# COM S 4760/5760 Homework 1: Discrete Planning

1. (8 points) Consider a robot navigating a grid-like warehouse, represented as a 2D grid. Each location in the warehouse is identified by a coordinate $(x, y)$, where $x \in \{0, 1, \ldots, X_{max}\}$ represents the horizontal position (x-axis) and $y \in \{0, 1, \ldots, Y_{max}\}$ represents the vertical position (y-axis) for some natural numbers $X_{max}$ and $Y_{max}$. The robot can move in the following directions:

   - Up: This corresponds to adding 1 to the y-coordinate,
   - Down: This corresponds to subtracting 1 from the y-coordinate,
   - Left: This corresponds to subtracting 1 from the x-coordinate, or
   - Right: This corresponds to adding 1 to the x-coordinate.

   Some locations may contain obstacles, making them inaccessible to the robot. The goal is to compute an optimal path for the robot from a given starting location $l_i = (x_i, y_i)$ to a given target location $l_g = (x_g, y_g)$, ensuring that the path is collision-free.

   The definition of "optimal" can vary depending on the criteria used. For each of the following criteria, your task is to

   - Identify the elements of the planning problem, including the state space $X$, the set $U$ of actions, the action space $U(x)$ for each state $x \in X$, the state transition function $f$, and if applicable, the transition cost.
   - Determine the most efficient algorithm for solving the problem based on the given criteria. Provide the time complexity of your chosen algorithm, and clearly state any assumptions made to derive the runtime.

   **Criteria:**

   (a) Minimizing the Number of Moves: Find a path that minimizes the total number of moves (or steps) required to reach the target location from the starting location.

   (b) Minimizing the Total Time: Find a path that minimizes the total time taken to reach the target location, considering that the time required for each move may vary depending on both the move and the location where it is made.

   (c) Minimizing the Total Cost: Each move has associated time and energy costs. Given a location $l$ and a move $m$, let $c_{time}(l, m)$ and $c_{energy}(l, m)$ represent the time and energy, respective, for making a move $m$ at location $l$. For a sequence of moves $m_1 m_2 \ldots m_k$ that leads the robot through the corresponding path $p = l_1 l_2 \ldots l_{k+1}$, the total cost of the path is defined as

   $$c(p) = w_{time} \sum_{i=1}^{k} c_{time}(l_i, m_i) + w_{energy} \sum_{i=1}^{k} c_{energy}(l_i, m_i),$$

where $c_{time}$ and $c_{energy}$ are non-negative functions and $w_{time}, w_{energy} \in \mathbb{R}_{\geq 0}$ are weights representing the relative importance of time and energy in the cost function.

(d) (COM S 5760 only) Minimizing the Hierarchical Cost: In this scenario, time is considered to be strictly more important than energy. Therefore, you need to select a path that has the least energy consumption among all paths that achieve the minimum total time. Specifically, let $P_{time}$ denote the set of paths that minimize the total time taken to reach the target location. Then, select the one with the least energy consumption among all paths in $P_{time}$.

**Hint:** You do not need to explicitly identify the set $P_{time}$. Instead, compute the optimal path directly by considering both time and energy costs in a hierarchical manner, where minimizing time is the primary objective and minimizing energy is the secondary objective among those time-optimal paths.

Let $G = \{(x_1, x_2) \mid x_1 \in \{0, 1, \ldots, X_{max}\}, x_2 \in \{0, 1, \ldots, Y_{max}\}\}$ represent the set of all locations in the warehouse and let $O \subseteq G$ denote the set of locations occupied by obstacles. For all criteria, the state space is defined as

$$X = G \setminus O,$$

representing the set of all accessible locations.

The set of actions is

$$U = \{(0, 1), (0, -1), (-1, 0), (1, 0)\},$$

where $(0, 1)$, $(0, -1)$, $(-1, 0)$, and $(1, 0)$ correspond to moving up, down, left, and right, respectively.

The action space for each state $x \in X$ is defined as

$$U(x) = \{u \in U \mid f(x, u) \in X\},$$

where the state transition function $f$ is defined as

$$f(x, u) = (x_1 + u_1, x_2 + u_2), \forall x = (x_1, x_2) \in X, u = (u_1, u_2) \in U(x).$$

(a) To minimize the number of moves, we can apply BFS as discussed in class. The number of edges is $|U||X|$, so the running time is $O(|X| + |U||X|)$, assuming that all basic operations are performed in constant time. In particular, we assume that $U(x)$ is precomputed and stored as a dictionary for each $x \in X$. The transition cost is not needed in this case.

(b) To minimize the total time, for each $x \in X$ and $u \in U(x)$, we define the transition cost $c_t(x, u)$ as the time required for move $u$ made at location $x$. We can then apply Dijkstra's algorithm to find an optimal path. The running time is $O(|X| \log |X| + |U||X|)$, assuming that the priority queue is implemented with a Fibonacci heap and all basic operations are performed in constant time. In particular, we assume that $U(x)$ is precomputed and stored as a dictionary for each $x \in X$ and that the time required for move $u$ made at location $x$ can be computed in constant time for all $x \in X$ and $u \in U(x)$.

(c) To minimize the total cost, for each $x \in X$ and $u \in U(x)$, we define the transition cost $c_t(x, u)$ as

$$c_t(x, u) = w_{time} c_{time}(x, u) + w_{energy} c_{energy}(x, u).$$

We can then apply Dijkstra's algorithm to find an optimal path. The running time is $O(|X| \log |X| + |U||X|)$, under the same assumption as in (b), and that $c_{time}(x, u)$ and $c_{energy}(x, u)$ can be computed in constant time for all $x \in X$, $u \in U(x)$.

(d) To minimize the hierarchical cost, for each $x \in X$ and $u \in U(x)$, we define the transition cost $c_t(x, u)$ as a vector with 2 components:

$$c_t(x, u) = (c_{time}(x, u), c_{energy}(x, u)).$$

To compare these cost vectors, we use lexicographical order:

$$(c_1, c_2) \leq (c_1', c_2') \text{ if and only if } c_1 < c_2 \text{ or } (c_1 = c_1' \text{ and } c_2 \leq c_2').$$

In other words, we first compare the time component $c_1$ with $c_1'$. If $c_1$ is less than $c_1'$, then $(c_1, c_2)$ is considered smaller. If $c_1 = c_1'$, then we compare the energy components $c_2$ and $c_2'$. Note that the comparison takes constant time. With this new cost definition and comparison, we apply Dijkstra's algorithm. The running time and assumptions remain the same as in the previous part (c).

2. (8 points) Implement 2 variants of forward search algorithm: **breadth-first and A***.

These algorithms will be used to solve discrete planning problems specified by classes that are derived from the following abstract base classes:

- `StateSpace` Class:
  - Represents the set of all possible states in the problem.
  - Key methods: Let `X` be an instance of a class derived from `StateSpace`.
    - `x in X`: Returns a boolean indicating whether the state `x` is in the state space `X`.
    - `X.get_distance_lower_bound(x1, x2)`: Returns a float representing the lower bound on the distance between the states `x1` and `x2`
- `ActionSpace` Class:
  - Defines the set of all possible actions at each state.

  - Key methods: Let `U` be an instance of a class derived from `ActionSpace`.
    * `U(x)`: Returns the list of all the possible actions that can be taken from the state `x`.

- `StateTransition` Class:
  - Describes how the system transitions from one state to another based on an action.
  - Key methods: Let `f` be an instance of a class derived from `StateTransition`.
    * `f(x, u)`: Returns the new state obtained by applying action `u` at state `x`.

Below is the implementation of these abstract base classes. Your implementation of the search algorithms should work with any derived classes that fully implement these methods.

```
class StateSpace:
    """A base class to specify a state space X"""

    def __contains__(self, x) -> bool:
        """Return whether the given state x is in the state space"""
        raise NotImplementedError

    def get_distance_lower_bound(self, x1, x2) -> float:
        """Return the lower bound on the distance
        between the given states x1 and x2
        """
        return 0

class ActionSpace:
    """A base class to specify an action space"""

    def __call__(self, x) -> list:
        """Return the list of all the possible actions
        at the given state x
        """
        raise NotImplementedError

class StateTransition:
    """A base class to specify a state transition function"""

    def __call__(self, x, u):
        """Return the new state obtained by applying action u at state x"""
        raise NotImplementedError
```

**Task:** Implement the function `fsearch(X, U, f, xI, XG, alg)` where

- `X` is an instance of a class derived from `StateSpace` and represents the state space.
- `U` is an instance of a class derived from `ActionSpace` and specifies the action space.
- `f` is an instance of a class derived from `StateTransition` and 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. You can assume that for each state `x` in `XG`, the statement `x in X` returns `true`. However, `XG` **may be empty**.
- `alg` is a string that specifies the discrete search algorithm to use ( `"bfs"` or `"astar"`).

The function `fsearch(X, U, f, xI, XG, alg)` should return a dictionary with the following structure:

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

where `visited_states` is a list of states visited during the search, in the order they were visited and `path` is a list of states representing 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` and `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** classes derived from `StateSpace`, `ActionSpace`, and `StateTransition`, provided that all the abstract methods are correctly implemented in these derived classes. 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 `XG` is empty, 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.

See `discrete_search.py` and `queue.py`.

3. (4 points) Let's revisit Problem 1. Suppose the grid is of size $5 \times 5$, i.e., $X_{max} = Y_{max} = 4$. The robot starts at location $l_i = (0, 0)$ and needs to reach the target location at $l_g = (4, 4)$. The obstacles are located at the following coordinates:

- $(1, 3)$
- $(2, 3)$
- $(1, 2)$
- $(1, 1)$
- $(2, 1)$
- $(3, 1)$

**Task:**

- Implement the derived classes of `StateSpace`, `ActionSpace`, and `StateTransition` for this warehouse navigation problem.

- Write a script `main.py` that uses these derived classes, along with your implementation from Problem 2, to compute a path that minimizes the number of moves. You should be able to run the script with the following command:

  ```
  python main.py
  ```

  This should print out the required path.

See `main.py`.