University of Alberta
Computing Science Department
CMPUT 366 - Fall 2023

Assignment 2
Due date: November 10
13 marks

Search & Planning in AI (CMPUT 366)

## Submission Instructions

Submit your code on eClass as a zip file (the entire "starter" folder) and the answer to the question of the assignment as a pdf. The pdf must be submitted as a separate file so we can visualize it on eClass.

## Overview

In this assignment, you will implement CBS for solving multi-agent pathfinding problems on video game maps. We will consider a grid environment where the agents can take one of the following actions: up, down, left, right, or wait; each action costs 1.0. Each search problem is defined by a video game map, and a start location and a goal location for each agent. We will consider only vertex constraints, which means that two agents can swap positions in the map. The assignment includes a large number of maps from `movingai.com`, but you will use a single map, specified in `main.py`, in our experiments; feel free to explore other maps.

Most of the code you need is already implemented in the assignment package. In the next section, we detail some of the key functions you will use from the starter code. You can reimplement all these functions if you prefer, their use is not mandatory. The assignment must be implemented in Python, however.

You will use several data structures in this assignment, including lists, sets, dictionaries, and heaps. That is why you should read the Python tutorial available on eClass before attempting to implement CBS.

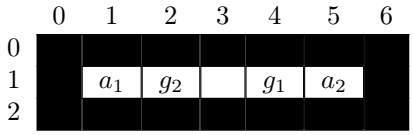
## Starter Code (0 Marks)

Similarly to Assignment 1, the starter code comes with a class implementing the map and another implementing the CBS states in the tree. In contrast with Assignment 1, in this assignment we also provide the code for A*, which is already adapted for CBS (you will be asked about this adaptation in a question below). We also provide the code for running the experiments (see `main.py` for details about the experiments).

The test cases include 12 problems with varied number of agents (from 10 to 50 agents). In addition to these instances, we also provide a simple problem that will help you debug your code. We recommend that you first test your CBS implementation on the easy problem before moving to the harder instances in `test-instances/instances.txt`. The easy instance uses `dao-map/test_map.map`, which is shown below.

Once CBS is implemented correctly, you will observe an output similar to the following for the easy problem.

```
Solution paths encountered for the easy test:
0 [[1, 1], [2, 1], [2, 1], [3, 1], [4, 1]]
1 [[5, 1], [4, 1], [3, 1], [2, 1]]
```




The output for the remaining problems will look like the following.

```
Correctly Solved:  1131 1131
Correctly Solved:  1393 1393
Correctly Solved:  1027 1027
Correctly Solved:  1436 1436
Correctly Solved:  1076 1076
Correctly Solved:  2451 2451
Correctly Solved:  2345 2345
Correctly Solved:  4284 4284
Correctly Solved:  3436 3436
Correctly Solved:  4457 4457
Correctly Solved:  5091 5091
Correctly Solved:  5587 5587
```

The output presents the optimal solution cost of each problem and the cost CBS finds.

**CBS State Implementation**

The core of the CBS algorithm will be implemented in the class `CBSState` in `algorithms.py`. This class implements the nodes in the CBS search tree. Each node carries the following information.

1. The set of agents in the problem, i.e., the initial and goal locations. In `CBSState`, the set of initial locations is given by the list `self._starts`, while the set of goal locations is given by `self._goals`. Note that `self._starts[i]` and `self._goals[i]` return the starting and goal location of the $i$-th agent in the problem. The number of agents in the problem is given by variable `self._k`.

2. The set of constraints of each agent. This is given by a dictionary of dictionaries, called `self._constraints`. `self._constraints` is indexed by the identifier of the agent. For example, `self._constraints[i]` accesses the dictionary storing the constraints of the $i$-th agent in the problem. If the $i$-th agent cannot be in state $(150, 75)$ in time steps 1 and 2, then `self._constraints[i][(150, 75)]` returns a set with the values of 1 and 2. Recall that in CBS, the constraints of the parent are passed to its children. So it is the structure `self._constraints` that needs to be copied from parent to children.

3. Since CBS runs A* for each of the agents, we store in each CBS state a reference to the problem's map. The map is a variable that we pass to A* while computing the cost of a CBS state.

4. Each CBS state also stores its cost, denoted as `self._cost`. We will use the sum of the cost of the paths of all agents as the objective function, so it is this sum that we will store in `self._cost`. It is `self._cost` that is used to implement the less-than operator of `CBSState`. This way, we know that the heap of the CBS search will be sorted by the cost of the CBS states.

5. One path for each agent in the dictionary `self._paths`, which is indexed by the agent identifier. For example, `self._paths[i]` returns the path of the i-th agent in the problem.

At this point, you should go look at the code starter to verify all these variables in the constructor of `CBSState` class. You should look for the method `def __init__(self, map, starts, goals)`, which receives the map and the lists with the starting and goal locations of all agents. All the other information stored in a CBS state will be computed by methods you will implement as part of this assignment.

# Implementing CBS State

You will implement the methods `compute_cost`, `is_solution`, and `successors` from `CBSState`. The starter code provides the implementation of `set_constraint`, which you will use in `successors`.

## Implement `compute_cost` (2 Marks)

The `compute_cost` method receives no parameter (in addition to Python's `self`) and it computes the value of `self._cost`, which should be the sum of the solution cost of the paths of each agent in the problem. In addition to the cost, this method also computes the solution path of each agent, which is a list of states determining the path of the agent. The path of each agent $i$ must be stored in `self._paths[i]`.

The starter code already provides an A* implementation that receives the map, as shown in the code below.

```
astar = AStar(self._map)
```

Here, `self._map` is the map stored in the CBS state. A* can be used to find the cost and the solution path for an agent and a set of constraints. This can be done as follows, for the $i$-th agent in the problem.

```
cost, path = astar.search(self._starts[i], self._goals[i], self._constraints[i])
```

The A* implementation returns the cost and the a solution path that respects the set of constraints given in `self._constraints[i]`. Since all test problems in the assignment have a solution, you do not need to implement the edge case where the problem has no solution. You can assume that A* will always return a valid solution path that satisfies the set of constraints.

Before implementing the next method, you should test your implementation of `compute_cost` on the easy problem provided in the starter code. Create a CBS state with no constraints and the set of start and goal locations defined in the easy problem, then call `compute_cost` for the state. Does it return the correct cost and the correct path for the two agents? Recall that the paths and cost it returns is equivalent to what one will observe in the root of the CBS tree.

## Implement `is_solution` (2 Marks)

This method needs to return true if the CBS state represents a solution, and false otherwise. In addition to the Boolean value, the method should also return a tuple with the state and the time step involved in a conflict that prevents the state from being a solution. For example, if the variable `state` is a state in the A* search (not the CBS search) that is involved in conflict, then the method will return the following.

```
return False, (state, state.get_g())
```

Note how the $g$-value of the states in the A* search represents the time step of the conflict. Since all actions cost 1.0, $g$-values and time steps represent the same quantity. In case the CBS state represents a solution, then the method can return the following.

```
return True, None
```

Here, `None` represents no value, which will be ignored by the code calling `is_solution`.

This method can be implemented with a set of tuples (state, time step). That way, you can iterate over all states $s$ with time steps $g(s)$ of the solution paths, while adding the tuples $(s, g(s))$ to the set. Once you find a tuple $(s, g(s))$ that is already in the set, you can stop and return false because you encountered a conflict.

Before implementing the next method, you should test your implementation of `is_solution` on the easy problem provided in the code starter. Create a CBS state with no constraints and run the method `compute_cost` followed by a call of `is_solution`. The latter should return false and a conflicting state.

## Implement `successors` (3 Marks)

The `successors` method should return a list with the two children of a CBS state that does not represent a solution. `successors` should find any conflicting state in the solution paths stored in the CBS state and the two agents involved in the conflict (recall from the lecture that there could be more than two agents involved in the conflict, but all you need is just two of them, as we are assuming a binary tree for CBS).

In the first child, we will constrain the first agent to not use the conflicting state in the conflicting time step; in the second child, we will constraint the second agent to not use the conflicting state in the conflicting time step. A new conflict can be added to the set of conflicts of the $i$-th agent with the line of code shown below, where `conflict_state` and `conflict_time` are the state and time step in the conflict and `c` is the variable name for the child state in the CBS tree. Recall that `is_solution` returns the state and time step of a conflict, so it might be a good idea to use it.

```
c.set_constraint(conflict_state, conflict_time, i)
```

Each child $c$ of a CBS state $s$ should contain the set of constraints of $s$ and a constraint given a conflict found in the solution paths of $s$. That is why, in addition to calling `set_constraint`, as explained above, we need to copy `self._constraints` from $s$ to $c$ before calling `set_constraint`. The code will be as follows.

```
c = CBSState(self._map, self._starts, self._goals)
c._constraints = copy.deepcopy(self._constraints)
c.set_constraint(conflict_state, conflict_time, i)
```

Here, `copy.deepcopy` produces a copy of the dictionary of dictionaries `self._constraints`.

Before moving on to the next method, you should test your implementation of `successors` on the easy problem in the starter code. Does successor return two CBS states with the correct set of constraints?

## Implement `search` (3 Marks)

The method `search` in class `CBS` performs the CBS search. It receives the root of the CBS tree, `start`, and performs a best-first search using the methods in `CBSClass`. The search should stop once a CBS state representing a solution is found. The method should return the paths of the agents and the solution cost. You should test the implementation of `search` in the easy problem before testing it on the harder problems.

## Question Answering (3 Marks)

What are the two main differences between the starter code's A* implementation (for use in CBS) and an A* implementation for single agent pathfinding? One sentence is enough to describe each difference.