Homework Assignment 1: Search Algorithms

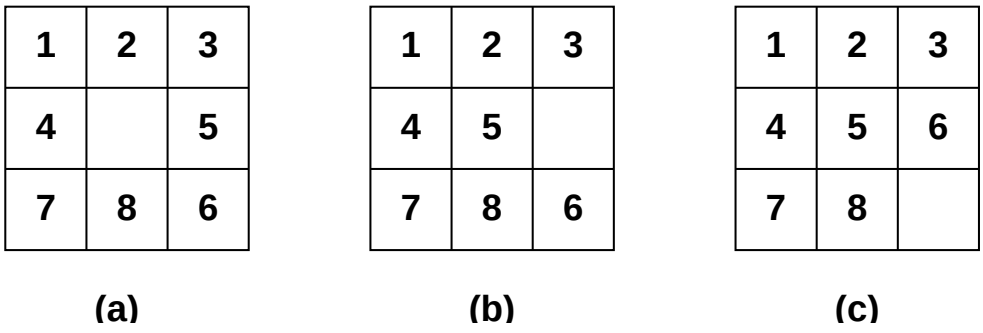
CS486/686 – Spring 2006
Instructor: Pascal Poupart

Out: May 9, 2006
Due: May 25, 2006 (no late assignment accepted)

Be sure to include your name and student number with your assignment.

In this assignment you will implement (using the language of your choice) two different search algorithms to solve the *8-puzzle*. You will first implement a neighbor function to define the implicit graph, then two heuristic functions and two search procedures (A* and IDA*). Finally, you will compare the performance of the heuristics and the search algorithms.

First, a bit of background. The 8-puzzle is the simple (one-person) game we discussed briefly in class where tiles numbered 1 through 8 are moved on a 3-by-3 grid. Any tile adjacent to the blank position can be moved into the blank position. By moving tiles in sequence we attempt to reach the goal configuration. For example, in the figure below, we see three game configurations: the configuration (b) can be reached from configuration (a) by sliding tile 5 to the left; configuration (c) can be reached from configuration (b) by sliding tile 6 up. Configuration (c) is the *goal configuration*. The objective of the game is to reach the goal configuration from some starting configuration with as few moves as possible. Note that not all starting configurations can reach the goal.



1. **(10 pts)** The first thing you should do is to implement a neighbor function. For any *state* of the game (i.e., any configuration), define a function `neighbor(state)` that returns a list of neighboring states. Feel free to use any representation to encode a state. Here is a possible representation: a state is a list with nine elements corresponding to the tiles (or the blank space). For instance, the three states shown above could be encoded, respectively, as:

   ```
   [1,2,3,4,b,5,7,8,6]      [1,2,3,4,5,b,7,8,6]      [1,2,3,4,5,6,7,8,b]
   ```

   In this game, we are interested in minimizing the number of moves necessary to reach the goal configuration, so you can assume a cost of 1 per move. For uniformity and ease of marking, the list of neighbors returned by your `neighbor` function should follow a specific order: each neighbor can be viewed as moving the blank position either up, right, left, or down; your neighbors should be listed in that order. Of course, if some of these

	Test Prob. 1	Test Prob. 2	Test Prob. 3
h1	L / E	L / E	L / E
h2	L / E	L / E	L / E

Table 1: L refers to the length of the path found and E refers to the number of nodes expanded.

neighbors are not possible (e.g., in configuration (b) above, the "right" neighbor doesn't exist), they won't be in the list.

What to hand in:

- Printout of your code
- Output of your `neighbor` function on three well-chosen test cases showing that it works. Choose those test cases yourself and give a short (one sentence) explanation of what distinguishes each test case from the others.

2. **(20 pts)** Implement the following two heuristic functions (to be used later in A* and IDA* search):

   - `h1(state,goal)` returns the number of tiles (excluding the blank) that are out of their desired `goal` position in state `state`. The $h1$-values of the 3 states above are 2, 1, and 0, respectively.
   - `h2(state,goal)` returns the total Manhattan distance of all tiles (excluding the blank) from their desired position. That is, for each tile, the Manhattan distance is the sum of the horizontal and vertical distances between its position in `state` and its desired position in `goal`. The $h2$-values of the 3 states above are 2, 1, and 0, respectively.

   What to hand in:

   - Printout of your code
   - Output of your heuristic functions on three well-chosen test cases showing that they work.
   - We have seen in class that those heuristics are *admissible*. Are they also *consistent*? Give a proof or a counter example for each heuristic.

3. **(20 pts)** Implement a function `astar(state,goal)` to do standard A* search. This function should return the path found to go from `state` to `goal`, the length of the path and the number of nodes *expanded* (i.e., number of nodes removed from the queue). Note that some `state-goal` pairs do not have any solution (i.e., no path), hence limit the number of expanded nodes to 4000. Note also that when a solution can be found (before the 4000-node cutoff), it is unique assuming that neighbors are inserted in the priority queue according to the order specified in Question 1 and that nodes inserted in the priority queue are placed after the nodes already in the priority queue that have the same f-value. I will post 3 test problems on the course website within a week of the assignment being handed out. Run your `astar` function with both heuristics on these 3 test problems.

   What to hand in:

   - Printout of your code
   - Printout showing the path found (if one is found before the 4000-node cutoff) for each test problem and each heuristic.
   - A table (like Table 1) indicating the length of the path found and the number of nodes expanded for each test problem and each heuristic.

4. **(20 pts)** Plain A* may repeatedly visit some states. How would you modify your A* function in Question 3 to avoid visiting the same state more than once? You do not have to implement this modified version of A*. Simply answer the questions below.

   What to hand in:

   - Describe briefly at a **high level** how to modify A* to avoid repeatly visiting the same states.
   - How does this modification affect the time and space complexity of A*?
   - Under what condition is this modified version of A* complete and optimal? Explain briefly.

5. **(30 pts)** Implement a function `idastar(state,goal)` to perform IDA* (iterative deepening A*) search. Allow this function to expand up to 9000 nodes. Using the same 3 test problems as in Question 3, run your `idastar` function with each heuristic.

   What to hand in:

   - Printout of your code
   - Printout showing the path found (if one is found before the 9000-node cutoff) for each test problem and each heuristic.
   - A table (like Table 1) indicating the length of the path found and the number of nodes expanded for each test problem and each heuristic.
   - What is the time and space complexity of IDA*?
   - Is IDA* complete? Explain briefly.
   - Is IDA* optimal? Explain briefly.