EC 504
Spring, 2022
HW 3 Software

Due Monday, 3/28/21, 8 pm

Note: This must be turned in by creating a HW3 folder in your directory in /projectnb/ec504/students directory on the SCC cluster (scc1.bu.edu). Please turn in the output file and the source files for your code. Repeat what you did in HW0 and HW1.

1. (30 pts)**Finding Connected Components** In this problem, you will search two undirected Graphs, to find the connected components. You will output a vector which is of the length number of vertices, where each vertex will be assigned the number of the component the vertex is in, from 1 to total number of components. You will need to complete the two functions to do Breadth First Search of the graph, as well as Depth First Search.

   An undirected graph $G(V, E)$ is defined in terms of two arrays: $FirstVertex[0 : Vsize]$ and $EdgeList[0 : Esize]$ that label the vertices from $0, 1, \ldots, Vsize-1$ and edges from $0, 1, \ldots, Esize-1$, respectively. The last "fake" vertex ($FirstVertex[Vsize] = Esize$) points to the null "fake" edge with "null" value ($Edge[Esize] = -1$). This is the Forward Star data structure discussed in class.

   In the web site, in the HWProblems/HW3_codes directory, you will find a `connect.cpp` function that has parts that need to be completed. The functions need to count the number of connected components, and fill out the `Connect` array for each vertex, listing the index of its connected component.

   The included Makefile can be used to `make connected` . Two graphs are included: `graph_100_190.txt` and `graph_2000_4090.txt` . To run the code, run `./connected graph_100_190.txt` and then run `./connected graph_2000_4090.txt` . Each of these runs will generate an output file `graph_100_190.txt_out` and `graph_2000_4090.txt_out` . Turn in your completed code for connect.cpp and the two output files in your HW3 folder.

2. (20 pts)**Searching Decision Graphs** There are many decision problems where finding a solution can be cast as a search problem in a graph of states. In this exercise, you will solve another large decision problem through implicit enumeration and search on the graph. Consider the classical 8 queens problem in chess: Given a chess board, how do you place 8 queens so that no two queens can capture each other? That is, each row must have only one queen, each column must have only one queen, and no two queens can be on the same diagonal.

   How do we pose the problem as a search problem in a graph? One needs to define the concept of a state of a computation. A state will correspond to a partial solution. An edge in this graph represents a transition between a partial solution and a new partional solution with one additional queen on the board. The idea is to build a complete solution from partial solutions, by traversing the graph of possible partial solutions until one finds a complete solution.

   What is the graph of partial solutions? To convert the 8 queens problem to a sequential problem, consider the placement of queens one row at a time. Thus, after placing a queen in the first row, one has a partial solution (a first-layer solution). Having placed queens in the first two columns, there is another partial solution (second layer solution), corresponding to a new vertex at the second layer.

   The problem is that this graph can be enormous. To illustrate, for a 10×10 board, the number of possible states (not all of them are partial solutions) in the ninth level are 10^9. We don't want to draw or store this graph. However, we want to find a path from the first node (the empty board) to any

   Fortunately, when we use Depth First Search, we don't have to enumerate this graph. We just know how to transition from one solution to the next layer by adding a queen. The DFS recursion keeps track of the partial solution to date, and returns with a full solution. The problem is thus reduced to finding a path from the root node (an empty board with no queens) to a solution with n queens on an $n \times n$ board.

   Note the following:

- For an $n \times n$ board, a vertex in this problem can be characterized as a set of positions of queens on the board.
- Assume that one is at a vertex which is at level k in the problem, so that there are Queens on the first k rows. Then, the children of this vertex have first k queens in the same position, plus one additional queen in column $k + 1$, but only if the additional queen is placed in a location where it cannot be captured. Thus, one has to check whether that position in the next level for that queen is one that cannot be captured by any of the queens in the previous columns.

In the SCC site, in the `HWProblems/HW3_codes` you will find a main program `chessQueens.cpp` and an include file Chess_Board.h that defines the class Chess_Board. You have to write a method goodMove to check whether a square is free from capture by the existing queens in a partial solution. You also have to write a depth-first-search function to search for a solution to the queens problem in an $n \times n$.

Test your program in a small chess board (e.g. 5 by 5). Then, try it for sizes from 8 to 31 and verify that your program finds correct solutions.

In the web site, you will find a makefile. You can make the program using `make chessQueens` . Run the program as `./chessQueens >> Queens.out` .

Turn in your code and the Queens.out file.