CS 534
Assignment #1: Search

Due: Feb 24 @ 10:59 a.m. [i.e., right before class]

This assignment will require you to explore a variety of search techniques on two problems. Note: in addition to the programming component of developing the algorithms, there is a moderate amount of analysis/writeup required as well. If you finish your code just before the deadline, **you will not finish the assignment**. Start early.

Part 1. Heavy N-queens problem

The problem

For this problem, you will work on a variant of the N-queens problem we discussed in class. Some of the queen pieces are very heavy, and therefore difficult to move. Queens have a weight ranging from 1 to 9 (integers). The cost to move a queen is the number of tiles you move it times its weight. So to move a queen weighing 6 upwards by 4 squares in a column costs 6 * 4 = 24.

In addition, your program must work with a given starting board and find a low-cost solution. In other words, if you use random restarts you may not rearrange the pieces on the board. You are allowed to make different choices about which Queen to move on the restart.

Otherwise, the rules are identical to the N-queens problem.

Heuristics

For your search, you need a heuristic function. The number of (directly and indirectly) attacking queens is a good place to start. Each pair of attacking Queens requires moving one of them -- presumably the lightest. You should look *at each pair* of attacking Queens, and select the lightest one to move. Construct two heuristic functions:

1. H1. The lightest Queen across all pairs of Queens attacking each other. Moving that Queen is the minimum possible cost to solve the problem (in real life it will probably take more effort than that though).
2. H2. Sum across every pair of attacking Queens the weight of the lightest Queen.

It is not immediately obvious that H2 is admissible. Either prove that it is admissible or construct a counterexample to prove that it is not.

Extra credit: develop a heuristic H3 that is is better than H1 and is admissible. Prove/demonstrate that H3 is admissible.

Approaches

You will use two approaches for this problem: A* and greedy hill climbing with restarts. You are free to make use of sideways moves, first choice hill climbing, or simulated annealing. You are not required to use any particular technique for deciding how to allocate your time. However, your decisions should have empirical support. In other words, you will need to present data for why (for example) you do first-choice hill climbing with at most 3 sideways moves.

For hill climbing with restarts, you should use H1 and H2 and perform greedy hill climbing. If you have not found a solution, and fewer than 10 seconds have elapsed since the program started running, you should do another iteration of hill climbing with the same start state (you may **not** change the initial configuration of Queens).

Program behavior

Your program should take as input the N value for the N-queens problem, the type of search to conduct (1 for A*, 2 for greedy hill climbing), and which heuristic to use (H1 or H2). For example, passing in a “30 1 H2” results in attempting to solve a 30-queens puzzle with A* search using H2. You can take input as command-line input, or read it from a file if command-line input is difficult for the language you are using. Just explain how to invoke your program in your writeup.

Your program should output:

- The start state (a simple text representation is fine)
- The number of nodes expanded. Consider an expansion as when you visit a node and compute the cost for all of the possible next moves. For hill climbing, remember to count this value across all of the restarts!
- Time to solve the puzzle
- The effective branching factor: consider how many nodes were expanded vs. the length of the solution path
- The cost to solve the puzzle. Note that A* computes cost automatically, but you will need to extend greedy search to keep track of this number. Note that cost does not include any failed runs with greedy search -- just compute the cost of the path it computes for the board it solves. (Note the comparison with A* is not quite fair since greedy search gets to discard its results from tougher boards)
- The sequence of moves needed to solve the puzzle, if any (hill climbing may fail).

Writeup

You should conduct experiments to determine:

1. How large of a puzzle can your program typically solve within 10 seconds using A*? Using greedy hill climbing?
2. What is the effective branching factor? For this computation, perform 10 runs of a puzzle half the maximum size you calculated for step #1.
    a. Compute the branching factor using H1 for A* and greedy search
    b. Compute the branching factor using H2 for A* and greedy search
3. Which approach comes up with cheaper solution paths? Why?
4. Which approach typically takes less time? Why?
5. Either prove the heuristic for A* is admissible, or provide a counterexample demonstrating that it is not.

Part 2. Urban planning

The problem

For this problem, you will use hill climbing and genetic algorithms to determine the ideal location of industry, commerce, and residential zones in a city. You will input a map with the following symbols:

- X: former toxic waste site. Industrial zones within 2 tiles take a penalty of -10. Commercial and residential zones within 2 tiles take a penalty of -20. You cannot build directly on a toxic waste site.
- S: scenic view. Residential zones within 2 tiles gain a bonus of 10 points. If you wish, you can build on a scenic site but it destroys the view. Building on a scenic view has a cost of 1.
- 1...9: how difficult it is to build on that square. To build a zone on any square costs 2+difficulty. So building a Commercial zone on a square of difficulty 6 costs 8 points. You will receive a penalty of that many points to put any zone on that square.

You will have to place industrial, residential, and commercial tiles on the terrain.

- Industrial tiles benefit from being near other industry. For each industrial tile within 2 squares, there is a bonus of 2 points.
- Commercial sites benefit from being near residential tiles. For each residential tile within 3 squares, there is a bonus of 4 points. However, commercial sites do not like competition. For each commercial site with 2 squares, there is a penalty of 4 points.
- Residential sites do not like being near industrial sites. For each industrial site within 3 squares there is a penalty of 5 points. However, for each commercial site with 3 squares there is a bonus of 4 points.

Note that all distances use the Manhattan distance approach. So distance is computed in terms of moves left/right and up/down, not diagonal movement.

Approaches

For this component you will use:

1. Hill climbing with restarts and simulated annealing
2. Genetic algorithms

When your hill climbing restarts, it may **not** modify the terrain on the map! It can only change where it initially places the different zones (and how many zones to place) to begin the hill climbing process.

Your hill climbing will have to trade off the number of restarts (more is better, but takes time) vs. how quickly to decrease the temperature (slower is better, but takes time). You should also think about what actions you will allow. Moving a zone is a sensible action. Should there also be an action for adding/removing a zone?

Your genetic algorithm will have to come up with a representation, selection and crossover methods, and **must** make use of culling, elitism, and mutation. If you set a parameter to 0 (for example, the mutation rate), that is acceptable, but you must provide data for why you did that.

Program behavior

Your program should accept command line input (unless difficult in your language) where the parameters are the file to read in, and the technique to use. For example “plan map.txt GA” would read in map.txt and use the genetic algorithm approach. “plan map.txt HC” would use hill climbing search.

Your program will read in a file where the first 3 lines are the *maximum* number of industrial, commercial, and residential locations (respectively). The remainder of the file is a rectangular map of the terrain to place the town. Your program should then run for approximately 10 seconds and output the following to a file:

- The score for this map
- At what time that score was first achieved.
- The map, with the various industrial, commercial, and residential sites marked.

Writeup

You should analyze your program’s behavior to answer the following questions:

1. Explain how you traded off number of restarts and the schedule for decreasing temperature. We are not expecting a dissertation-level analysis, but you should provide a couple of experiments explaining how you selected your parameters. Running a few trials works much (!) better than just making the numbers up.

2. Explain how your genetic algorithm works. You must describe your selection, crossover, elitism, culling, and mutation approaches. Give some data to explain why you selected the values that you did. How do elitism and culling affect performance?
3. Create a graph of program performance vs. time. Run your program 10 times and plot how well hill climbing and genetic algorithms perform after 0.1, 0.25, 0.5, 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10 seconds. If you only had 0.25 seconds to make a decision, which technique would you use? Does your answer change if you have 10 seconds?
4. How did you perform selection and crossover for your population? Find some known method for doing selection, other than the one described in the slides. Explain what you did.

What to hand in:

1. You should submit your assignment on Canvas.
2. Submit one file for part 1, and one file for part 2.
3. Include instructions for how to run your code. If the TA needs help from you or is confused, you will lose points.
4. The writeups from part 1 and 2.

Hints

1. There is a moderate amount of analysis and writeup for this assignment -- do not forget that component.
2. Your program should run for the specified length of time. Note the person grading your program may have a different hardware setup, so do not have a fixed number of iterations before the program terminates. Instead use calls to determine the amount of elapsed time. We are not worried about perfect precision -- if your program terminates after 10.2 seconds, that is ok. Anything in the range of 9.5 seconds to 10.5 seconds is fine.
3. Some points for this assignment are based on algorithm performance. For example, we will give you a test map for urban planning and see how well your program does. The scoring approach is there is a notion of “reasonable” performance, and failure to get a score that high will cost points.