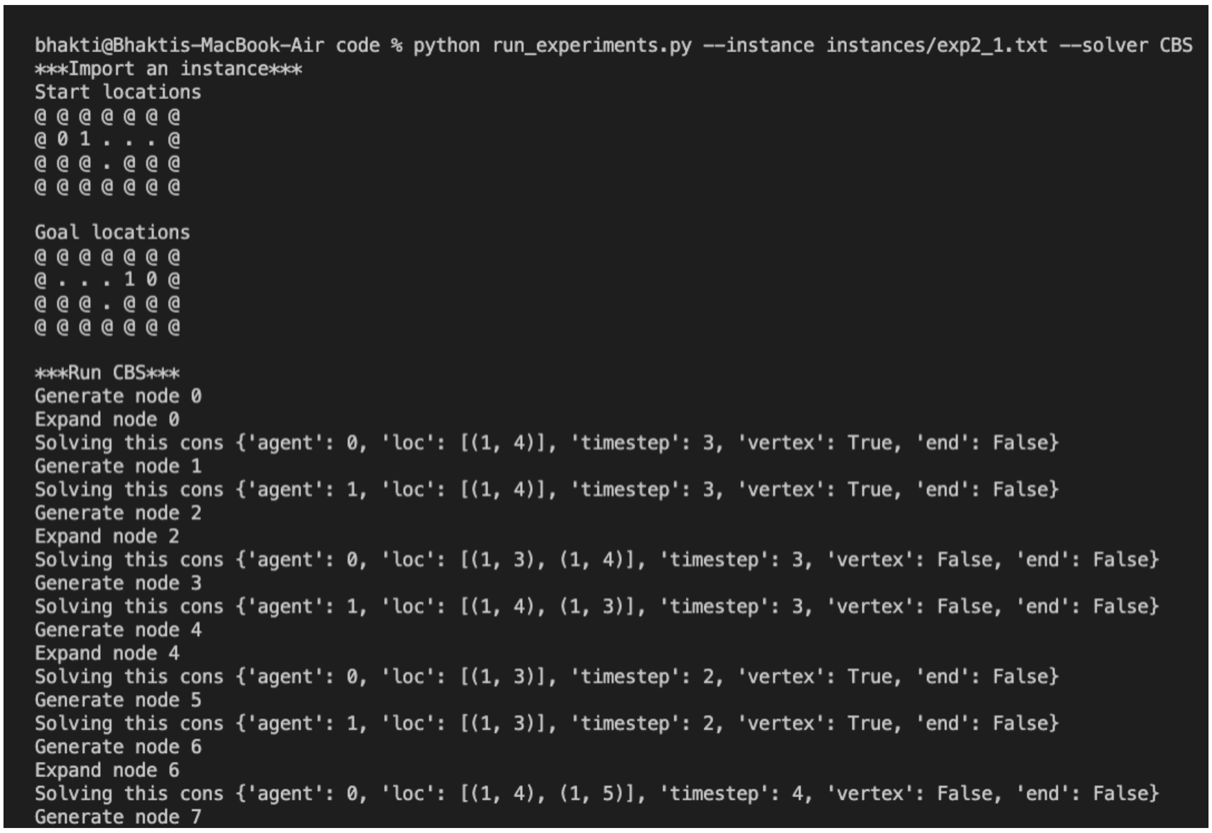
2. **Returning the Constraints**:
   - Both constraints are added to a list and returned as the result of `standard_splitting`.

- This list is then used by CBS to create new nodes that avoid the current collision.

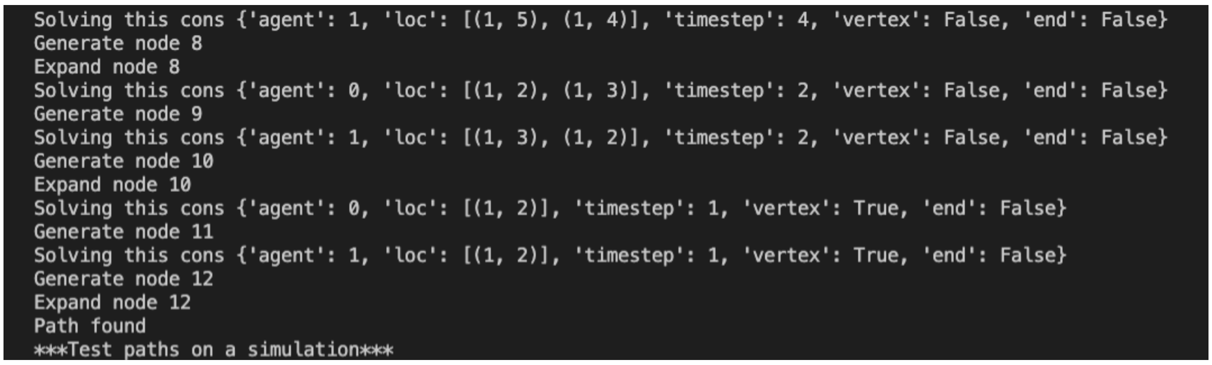
3.3 Task 3.3 High-Level CBS Search in `find_solution`

The `find_solution` function implements Conflict-Based Search (CBS) to find collision-free paths for multiple agents. Here's how the main components work:

1. **Root Node Initialization**:
   - The algorithm begins by creating a root node with:
     - An empty constraint list,
     - Initial paths for all agents (computed using `a_star`), and
     - A list of detected collisions among these paths.
   - The `root` node's `cost` is set to the sum of path costs across agents.
2. **High-Level Search Loop**:
   - The search continues until there are no nodes left in the open list (maintained with `push_node` and `pop_node`).
   - Each node represents a potential solution, and nodes are expanded by resolving the first detected collision.
3. **Collision Resolution**:
   - If a node has no collisions, it's a solution, and the function returns the paths.
   - If collisions are present, the first collision is chosen and split into two constraints using `standard_splitting`. These constraints prevent each agent from occupying the conflicting cell or traversing the conflicting edge simultaneously.
4. **Generating Child Nodes**:
   - For each constraint, a child node is created by:
     - Copying the current node's constraints and paths,
     - Adding the new constraint to the list,
     - Recomputing the path for the agent affected by the constraint.
   - After finding the new path, the algorithm updates the child node's list of collisions and total cost.
5. **Pushing Nodes to the Open List**:
   - If a valid path is found for the agent under the new constraint, the child node is added to the open list for further exploration.
   - If no valid path can be found, the node is discarded, effectively pruning infeasible solutions.
6. **Termination**:
   - If the open list is exhausted without finding a collision-free solution, the function raises an exception, indicating that no solution exists.



```
bhakti@Bhaktis-MacBook-Air code % python run_experiments.py --instance instances/exp2_1.txt --solver CBS
***Import an instance***
Start locations
@ @ @ @ @ @ @
@ 0 1 . . . @
@ @ @ . @ @ @
@ @ @ @ @ @ @

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

***Run CBS***
Generate node 0
Expand node 0
Solving this cons {'agent': 0, 'loc': [(1, 4)], 'timestep': 3, 'vertex': True, 'end': False}
Generate node 1
Solving this cons {'agent': 1, 'loc': [(1, 4)], 'timestep': 3, 'vertex': True, 'end': False}
Generate node 2
Expand node 2
Solving this cons {'agent': 0, 'loc': [(1, 3), (1, 4)], 'timestep': 3, 'vertex': False, 'end': False}
Generate node 3
Solving this cons {'agent': 1, 'loc': [(1, 4), (1, 3)], 'timestep': 3, 'vertex': False, 'end': False}
Generate node 4
Expand node 4
Solving this cons {'agent': 0, 'loc': [(1, 3)], 'timestep': 2, 'vertex': True, 'end': False}
Generate node 5
Solving this cons {'agent': 1, 'loc': [(1, 3)], 'timestep': 2, 'vertex': True, 'end': False}
Generate node 6
Expand node 6
Solving this cons {'agent': 0, 'loc': [(1, 4), (1, 5)], 'timestep': 4, 'vertex': False, 'end': False}
Generate node 7
```



```
Solving this cons {'agent': 1, 'loc': [(1, 5), (1, 4)], 'timestep': 4, 'vertex': False, 'end': False}
Generate node 8
Expand node 8
Solving this cons {'agent': 0, 'loc': [(1, 2), (1, 3)], 'timestep': 2, 'vertex': False, 'end': False}
Generate node 9
Solving this cons {'agent': 1, 'loc': [(1, 3), (1, 2)], 'timestep': 2, 'vertex': False, 'end': False}
Generate node 10
Expand node 10
Solving this cons {'agent': 0, 'loc': [(1, 2)], 'timestep': 1, 'vertex': True, 'end': False}
Generate node 11
Solving this cons {'agent': 1, 'loc': [(1, 2)], 'timestep': 1, 'vertex': True, 'end': False}
Generate node 12
Expand node 12
Path found
***Test paths on a simulation***
```

Task 3.4

Results.csv matches with min-sum-of-cost.csv.

Task 4.1 Supporting Positive Constraints

1. `build_constraint_table` with Positive Constraints

- **Constraint Type Classification**:
  - The function categorizes each constraint as either `"positive"` or `"negative"` using the `positive` key in each constraint dictionary (`"positive": True` for positive constraints, `"positive": False` for negative constraints).
  - Based on the `positive` flag, the function assigns the constraint to either the `"positive"` or `"negative"` section of the `constraint_table`.
- **Constraint Structure**:
  - For each constraint:
    - If it is a **vertex constraint** (specifying a cell the agent must or must not be at a given time), it's added to the `"vertex"` dictionary, indexed by time step.
    - If it is an **edge constraint** (specifying a move between two cells), it's added to the `"edge"` dictionary, also indexed by time step.
    -
```
constraint_table = {
    "negative": {
            'edge': {},
            'vertex': {},
            'end': {}
     },
     "positive": {
            'edge': {},
            'vertex': {}
     }
}
```
  - This structure allows both positive and negative constraints to be easily referenced at each time step during the search.

2. `is_constrained` with Positive Constraints

The `is_constrained` function checks whether a specific move (from `curr_loc` to `next_loc` at `next_time`) respects all constraints. Here's how positive constraints are handled:

- **Positive Vertex Constraint**:
  - If there is a positive vertex constraint for `next_time`, it specifies a cell where the agent **must** be.
  - The condition `if (next_loc not in constraint_table["positive"]['vertex'][next_time])` verifies that `next_loc` matches the required position. If not, `True` is returned,

indicating a violation.

- **Positive Edge Constraint**:
    - If there is a positive edge constraint for `next_time`, it specifies a required transition between two cells.
    - The function checks that the move from `curr_loc` to `next_loc` matches the required edge in `constraint_table["positive"]['edge'][next_time]`. If it doesn't match, `True` is returned, indicating the move violates the positive constraint.
- **Negative Constraints Check**:
    - After checking positive constraints, `is_constrained` verifies that the move does not violate any negative vertex or edge constraints, following a similar structure to block prohibited moves or locations.

Task 4.2 Disjoint Splitting for Collision Resolution

The `disjoint_splitting` function resolves a collision between two agents by generating constraints for a **single agent** chosen randomly. This approach creates a **positive constraint** for one agent to follow a specific path and a **negative constraint** to prevent that same agent from causing further conflicts by deviating from the new path.

Steps in `disjoint_splitting`

1. **Random Agent Selection**:
    - Using `random.randint(0,1)`, one of the colliding agents (`collision['a1']` or `collision['a2']`) is selected randomly to be the subject of the constraints.
    - The selected agent, `posAgent`, is the one who will be constrained with both positive and negative constraints.
2. **Creating Constraints Based on Collision Type**:
    - **Vertex Collision**:
        - A vertex collision occurs when two agents attempt to occupy the same cell at the same time.
        - The function generates:
            - **Positive Constraint (cons1)**: Requires `posAgent` to be at the specified location at the given time (`"positive": True`).
            - **Negative Constraint (cons2)**: Prohibits `posAgent` from being at the same location at the same time (`"positive": False`).
        - This setup pushes the selected agent to occupy the specified cell while preventing it from violating the collision constraints.
    - **Edge Collision**:
        - An edge collision occurs when two agents swap locations at the same time.
        - The function creates:
            - **Positive Constraint (cons1)**: Requires `posAgent` to traverse the specified edge (move from `curr_loc` to `next_loc`) at the collision time.
            - **Negative Constraint (cons2)**: Prevents `posAgent` from making the reverse move at the same time, avoiding a potential collision.

        - By enforcing one edge direction while blocking the reverse direction, this approach ensures the selected agent's path is clear of conflicts.

3. **Return Constraints**:
    - Both constraints (`cons1` and `cons2`) are returned as a list. CBS then uses these constraints to create new search nodes, each representing a unique resolution path for the collision.

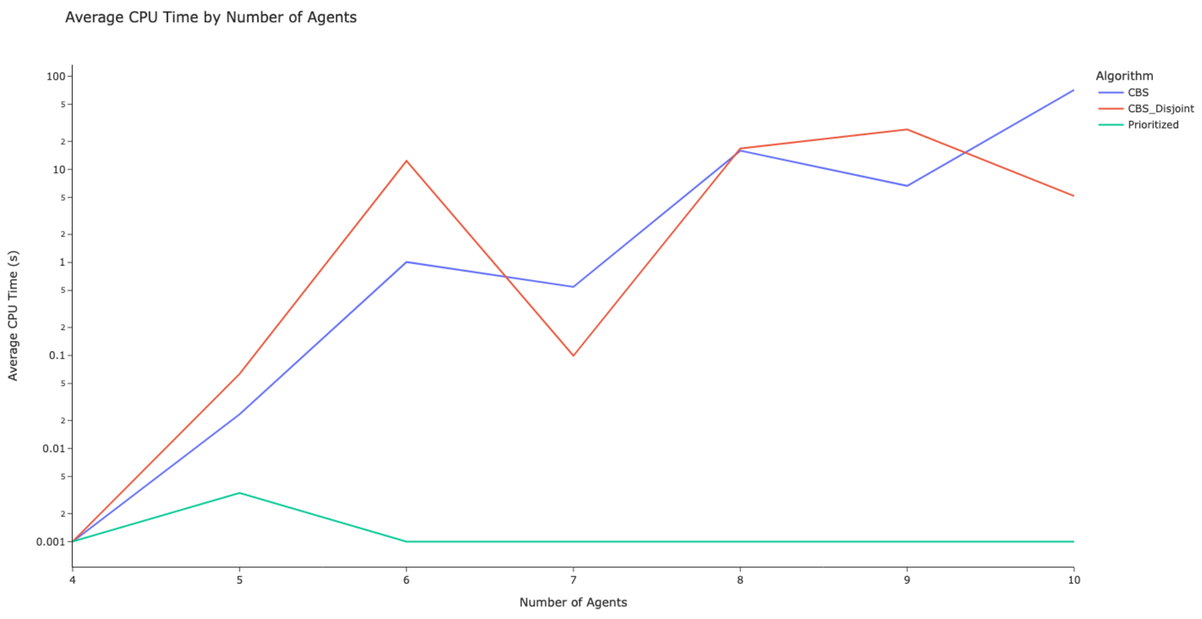
```
   randomAgent = random.randint(0,1)
   if(randomAgent):
       posAgent = collision['a1']
   else:
       posAgent = collision['a2']

   if(collision['vertex']):
       cons1 = {"agent": posAgent, 'loc': collision['loc'], 'timestep':
collision['timestep'],"vertex":True,"end":False, "positive": True}
       cons2 = {"agent": posAgent, 'loc': collision['loc'], 'timestep':
collision['timestep'],"vertex":True,"end":False,  "positive": False}
       res.append(cons1)
       res.append(cons2)

   else:
       # {'a1': i, 'a2': j, 'loc': [pos,get_location(path2,time+1)],
'timestamp':time+1, 'vertex': False}
       cons1 = {"agent": posAgent, 'loc': collision['loc'], 'timestep':
collision['timestep'],"vertex":False,"end":False, "positive": True}
       cons2 = {"agent": posAgent, 'loc':
[collision['loc'][1],collision['loc'][0]], 'timestep':
collision['timestep'],"vertex":False,"end":False, "positive": False}
       res.append(cons1)
       res.append(cons2)
```

Task 4.3 Handling Disjoint Splitting and Positive Constraints

1. **Collision Detection and Constraint Creation**:
    - Each node in the CBS tree maintains a list of detected collisions.
    - When a collision is found, the `disjoint_splitting` function is called, creating:
        - A **positive constraint** for one agent (chosen randomly) that enforces a specific position or edge transition.
        - A **negative constraint** for the same agent, prohibiting it from revisiting or reversing the move at that time.
2. **Positive Constraints and Affected Agents**:
    - In CBS with disjoint splitting, a positive constraint for one agent also implies negative constraints for all other agents.
    - The `paths_violate_constraint` function is used to identify which other agents' paths are affected by this positive constraint.
3. **Path Replanning for Violated Constraints**:
    - For nodes with a positive constraint, each affected agent must be replanned to avoid violating this constraint.
    - For each affected agent:
        - The algorithm generates a new path that respects the additional constraints.
        - If any affected agent cannot find a valid path, the node is discarded to reduce unnecessary branching.
4. **Node Creation and Expansion**:
    - Each constraint generates a child node with:
        - Updated constraints and paths,
        - Newly computed collisions and costs based on the added constraints.
    - Only nodes with valid paths for all affected agents are pushed onto the open list, optimizing the search by avoiding infeasible paths.

Expanded 3 nodes

Task 5
I benchmarked my algorithms by using instances from maze-32-32-4.map
I made individual map scenarios using ‘make_map.py’ which are then inside ‘custominstances/’. 250 instances were created but I only ran my benchmarking on the first 100.
The algos using ‘compare.py’.
These plots were created using ‘plot.py’



This plot illustrates the CPU time each algorithm takes (CBS, CBS_Disjoint, and Prioritized) as the number of agents increases.

CBS vs. CBS_Disjoint:

- Both CBS and CBS_Disjoint show an increase in CPU time as the number of agents grows, indicating that these algorithms become computationally more demanding with higher agent counts.
- CBS_Disjoint generally shows higher CPU time than CBS, possibly due to the additional constraints and the more complex collision resolution in the disjoint splitting approach. This method adds positive constraints that may require extra recomputation, making CBS_Disjoint slower in some cases. But for higher agents CBS Disjoint has a smaller CPU time, proving its runtime efficiency over CBS.

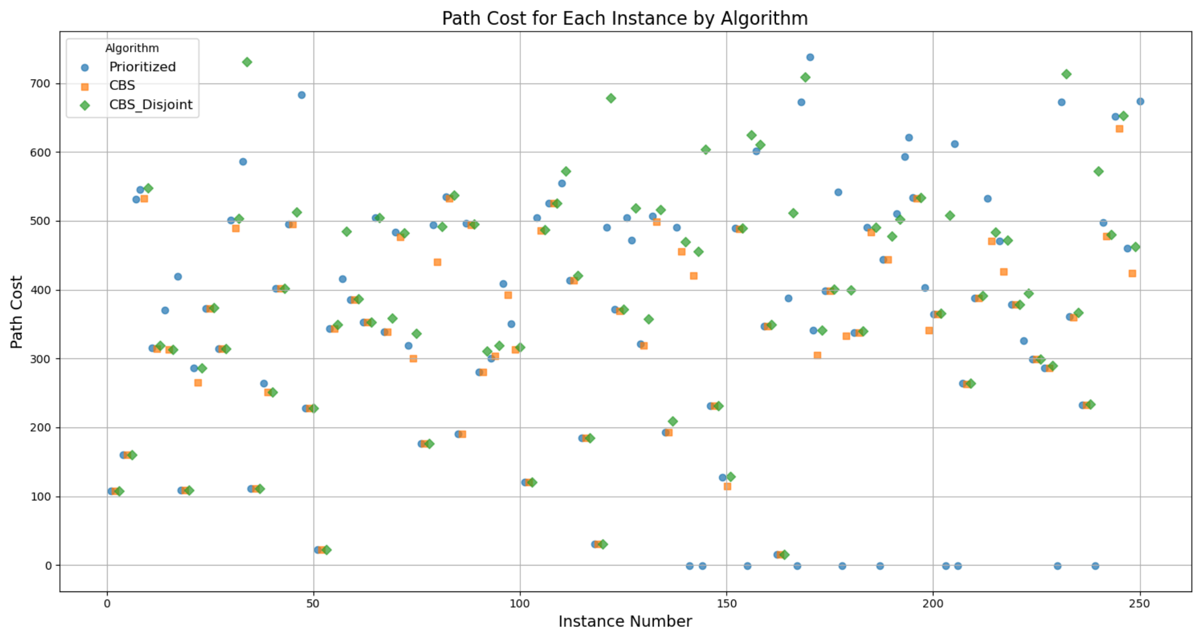
Prioritized Planning:

- The Prioritized algorithm has the lowest CPU time across all agent numbers, showing a minimal increase as the agent count rises. This is likely because it processes each agent independently in a prioritized order, avoiding the complex constraint-based conflict resolution that CBS and CBS_Disjoint require.

- This lower CPU time indicates that Prioritized Planning is computationally efficient but does not guarantee finding an optimal or even feasible solution in all cases. Its simplicity is advantageous for runtime but may sacrifice solution quality.

Even though prioritized planning seems to be the best choice for this dataset, I think that's because of the large size of the maze which resulted in fewer bottlenecks.

With fewer conflicts, the downside of the fixed priority order in prioritized planning is minimized. In CBS and CBS_Disjoint, there is additional overhead for handling complex collision constraints, making it a bit inefficient.



- Prioritized (blue circles) generally has path costs comparable to CBS and CBS_Disjoint, though there are instances where it deviates and yields both higher and lower path costs.
- CBS (orange squares) is often similar in path cost to CBS_Disjoint and Prioritized but sometimes achieves a lower cost, suggesting it can occasionally be more efficient.
- CBS_Disjoint (green diamonds) shows slightly more variability in some instances, as seen from instances where its costs are higher or lower compared to CBS, likely due to its unique constraint-handling mechanism.
- The points for each algorithm often overlap or cluster together, indicating similar performance on many instances. However, the spread on certain instances shows that the algorithms' performance diverges in specific cases.