CPS 170: Artificial Intelligence

Homework 1: Search (due January 29 **before** class)

Please read the rules for assignments on the course web page. Of course, **do not share code**. Contact Dima (dmytro.korzhyk@duke.edu) or Vince (conitzer@cs.duke.edu) with any questions.

In this assignment, you will code up A*, and apply it to the two problems below. In fact, you do not need to write all the code; some of it is already provided in Java (see the website), and you just need to fill in the missing parts. If you would much rather use another language, that is acceptable, but we **strongly** recommend using the existing code (which will also be easier because we will not provide support code in any other language). If you are not comfortable with some of the concepts used in the existing code (for example, abstract classes), Dima can help you—but please do not put this off until the last minute.

In the descriptions of the problems below, there are some test cases which you should use to test your code. Your code will also 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 necessarily 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 homework 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 the code very general and modular. The existing code will effectively force you to do this.

You should turn in your files by e-mailing them to Dima. Please include a README file describing your files. Also, please make sure your code is legible.

1. Fifteens puzzle.

The first problem that you will solve using A* is the classic fifteens puzzle (the four-by-four version of the eights puzzle studied in class). Every move has a cost of 1, and of course, since you are using A*, your program should find the optimal (lowest-cost) solution. In principle, there may be more than one optimal solution; your program is just expected to give one of these optimal solutions—it does not matter which one. The below shows some test cases with the expected input-output behavior. 0 represents the empty square.

Input #1:

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

Output #1:

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

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

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

Total # of moves: 2
```

Input #2:

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

Output #2:

(Step by step solution skipped here...)

```
Total # of moves: 24
```

For your heuristic, you can use the sum of Manhattan distances, or, if you want, you can try to design something even better (the book has some more detail).

2. Superqueens puzzle.

Consider a modified chess piece that can move like a queen, 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

The cost of a node should be the number of pairs of superqueens that attack each other so far; you can set the heuristic 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. For example:

Output for $n = 4$:

```
.Q..
Q...
..Q.
...Q
Number of conflicts: 4
```

Output for $n = 6$:

```
...Q..
Q.....
....Q.
.....Q
.Q....
..Q...
Number of conflicts: 3
```