CMPSC 370
Artificial Intelligence
Spring 2012
Bob Roos

Lab 3
Corrected on 9 Feb., 3:15 pm
Due Weds., 15 Feb., 11:59 pm

Brief Summary

The A^* algorithm and the "Eight puzzle".

There are two parts to this lab—an experimental part/analytical part and a programming part. Failure to do the programming part does not mean you can't do the experimental/analytical part; in fact, I am allowing for the possibility that some people might choose not to attempt the programming part.

Grading for this lab will therefore be as follows:

- Successful completion of all parts: A
- Successful completion of all parts except problem 8: B+ or $A-$ (depending on quality of answers and code)
- Successful completion of all parts except problems 7 and 8: B− or B

Note: "Successful" means correct and complete; grades will, of course be reduced if there are errors.

Useful Unix Fact

You can direct output to a file using the ">" operator, e.g.,

```
rroos$ java EightAStar 31245678_ > experiment1.txt
rroos$ java EightAStar 12345_786 > experiment2.txt
 ... etc. ...
```

These text files can then be pasted into your final report (with appropriate editing, e.g., delete long sequences of uninformative data).

Part 1: Experiments and Analysis

Download the files "`Puzzle.class`" and "`EightAStar.class`". (Don't use the "`EightAStar.java`" file for this part.) You can directly execute these in the terminal window without compiling. You must provide an initial puzzle state, e.g.,

```
$ java EightAStar 123_45786
```

The output will show a minimum-move solution (if one exists), the number of moves, and the number of nodes examined (the goal state is hard-coded in the program as `"12345678_"`).

For this first part, write up a report with your answers to the questions. You do not need to include output from all (or any of) the experiments. However, if you run an experiment that is in any way "interesting" (e.g., takes 30 or 31 moves to solve, or is shorter than you expected it to be), feel free to include its output.

1. (a) There are only two one-move-solution puzzles; test them. How many nodes are visited?
   (b) Create all different two-move-solution puzzles and test them. How many nodes are visited?
   (c) Create at least five (or more) different three-move-solution puzzles and test them. How many nodes are visited?
   (d) Do the same for four-move puzzles (you should be able to construct five of these easily).
   (e) For puzzles solvable in four or fewer moves, what is the pattern in terms of number of nodes visited by the A^* algorithm as a function of the number of moves needed to solve the puzzle?
   (f) Does this pattern continue? If so, demonstrate with random puzzles. If not, where does it break down and, more importantly, why does it break down?

2. How many states do you think the A^* algorithm will visit if we give it an unsolvable puzzle? (You probably can't determine this experimentally—on my machine it runs so long that I just gave up.) I'm not asking for an exact number here, just a qualitative answer.

3. Perform experiments that show average number of nodes visited during the search as a function of the number of moves needed to solve the puzzle; plot this information on a graph, labeled "Solution length" on the x-axis and "Avg. nodes visited" on the y-axis. I don't expect you to do an exhaustive search of all $9!/2$ possible solvable puzzles; however, try to gather as much information as you can. Describe how many puzzles of each length were used to compute each average value, e.g., "I used 5 puzzles of length 22, 2 puzzles of length 25," or whatever you can come up with. There may be holes in your data (for instance, there are only two puzzles that require 31 moves—I don't expect you to find either of them, although this information is "out there" on the Web somewhere).

4. Without actually performing any experiments, how do you expect the A^* algorithm to compare with breadth-first search in terms of number of nodes examined? How do you expect it to compare with iterative deepening? Explain your answers. (Why didn't I bother to ask about depth-first search?)

5. The `EightAStar` program uses the "Manhattan" heuristic to estimate distance to the goal. How accurate is it as a measure of number of moves needed to solve the Eight puzzle—does it ever overestimate the distance? does it ever underestimate it?

6. Suppose we used the following heuristic: add up the differences between each cell's contents and the digit that is supposed to be in that cell. (Treat the empty cell as zero.) For instance,

in the puzzle

8	5	1
4	2	7
3	_	6

, the value of h would be 34 ($|8-1|+|5-2|+|1-3|+\cdots+|0-8|+|6-0|$).

Will the A^* algorithm still be guaranteed to find a minimum solution to the puzzle? (Before answering, read section 4.3.1 of Luger! Ask if you don't understand.)

Part 2: Programming

Program `EightAStar.java` contains most of the code needed to implement the A^* puzzle. It will not work correctly as it is.

7. Implement the missing portions of the "`solve`" algorithm *except for* the second and third cases inside the "for each child" loop. In other words, check to see if the child is not on the `open` or `closed` list and do the appropriate thing; however, don't worry about the other two cases ("already on `open`" or "already on `closed`").

   This should compile and produce output, although it will not necessarily find a minimum solution.

   You should not need to modify any code other than what is marked with comments saying "complete the algorithm." In particular, you should not need to change the `findChildren` or `main` methods. You should not need to make any changes at all to the `Puzzle` class.

   If you can get your program to compile, it will probably produce a warning about using "unchecked or unsafe operations"; you may ignore this.

   You will not need to use the method named "`locate`" for this part.

8. Complete the implementation of the algorithm. To do this, you will need to use the method "`locate`," which is provided for you. Here's what's going on:

   - When we discover a child that is already in the `open` or `closed` list, we need to see whether the child's g-value (which is automatically set to 1 plus `x`'s g-value) is less than the g-value of the node in the `open` or `closed` list.
   - To do this, we must have access to the object in the list; the "`contains`" method doesn't do this for us, it only tells us that the object is there.
   - The "`locate`" method returns the object in the list so that we may compare its g-value to that of the child (there is a method of the `Puzzle` class named `g()` that returns this information).
   - If we discover that the child's g-value is smaller, we must remove the object we located from the given list and insert the child into the `open` list. (Both the `ArrayList` and `PriorityQueue` classes have a "`remove(Object)`" method.)

   To do this problem, you really have to understand the Java code at a fairly deep level. If you can't do this, but you were able to do the rest, you can still get an A- for this assignment.

Consult the "lab requirements" document for instructions on submitting code and reports.