COM S 476/576 Homework 1: Discrete Planning

Task 1 (General forward search algorithm) [10 points]: Implement the 3 variants of forward search algorithm: **breadth-first, depth-first, and A***. You can find the template for this task in `discrete_search.py` in the course repo. There are 3 important classes that we will use to specify a discrete planning problem (and you will have to implement them in later tasks).

- `StateSpace`: Let `X` be an instance of this class or its derived class. Then, given a state `x`, the statement

  ```
  x in X
  ```

  returns a boolean that indicates whether the state `x` is in the state space `X`. Additionally, given 2 states `x1` and `x2`, the statement

  ```
  X.get_distance_lower_bound(x1, x2)
  ```

  returns a float that represents the lower bound on the distance between the states `x1` and `x2`.

- `ActionSpace`: Let `U` be an instance of this class or its derived class. Then, given a state `x`, the statement

  ```
  U(x)
  ```

  returns the list of all the possible actions (i.e., the action space) for the state `x`.

- `StateTransition`: Let `f` be an instance of this class or its derived class. Then, given a state `x` and an action `u`, the statement

  ```
  f(x, u)
  ```

  returns the new state obtained by applying action `u` at state `x`.

Your task is to implement the function `fsearch(X, U, f, xI, XG, alg)` where

- `X` is an instance of the `StateSpace` class (or its derived class) that represents the state space.
- `U` is an instance of the `ActionSpace` class (or its derived class) that specifies the action space.
- `f` is an instance of the `StateTransition` class (or its derived class) that specifies the state transition function.
- `xI` is an initial state such that the statement `xI in X` returns `true`.

- `XG` is a list of states that specify the goal set. Note that **some states in `XG` may not be in `X` and `XG` may be empty**.
- `alg` is a string that specifies the discrete search algorithm to use (`"bfs"`, `"dfs"`, or `"astar"`).

This function should return a dictionary

```
{"visited": visited_states, "path": path}
```

where `visited_states` is the list of states visited during the search and `path` is a path from `xI` to a state in `XG`.

Requirements:

- **Do NOT implement each algorithm as a separate function.** Instead, you should implement the general template for forward search (See Figure 2.4 in the textbook). Then, implement the corresponding priority queue for each algorithm.

  Hint: The only difference between different algorithms is how an element is inserted into the queue. So at the minimum, you should have the following classes (You can name them differently):

  - `Queue`: a base class for maintaining a queue, with `pop()` function that removes and returns the first element in the queue. You may also want this class to maintain the parent of each element that has been inserted into the queue so that you trace back the parent when computing the path from `xI` to a goal state in `XG`.
  - `QueueBFS`, `QueueDFS`, `QueueAstar`: classes derived from `Queue` and implement `insert(x, parent)` function for inserting an element `x` with parent `parent` into the queue.

  With the above classes, you can implement a function `get_queue(alg, X, XG)` that returns an appropriate queue `Q` for the given search algorithm `alg` and containing `insert` and `pop` functions (and possibly some other functions, e.g., for computing a path). Note that `X` and `XG` may be needed to construct the queue for `"astar"` algorithm since `X` provides the `get_distance_lower_bound` function. Together with `XG`, you can implement a function to compute the lower bound on the distance between any given state `x` to a state in the goal set `XG`. You can then use this `Q` in the `fsearch` function.

- Following the previous bullet, there should be only one conditional statement for the selected algorithm (`"bfs"`, `"dfs"`, or `"astar"`)) in the entire program. **Points will be deducted for each additinoal conditional statement.**
- For A*, assume that the cost of each transition is 1, i.e., cost-to-come to state x' is given by $C(x') = C(x) + 1$ where x is the parent of x'.

- Your implementation should work for **any** implementation of the `StateSpace`, `ActionSpace`, and `StateTransition` and their derived classes, not necessarily only those you will implement in Task 2. So you should not assume, e.g., that a state will be of any particular form.
- Your implementation should be able to handle corner cases, including but not limitted to the case where each of the goal states is not in the state space `X`, in which case `"visited"` may be either the entire state space or empty.
- Please feel free to use external libraries, e.g., for heap, queue, stack or implement them yourself.

Task 2 (Path planning with 2D grid) [10 points]: Consider a robot moving on a 2D grid. Each grid point has an integer coordinate of the form (i, j) where $i \in \{0, \ldots, M\}$, $j \in \{0, \ldots, N\}$, and M and N are positive integers. The world is, therefore, defined as

$$\mathcal{W} = \{(i, j) \in \mathbb{Z} \times \mathbb{Z} \mid 0 \leq i \leq M, 0 \leq j \leq N\}. \tag{1}$$

Some of the grid point may be occupied by an obstacle. Let $\mathcal{O}$ denote the set of all the grid points that are occupied. For example,

$$\mathcal{O} = \{(0, 0), (1, 0)\} \tag{2}$$

means that grid points $(0, 0)$ and $(1, 0)$ are occupied by an obstacle.

The state space includes all the grid points that are not occupied by an obstacle. Formally, the state space is defined as

$$X = \mathcal{W} \setminus \mathcal{O}. \tag{3}$$

For example, consider the case where $M = N = 1$ and the set $\mathcal{O}$ of obstacles is defined in (2). Following the definition in (1) and (3), the state space in this case is given by $X = \{(0, 1), (1, 1)\}$.

The robot takes discrete steps in one of the four directions (up, down, left, right) with the corresponding actions $u_{up} = (0, 1)$, $u_{down} = (0, -1)$, $u_{left} = (-1, 0)$, and $u_{right} = (1, 0)$. Let $U = \{u_{up}, u_{down}, u_{left}, u_{right}\}$.

The action space for $x = (i, j)$ is defined as $U(x) \subseteq U$ such that

$$x + u \in X, \forall u \in U(x). \tag{4}$$

Here, $+$ is the elementwise addition, i.e., $x + u = (x[0] + u[0], x[1] + u[1])$.

Finally, the state transition function $f : X \times U \to X$ is defined as

$$f(x, u) = x + u, \forall x \in X, u \in U(x). \tag{5}$$

As in Task 1, we assume that the cost of each action is 1, i.e., cost-to-come to grid point x' is given by $C(x') = C(x) + 1$ where x is the parent of x'. The lower bound on the distance between state $x_1 = (x_{1,1}, x_{1,2})$ and $x_2 = (x_{2,1}, x_{2,2})$ is given by

$$|x_{1,1} - x_{2,1}| + |x_{1,2} - x_{2,2}| \tag{6}$$

Your task is to implement the following functions in `hw1.py` in the course repo.

- `Grid2DStates.__contains__(self, x)` that returns `true` if `x` is in the state space as defined in (3).
- `Grid2DStates.get_distance_lower_bound(self, x1, x2)` that returns the lower bound on the distance between state `x1` and `x2` based on (6).
- `GridStateTransition.__call__(self, x, u)` that returns the new state obtained by applying action `u` at state `x` based on the state transition function f defined in (5).
- `Grid2DActions.__call__(self, x)` that returns the list of all the possible actions for state `x` based on the action space $U(x)$ defined in (4).

The problem description will be provided in a json file, containing the following fields:

- `"M"`: a positive integer that specifies the parameter M of the world as defined in (1).
- `"N"`: a positive integer that specifies the parameter N of the world as defined in (1).
- `"O"`: a list of grid points that are occupied by an obstacle.
- `"xI"`: a list $[i, j]$ specifying the initial state $(i, j) \in X$, and
- `"XG"`: a list of $[i, j]$'s, each corresponding to a goal state in `XG`. Note that **some states in `XG` may not be in X and `XG` may be empty**.

Once you have implemented all the necessary functions, you can run `hw1.py`, which takes 2 inputs: (1) the algorithm (`bfs`, `dfs`, or `astar`), and (2) the path to the json file, which specifies the problem description with the above format. It should output a json file, containing the following fields:

- `"visited"`: the list of visited cells, and
- `"path"`: the list of cells specifying the path from x_I to X_G.

For example, running

```
python hw1.py hw1_grid.json --alg bfs --out hw1_bfs.json
```

should output `hw1_bfs.json`, which contains the output of running breadth-first search, including `"visited"` and `"path"` for the problem described in `hw1_grid.json`. Example of `hw1_bfs.json` and `hw1_grid.json` can be found in the course github repo.

Requirements:

- To simplify grading, please make sure that `Grid2DActions.__call__(self, x)` returns the actions in the following order: u_{up}, u_{down}, u_{left}, u_{right}.

Submission: Please submit a single zip file on Canvas containing the followings:

- your code (with comments, explaining clearly what each function/class is doing), and
- a text file explaining clearly how to compile and run your code.