COMPSCI 570: Artificial Intelligence

Homework 1: Search (due September 21, 11:59pm)

Please read the rules for assignments on the course web page (http://www2.cs.duke.edu/courses/fall21/compsci570/). In particular, you must **write all your code from scratch** for this assignment. Please let us know if you have questions about what you can use. Please use Ed Discussion for questions (use private questions if your question is likely to reveal part of the answer to others). Use Gradescope to turn this in.

In this assignment, you will code up A* **or** IDA* (your choice), and apply it to the two problems below. (If you are discussing this homework with another person, I recommend that one person does A*, and the other IDA*.) You may use any mainstream programming language that you like (though you are more likely to get useful help from the TAs with Python, if you need it), but your code should be legible. Your code will be tested on some secret instances of the problems, so you should be careful to follow the exact conventions for representing input and output, below. Note that you should not expect your algorithms to solve every instance (difficult instances may require too much time or memory; that does not mean that you did not solve the problem correctly). Of course, your code is expected to output the right answer when it outputs something.

Because you will apply your search algorithm to two different problems, it is a good idea to keep your code very general and modular, to make it easy to apply it to different problems. For example, you may wish to define abstract classes for "states" and "search nodes." Of course, the procedure for, say, computing the heuristic value will be specific to the problem.

Some potentially helpful files are available on the website. You should make sure that your code passes the public test cases provided there.

You should turn in your files using Gradescope. Please include a README file describing your files, instructions for compiling your code, etc.

1. Fifteens puzzle with knight's moves.

Normally, in the fifteens puzzle, you can move any horizontally or vertically adjacent tile into the empty slot. We now modify the puzzle so that you can only move a tile that is a knight's move (two spaces in one direction, one in the other) away from the empty slot.

For example, here is the goal state:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

The following states are one move away from the goal state:

1	2	3	4
5	6		8
9	10	11	12
13	14	15	7

1	2	3	4
5	6	7	8
9		11	12
13	14	15	10

The following state is nine moves away from the goal state (move the 6 out of the way; rotate 13, 11, and 4 to their correct positions; move the 6 back; then 12, 3, 10 to their correct positions).

1	2	12	13
5	6	7	8
9	3	4	
11	14	15	10

You should create your own heuristic (something similar to the sum of Manhattan distances that we used for the regular fifteens puzzle, though if you want to try to design even better ones, that is fine—the book will have more detail for this).

The input for your executable should look as follows (0 denotes the empty slot):

```
 1   2  12  13
 5   6   7   8
 9   3   4   0
11  14  15  10
```

The output should display the sequence of states (each in the above format) in an optimal solution, and the length of an optimal solution.

2. Superqueens puzzle.

Consider a modified chess piece that can move like a queen (i.e., any distance horizontally, vertically, or diagonally in one move), but also like a knight. We will call such a piece a "superqueen" (it is also known as an "amazon"). This leads to a new "superqueens" puzzle. We formulate the puzzle as a constraint optimization problem: each row and each column can have at most one superqueen (that's a hard constraint), and we try to minimize the number of pairs of superqueens that attack each other (either diagonally or with a knight's move).

For example, the following is an optimal solution for a 7 by 7 board (with the 1s representing where the queens are): there are 3 total attacks (two diagonal, one a knight's move).

1	0	0	0	0	0	0
0	0	0	0	1	0	0
0	0	0	0	0	1	0
0	1	0	0	0	0	0
0	0	1	0	0	0	0
0	0	0	0	0	0	1
0	0	0	1	0	0	0

You can use the number of pairs of superqueens that attack each other as either your cost or your heuristic, and set the other to 0. You are not required to implement sophisticated variable ordering (that is, you can always consider the variables in the same order), constraint propagation, etc. (although you are welcome to do so).

The input for your executable should consist of a single number, indicating the size of the board (for example, 8 means a standard 8 by 8 chessboard with 8 superqueens). The output should draw an optimal positioning of the superqueens on the board, and it should state the number of pairs of superqueens that attack each other.