MIDDLE EAST TECHNICAL UNIVERSITY
Department of Electrical and Electronics Engineering

EE 586 Artificial Intelligence

Inst: Assist. Prof. Afsar Saranli

Programming Assignment #1
Given: March 15, 2006, Due: April 5, 2006

In this programming assignment, you will be required to consider a simple puzzle and develop and compare the problem solving techniques using different search procedures that we have discussed in class. You are required to submit both a written report and a program pack (a zip file). You will work with your (already formed) project teams. The homework will refresh (or establish) some of the programming skills that you will need for the term projects. Note that you can follow the pseudo-code algorithm descriptions in your main reference text (Russel & Norvig) whenever feasible and convenient in implementing the required algorithms.

Important: In your written report, (and in a text file in the zip pack) reference all web locations that you have benefited from and links to any source code that you have made use of. You can get idea and inspiration from code available on the web but you are required to write your own code for any direct questions being asked as part of the assignment.) Also, you are required to documented your code so that someone reading your source knows exactly what you are doing!

Consider the 8-puzzle game illustrated in the figure below for a random *start state* (a) as well as the desired *goal state* (b). The aim of your AI search program is to attain the goal state from any arbitrary start state. You can find one Java implementation of this game at the link http://www.permadi.com/java/puzzle8/

4	8	6
	1	5
2	3	7

(a)

1	2	3
4	5	6
7	8	

(b)

1. In your chosen programming language, (C/C++ is preferred but Java and Matlab are equally acceptable) develop a computer representation to represent and store the puzzle state. Develop the representation such that it can handle an *L*-puzzle where the size of the puzzle is configurable by the user.

   Explain the structure of your representation. Note that apart from the “visible” puzzle state, this representation will also need to keep any bookkeeping information required for a particular search algorithm that you will implement (such as nodes awaiting expansion, back-pointers etc) One possible alternative is to use the heap memory and pointers to construct a data structure with nodes representing the puzzle states. You might wish to find and review a tutorial on *data structures* from the web. Those of you who are skilled and interested in Object

Oriented languages such as C++ may wish to consider separating the representation of the game from the representation of the algorithm so that the search algorithm(s) developed can be used on search problems other than the *L*-puzzle. If you do not have a clue what I am talking about, then this may not be the option for you!

2. Develop a basic GUI for your program that would satisfy as a minimum the following requirements:

    a. Let the user select the search algorithm used,
    b. Display the puzzle state continuously on the screen,
    c. Let the user single step through the search iterations of the chosen algorithm,
    d. Display the current iteration count of the search,
    e. Let the user start/pause/cancel a free-run through the search while displaying each puzzle state and the changing iteration count,
    f. Be able to generate a random initial state,
    g. Be able to perform a desired number of monte-carlo simulations (the experiment will be described in more detail as question 8): For *N* times (selectable by the user), start from the goal state, move *M* random steps (selectable by the user) away from the goal, consider the resulting state as the initial state and run the chosen search algorithm to reach back the goal. Accumulate and average over the *N* simulations the number of steps taken to reach the goal as well as the memory units used to perform the search.

The initial state of the puzzle can be loaded from a formatted text file for simplicity but you may develop an interface to let the user set up an initial state by mouse interaction with the GUI. If data is loaded from a file, then the interface should have the ability to choose the file name to be loaded.

3. Implement a function called successors() that, given any state, can produce the possible puzzle moves (next states) from that state in the chosen representation. By using the output of your function, your program should be able to move to those next states in the chosen representation. The function should also be able to handle L-puzzle case where *L* is configurable by the user.

4. Consider the following start state of the puzzle. If you are good at 8-puzzle, you may try to solve it yourself and tell how many moves away it is from the goal (i.e., what is the minimum path to the goal?) although this is not necessary.

    Now consider the Breadth-First Search algorithm and implement it in your program as one algorithm option. Run it on the given start state.

2	3	5
0		7
6	1	4

    a. How many nodes were expended to reach the goal state during the execution of your program?

b. How many tiles are moved in order to reach the goal state?
c. How much time did you program take to reach the goal state? (You can use a time measurement function available to your language/OS to answer this. Also provide your platform details)

5. Now consider the Depth-First Search. Can it be used in its simplest form to solve this puzzle? If not, what is the fix? Implement it the way you see fit to run on the same start state.

   a. How many nodes were expended to reach the goal state during the execution of your program?
   b. How many tiles are moved in order to reach the goal state?
   c. How much time did you program take to reach the goal state?

6. Consider now iterative-deepening search. Implement the search and answer (a)-(c) above for this case.

7. Now consider the A*-search algorithm for the same problem with the same initial state.

   a. Comment on the expected performance of the A*-search algorithm for different heuristic function definitions.
   b. Implement the two functions heuristic_misplaced(state) and heuristic_manhattan(state) to compute two alternative heuristic function values for the current state as an estimate of the distance to the goal,
   c. Implement the A*-search case with each of the two proposed heuristic functions to solve the problem, again providing your answers for (a)-(c) in each case.

8. Now you are required to provide a comparative study of the behaviors for the algorithms discussed above in terms of number of node expansions until reaching the goal and the required memory storage. These graphs will be plotted against the x-axis which is "true minimum distance to goal". This true distance can be determined while generating the initial state by starting from the goal state and moving away from the goal by a pre-determined number of steps.

   For performing this comparison, it is not enough to run the algorithms on a single initial state and record the algorithm performance. Since each initial state generated by moving away from the goal by a pre-determined is one possibility among many others, and algorithms will perform slightly differently with each initial state we need to average performance over many such initial states. This is called a "Monte-Carlo Simulation".

   Consider the following experimental procedure:

   a. For each experimental point in the two graph (e.g., for a fixed true distance *d* steps from the goal state) start from the goal state and move away *d* random steps to generate an initial state for the puzzle,

   b. Start from the resulting intial state and run the tested algorithm until goal state is reached. Record the number of node expansions (node processing) and the memory allocated (Maybe not bytes stored but rather more high level memory units such as "nodes stored" or "data items stored" etc.)

   c. Repeat the simulation *N* times for the same algorithm with a different initial state generated again by step (a). You may take *N*=20 or *N*=100 depending on the speed of your computer and hence the resulting duration of the simulations. A higher number of experiments, when averaged, would result in a better estimate of the true performance.

d. Average the performance parameters (number of nodes expanded and memory units utilized) over these *N* simulations and generate one graph point.

e. Repeat (a)-(d) for different true distances *M* from the goal state, generating *M* points on each performance graph,

f. Repeat (a)-(e) for each algorithm considered, plotting each curve on top of the other on the same graph (of the same type!) with a different color. Matlab is the best option for presenting these types of graphs even if the data was generated by another language such as C/C++.

g. Comment on your findings and compare with expected behavior.

Monte-Carlo simulations are standard practice where performance depends on a random initial condition or random behavior and single simulation performance is not really indicative of true "average" or "expected" performance.

9. Experiment with different puzzle size by picking an algorithm (e.g. A*-search) and a true distance from the goal. Simulate for a number of different puzzle sizes and comment on your findings.