Lab 12 — Heuristic Search

In class, we developed a heuristic version of the Mazerunner program, in which the search process is guided by an estimate of the remaining distance to the goal. In this lab, your job is to extend the 8-puzzle solver to use heuristic search in a similar way. To get started, download **lab12_files.zip** from the class web page, which contains the starting code for the 8-puzzle solver. Together, we will first implement a version that uses breadth-first search.

1. As an initial benchmark test, run the breadth-first search (BFS) version of the 8-puzzle solver on each of the predefined starting boards `Board.a` through `Board.h` provided in the Board class. In each case, you should use `Board.goal1` as your goal. The more challenging boards `x`, `y`, and `z` are much harder for breadth-first search to solve, so just ignore those for now. For each board, fill in the "BFS" column of the table on the next page with the length of the solution path found, followed by the number of boards examined, separated by a slash. For example, for a solution path of 10 moves, found after examining 769 boards, you would write "10 / 769".

2. To implement heuristic search, the first thing you'll need to do is make Board objects be comparable to each other, so that they can be stored in and retrieved from a priority queue. Add the instance variables `estimatedMovesLeft` and `priority` to the Board class, and implement a `compareTo` method that compares Board objects based on their `priority` values.

3. We also need a way to set a Board's priority value. Add a new method `setPriority` that takes a *goal* board as input and sets the priority based on an estimate of how far away the goal appears to be. As a first approximation, define a heuristic function called `heuristic1` that simply counts the number of *out-of-place tiles* on the board relative to the goal, and returns that as the estimate (hint: the `find` method will be helpful here). Make sure not to count the blank! Also define a more accurate heuristic function called `heuristic2` that computes the *Manhattan distance* (also called the "city block distance") between the board and a goal. Code for testing your heuristic functions on the following example is available in the EightPuzzle `main` method:

```
 board        goal
5 2 1        1 2 3          heuristic1 = 1 + 0 + 1 + 1 + 1 + 0 + 0 + 1 = 5
3 4 8        8   4          heuristic2 = 2 + 0 + 3 + 1 + 4 + 0 + 0 + 2 = 12
7 6          7 6 5
```

4. The next step is to modify the search algorithm to use a priority queue instead of an ordinary queue. Make a copy of the `breadthFirstSearch` method in EightPuzzle.java and change its name to `heuristicSearch`, and then modify it to use a PriorityQueue<Board>. Whenever a new board is created, you'll need to set the board's priority value before adding it to the queue.

5. Now you're ready to try out `heuristicSearch` on all of the starting boards. You should use `goal1` with boards `a` through `h`, and `goal2` with the more challenging boards `x`, `y`, and `z`. To begin with, try a **greedy search** based on heuristic1. In a greedy search, we only pay attention to the heuristic estimate of the remaining distance to the goal, and ignore the number of moves already made from the start state to the current state. Thus your `setPriority` method should look like this for greedy search:

```
public void setPriority(Board goal) {
    this.estimatedMovesLeft = heuristic1(goal);
    this.priority = this.estimatedMovesLeft;
}
```

Fill in the “greedy (h1)” column of the table with your results for each of the starting states. How do the solutions found compare to those found by BFS? Is greedy search able to find solutions to the harder boards `x`, `y`, and `z`? How do the total number of boards searched compare?

6. Now change the heuristic function used by greedy search to heuristic2 and repeat the tests, filling in the “greedy (h2)” column of the table below. How does heuristic2 (Manhattan distance) compare to heuristic1 (displaced tiles) in terms of the total number of states examined?

7. Now try **A* search**, in which the priority of a state is the *sum* of the distance already traveled from the start (`movesMade`) plus the estimated distance remaining to the goal (`estimatedMovesLeft`). This strategy, which relies on both pieces of information to guide the search—the first of which contains no uncertainty—can be shown to be optimal in the sense that it is guaranteed to find the shortest path to a solution while examining the fewest possible states, as long as the heuristic function never overestimates the remaining distance. Modify your `setPriority` method accordingly, and fill in the “A* (h1)” and “A* (h2)” columns of the table for heuristic1 and heuristic2. Does heuristic2 give better performance than heuristic1 with A* search? How does A* compare to greedy search, in terms of the solutions found and the total number of boards examined?

Start / Goal	BFS moves / boards	greedy (h1) moves / boards	greedy (h2) moves / boards	A* (h1) moves / boards	A* (h2) moves / boards
a / goal1					
b / goal1					
c / goal1					
d / goal1					
e / goal1					
f / goal1					
g / goal1					
h / goal1					
x / ***goal2***					
y / ***goal2***					
z / ***goal2***					