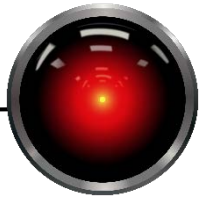


Assignment 02

1. (20 points) Explain the difference between a problem state and a search state.

2. (20 points) Work problem 4.1 from your book. For 4.1 c, ignore that the simulated annealing algorithm presented in figure 4.5 would terminate immediately when the temperature is zero.

3. (20 points) Consider the problem of managing the reproduction of a critically endangered species in a captive environment. One fitness function could be the population size although more sophisticated ones might take into account indicators such as the number of births, mortalities, genetic diversity, and behavioral indicators of stress, fertility, etc. Assume that you are managing this population and that your facility has the following resources:
   - N_m males
   - N_f females
   - N_d units of food/day
   - N_e enrinchement items (toys to prevent boredom)

   Devise a state representation that partitions the food, animals, and enrichment items amongst three habitat units. Propose a crossover function that is different from the one used on the N-queens problem that could be used in a genetic algorithm search if we could predict the fitness from your state.

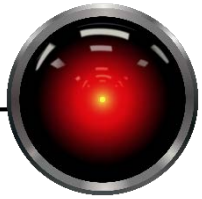
4. (20 points) Consider the fairly simply problem of selecting 2 matching cards from a deck of cards with pictures on them. The pictures consist of moon, sun, and stars, and there are two instances of each card (6 cards in total). Legal moves are pick a card that has not been picked, and the goal state is matching the first card drawn. Sketch an and-or search tree for this problem. You do not need to draw the full tree, just enough to make it clear that you understand what the full tree would look like.

Coding: N-Puzzle Search (120 points)

In this assignment, you will write a generic graph search routine that can be used to conduct breadth-first, depth-first, and A* search by varying the parameters given to it. Several classes are provided to you in package basicsearch_lib02 to help you in this endeavor and you are required to use them.

Skeleton code is provided in the following files that can be accessed from Blackboard:

driver02.py – The driver module will create 31 different tile board puzzles. For each of these, a solution will be searched using breadth first, depth first, and A* search using a Manhattan distance heuristic. Keep track of the number of nodes expanded and the amount of time used for each of these. Upon completion, print a table summary that indicates the mean (average) and standard deviation of the following measurements:



- length of plan (number of steps to solution)
- number of nodes expanded
- elapsed time in seconds

Running the 31 trials will take a little bit of time, so use a much smaller number when debugging. One execution of the 31 trials on an Intel i7-6600U took about 25 minutes (duration depends on the difficulty of the randomly generated puzzles) and it is recommended that you make sure that each search type is working before you try to start looping on solutions.

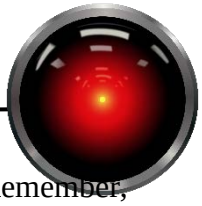
npuzzle.py – Implement class NPuzzle. It is subclassed from the Problem class in basicsearch_lib02.searchrep. You will need to provide an implementation that allows you to pass in functions g and h to provide the search strategy as well as support TileBoard constructors. Some methods will need to be overridden. For example, the actions(state) method will need to invoke the appropriate method associated with a TileBoard.

problemsearch.py – Implement function graph_search. It takes an instance of NPuzzle and flags for controlling verbosity and debugging (see file for details) and conducts a search. When instantiating the NPuzzle, be sure to use parent class's g and h arguments to set the path cost and heuristic functions which will govern the type of search that is conducted. It returns a tuple consisting of a plan (list of actions) and the number of nodes that were expanded. You will need to use a priority queue and search nodes (see below)

explored.py – Implement class Explored. Apart from the no argument constructor, it has two methods: exists(state) and add(state). Both of these expect state tuples from a TileBoard and use a hash table to determine whether a state has been seen before (exists) and to add new states as they are removed from the frontier set (add). The Python builtin hash will generate a hash key from any hashable value. Handle hash key collisions as a bucket list.

searchstrategies.py – Implement classes BreadthFirst, DepthFirst, and Manhattan. These are classes that provide implementations for the cost to node (g) and cost from node to goal heuristic (h) functions. Note g and h are class methods (see details on class methods in the comments of the code provide) and that when you pass them to the NPuzzle constructor, you need to pass the function handles to the constructor rather than invoking the function. Concretely, if you wanted to pass BreadthFirst's function handle for g to NPuzzle, you would call NPuzzle(num_tiles, g=BreadthFirst.g, …). NPuzzle's parent class stores g in a publicly accessible instance variable and other code (e.g. the Node class discussed below) will invoke the functions to evaluate search nodes. Note that we defined DepthFirst's g as a constant and h as the negative depth. The structure of the Node implementation expects h to be called with a single argument: state. As the depth is captured in the Node, and not the state, reverse the roles of g and h when implementing DepthFirst. The depth can be accessed from the g function which expects a parent, action, and the search node itself.

There are several helper classes that you should use from modules in the basicsearch_lib02: queues.py and searchrep.py



From queues.py use class PriorityQueue to maintain the order of the queue. Remember, we defined g and h in a way that priority queues can be used all search types and f is the sum of these.

From searchrep.py, in addition to the previously mentioned Problem class, class Node should be used to construct a node in the search space, and these are the objects that should be generated in your search. To create the first node, call the constructor with instances of the problem representation, NPuzzle, and the initial state (a TileBoard). To obtain the list of new search states obtainable from a search node, call expand with an instance of the problem. It will use the problem's actions method to determine the possible actions that can be generated and derive new child nodes. Node's get_f function will be useful for maintaining the order of your priority queue. While you need not modify anything in the library, be sure that you understand how this is working. Step through the code with a debugger if you cannot follow it by reading. Finally, searchrep provides function print_nodes that takes a list of search nodes and prints out their puzzle representations on a single line. You may find this useful for debugging, but otherwise the function provides no purpose.

Turn in hard copy of your code and output and a soft copy of your code excluding the provided library module. Be sure to attach a single or pair programming affidavit. If you are pair programming, turn in the two sets of individual questions and the joint programming assignment as one package and only submit to Blackboard under one id.