Fall 2015 **COSC 3P71 Artificial Intelligence: Assignment 1**

Instructor: Beatrice Ombuki-Berman
TA & Tutorial Leader: Justin Maltese

Due date: Monday October 19th, 9:30 am (**No lates)**
Goal: Using blind and informed search algorithms to solve the N-Queens problem.
Languages: Any programming language available in our labs!

N-Queens: A queen is a chess piece that can move over all squares of a chessboard that are horizontal, vertical, or diagonal to it. For example:

		*			*		
*		*		*			
	*	*	*				
*	*	Q	*	*	*	*	*
	*	*	*				
*		*		*			
		*	X		*		
		*				*	

In the above diagram, the queen (shown as the symbol Q) can attack any square with a * on it. The problem is simple – we desire to place N queens on an NxN chessboard such that no queen is in a position to attack any other queen. Of course, this logically means that at most a single queen is in each row, in each column and in each diagonal. Only 1 queen can be placed on each board square at a time.

Referring to the board above, a second queen could be safely placed at the location marked with an "X". That queen would then further restrict the available squares to place another queen. Getting 8 queens legally on the 8x8 board above is possible, but depends entirely on where the queens themselves are positioned. It is thus important to choose carefully when placing queens, since poor placement may diminish all legal positions before actually placing N queens**!**

Your Task: You are going to solve the classic N-Queen problem using 2 different techniques discussed in class:

1. Use a **blind search** algorithm from Section 3.4 (Class textbook).
2. Use a **heuristic search i.e., informed search** from Section 3.5. Note that heuristic searches are reliant on pseudo-random number generators. If you change the seed of the RAND generator, you will get a different search. Hence you should run heuristic searches a number of times with new seeds. An example of a scheme you can use here is as follows. The initial board is one in which the queens are randomly placed all over the board. The next-state generator will take a current board, and generate one or more new boards by taking one of the queens, and moving it to a different location. Each new board needs to have a heuristic score assigned to it. One straight-forward heuristic is to count the

number of queens that are threatening to attack each other on a given configuration. Hence, some new boards will be "fitter" than others, in that there may be fewer queens threatening to attack one another.
(Can you think of other heuristics? Feel free to give them a try)

You can use any blind or heuristic search from chapter 3 that you like. Remember that the blind technique will run exhaustively, to find the best solution. The heuristic technique will use a heuristic score to try to find a good solution.

Comments:
Each run should print the following (You can capture text output in a unix window using the "script" utility):

a) The random number seed used (for heuristic searches). This lets you reproduce your results, and is useful for debugging your program.
b) The final board configuration of the solution, written as an ASCII table.
c) The total # board configurations tested during a given search. This will give you an idea of how much work was done to find a solution.

Be sure to organize all your search experiments. Clearly document the results of each experiment. The marker should be able to tell exactly what experiment is being run, by reading your documentation for the experimental results.

The "next state" generator should return a configuration of queens on the NxN board given the current configuration. This could be a NxN binary table/matrix.

Your program will be marked on correctness and style. Marks will be deducted for inadequate documentation; please discuss your design decisions in your program comments.

Hints: One of The most important design considerations is your knowledge representation for the problem at hand. How will you represent the board and pieces? Problem states? Problem transitions? The goal state?

The programming language you use may influence your representation and implementation. Note that your program shouldn't literally denote the search tree with a tree data structure. The textbook uses queues, and recursive programs can use the recursive stack calls to implicitly represent the search tree. Recursion can simplify your algorithm. Beware of infinite looping, especially with blind searches (i.e., repeated states).

Hand in: (i) A listing of your source code, and a listing of your program execution for a representative sample of cases (must include a case when N = 8), including those requested. Include department cover page. **(ii)** Electronic submission of all your code and data. Use "c3p71a" to submit your code. Run this at the top-level folder of your assignment directory. Be sure to name your top-level folder "Windows" or "linux", to

indicate what platform your executable is compiled for! The marker will use this to try your program.