Homework Assignment 1: Search Algorithms

CS486/686 – Spring 2005
Instructor: Pascal Poupart

Out: May 10, 2005
Due: May 26, 2005 (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) three 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 three search procedures (A*, A* with multiple-path checking and IDA*). Finally, you will compare the performance of the two heuristics and the different 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	2	3
4		5
7	8	6

(a)

1	2	3
4	5	
7	8	6

(b)

1	2	3
4	5	6
7	8	

(c)

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 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: Hand in a listing of your code and a script showing three well-chosen test cases illustrating that your `neighbor` function 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: Hand in a listing of your code and a script showing three well-chosen test cases illustrating that your heuristic functions work.

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 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 your `neighbor` function produces neighbors in the order specified in Question 1.

   What to hand in: 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 and report in a script the path found (if one is found before the 4000-node cutoff), the length of the path and the number of nodes expanded. Hand in also your code listing.

4. **(15 pts)** Implement a second version of A* called `astar2`. This will not be too different from `astar` (i.e., you should be able to re-use most of your code), but `astar2` should perform *multiple path checking*. The simplest way to do this is to maintain a *closed list* of nodes that have been expanded. Before expanding a node, check in your closed list that the node hasn't been already expanded.

   What to hand in: Exactly as in Question 3.

5. **(20 pts)** Implement a function `idastar(state,goal)` to perform IDA* (iterative deepening A*) search. Allow this function to expand up to 9000 nodes.

   What to hand in: Exactly as in Question 3, but with a 9000-node cutoff.

6. **(15 pts)** Draw up a table that compares the number of nodes expanded by each of the three search algorithms (A*, A*-MPC, IDA*) for both heuristics and for each of the test cases posted. Your table should take the form:

	A*	A*-MPC	IDA*
Case 1, h1	x	x	x
Case 1, h2	x	x	x
Case 2, h1	x	x	x
Case 2, h2	x	x	x
Case 3, h1	x	x	x
Case 3, h2	x	x	x

   If any combination reaches the bound on the number of expanded nodes before solving the problem, please indicate that fact in your table. Discuss the results. What conclusions can you draw about the quality of the heuristics $h1$ and $h2$? What conclusions can you draw about the running times of each search algorithm when measured in terms of the number of nodes expanded?