

CIS*3700 (Winter 2007) Assignment One

Instructor: F. Song

Due Time: *Paper-and-pencil questions due on Feb. 1, right after the class and the implementation questions due on Feb. 4, by midnight.*

Part I: Paper-and-Pencil Questions (30 marks)

1. Artificial intelligence is the basis for a wide range of applications. For each of the following two examples, develop a PEAS description for the task environment and further characterize the environment in terms of the six properties discussed in the lectures.
 - (a) Robot soccer player.
 - (b) Automated language translator that translates what you say in one language to another foreign language.
2. The missionaries and cannibals problem is usually stated as follows. Three missionaries and three cannibals are on one side of the river, along with a boat that can hold one or two people. Find a way to get everyone to the other side of the river, without ever leaving a group of missionaries outnumbered by the cannibals at any side. Formulate this problem in terms of state space representation and draw a diagram for the complete state space. Is it a good idea to check for repeated states for this problem?

Part II: Implementation Questions (70 marks)

3. Implement the breadth-first search strategy and test it on the missionaries and cannibals problem. The top level pseudo-code for “TREE-SEARCH” is given in the lecture notes for “3-Uninformed Search”. Note that in the pseudo-code, INSERT-ALL is a generic function that adds new successor nodes to the fringe. For the breadth-first search, the fringe is maintained as a queue and all the new successor nodes are added at the end of the fringe. Also note that some functions are general and related to the tree search process, such as INSERT, INSERT-ALL, REMOVE-FRONT, and EXPAND, while others are specific and related to the problems themselves (e.g., GOAL-TEST and SUCCESSOR-FN will be different for the map search and water jugs problems, respectively).

With object-oriented programming, we can create abstract classes for the generic cases and sub-classes for extensions/differences. For example, we can first create an abstract class TREE-SEARCH for maintaining the fringe, along with the methods for INSERT, INSERT-ALL, REMOVE-FRONT, and EXPAND. After that, we can create a sub-class BREADTH-FIRST-SEARCH that implements the

fringe as a queue by inserting all new successor nodes at the end of the fringe. Accordingly, we should make INSERT-ALL a virtual method in TREE-SEARCH and provide a specific implementation for it in BREADTH-FIRST-SEARCH.

Since TREE-SEARCH can be applied to different problems, it will be useful to create another abstract class NODE, which can be used to represent the search graph for any given problem. In the lecture notes, we show that a node can have five different fields: state, parent node, action, path cost, and depth. Here, state corresponds to the new state as a result of an action; parent node allows us to trace back for a solution path; action indicates which action is used to create the new state from the parent state; path cost helps us find an optimal solution; and depth is useful for depth-limited search or iterative deepening search. Obviously, we may not need “path cost” and “depth” for the breadth-first-search; so it is desirable to trim the number of fields for NODE class and introduce sub-classes that contain the fields for “path-cost” and “depth”. The most specific sub-classes that extend from the chain of the NODE class will be the nodes for real problems, such as those for the map problem, water jugs problem, and 8-puzzle problem. Along with the desired fields for the NODE class and its sub-classes, we can also add problem-specific methods, such as GOAL-TEST and SUCCESSOR-FN. Again, we can make them virtual functions in the abstract classes and then provide concrete implementations in the specific classes.

Based on the above considerations for object-oriented design, you can fill in more details as you see fit in your implementation. In particular, you can improve the search process by showing the trace, i.e., displaying all the nodes examined during the search process, and the solution, i.e., displaying the path from the start state to the goal state.

4. Implement the A* search strategy and test it on the 8-puzzle problem. The 8-puzzle consists of a 3×3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into that space, or you can view these actions as moving the blank space in the left, right, up, and down directions. You need to choose a suitable representation for all the states and actions, and compute the f-values. The suggested heuristic function is the Manhattan distance and you can assume the unit costs for all the actions. You can test your program with the following pair of initial and goal states:

7	2	4
5		6
8	3	1

Initial State

	1	2
3	4	5
6	7	8

Goal State

Once again, you need to follow the object-oriented design described in question 3 and provide implementations for A-Star-Search and the problem-specific functions such as GOAL-TEST and SUCCESSOR-FN. You should further improve your program by considering the following possibilities:

- (a) Show the trace and the solution path, and in particular, display the board as a 3×3 grid for improved readability.
- (b) Dynamically check cycles so that the repeated states are pruned away before they are added to the fringe. In the Romania Map, for example, we could start from Arad to Zerind to Oradea to Sibiu and then back to Arad.
- (c) Reduce the search effort by removing all the longer paths to the same node in the current fringe. Again in the Romania Map, we could have two Craiova nodes in the fringe: one is the path Arad → Sibiu → Rimnicu Vilcea → Craiova and the other is the path Arad → Timisoare → Lugoji → Mehadia → Dobreta → Craiova. If we assume the unit cost between two cities, the second path is longer than the first path and thus can be removed from the fringe for further consideration.

Submission requirements for the implementation question(s):

Your implementation can be done in either C++/STL or Java so that you can take advantages of the support for object-oriented programming and data structures. In addition, your programs should run in a Linux machine such as those in Reynolds 004/008 labs. Your program will be marked for both correctness and style. In particular, a complete submission should include: (i) a README file, describing what has been done, what are the limitations, what improvements could be made if you were to do it again, how a user can use it, and how it is tested for correctness; (ii) a Makefile that allows for “make clean” and “make” so that we can create a new build for marking; and (iii) all the source code and testing files. Tar and gzip all of your files into one compressed file (name it in the form of <userid>_a<#>.tar.gz, e.g., fsong_a1.tar.gz) and email it to ta3700@cis.uoguelph.ca by the due time.

Important Note: You should carefully test your program; incomplete code and code that doesn't compile will face a big penalty.