Daniel Breitigan

Dr. Briggs

Artificial Intelligence – CSC 462

25/9/2016

N-Queens

The n-queens problem is a popular toy problem in computer science. The goal is to create an algorithm that places N number of queens on an N \* N chessboard where none of the queens are able to attack each other. This problem first became popular by focusing on the number of solutions for an 8 x 8 board, which is 92 solutions. The goal of this project was to create a program in Common LISP that was able to solve the n-queens problem using breadth-first search.

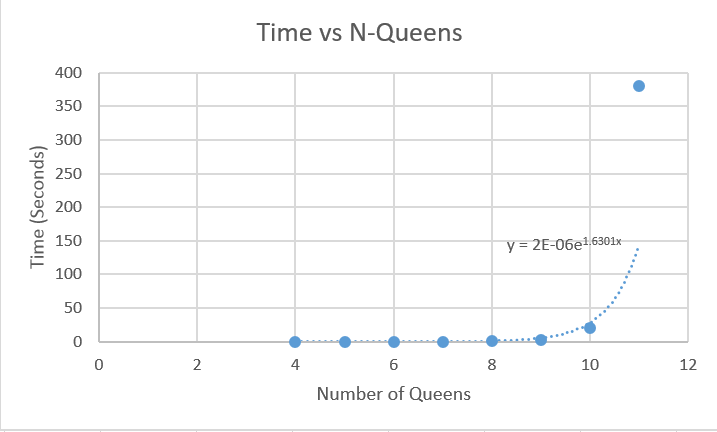
The results from the LISP program and the theoretical analysis of the n-queens problem are consistent. The program is able to properly see if a board is correct by using it’s VALID-BOARD? Function. This function looks at each piece on the board and compares it to the other pieces. It checks if any pieces are on the same x or y coordinates, and then checks if they are diagonal by finding the absolute value of the (x1-x2) and absolute value of (y1-y2) and checking if they are equivalent.

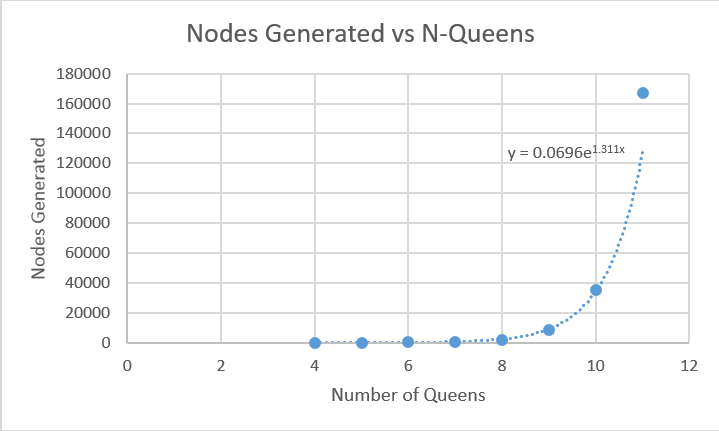
The EXPAND function in the program finds all the valid states from the inputted state and adds it to the NEXT list. EXPAND finds which queen has yet to be positioned by looping through the queens’ position list and finding the first queen with a y-coordinate of 0. It then sees which values that can be given to the y-coordinate without it causing an invalid board. If the board is valid, it is added to the NEXT list.

The final main function in the program is BFS. BFS takes an initial state places it into the NEXT list. It then generates its children using EXPAND and checks if the board is a solution. BFS continues to do this until either a solution is found or there are no more possible solutions in the NEXT list.

Since the program uses breadth-first search, all of the possible nodes should be generated before a solution is found. This is because breadth-first search uses the First-in-First-Out approach. Knowing this, the program will analyze all incomplete boards, which means that all possible nodes will be generated. The LISP program properly generates these nodes. This is known because there are 2057 valid states for an 8-queens problem, and the program generates 2057 total states into the NEXT list. The solution that the program gives is also always a valid state. For example, when the initial input is ((1 0)(2 0)(3 0)(4 0)(5 0)(6 0)(7 0)(8 0)), the output is

((1 1) (2 5) (3 8) (4 6) (5 3) (6 7) (7 2) (8 4)). This result is what is expected for a breadth-first search algorithm due to it being the first solution added into NEXT.

 The program runs very quickly when the number of queens is small, but quickly gets out of hand. For example, at 8 queens the program takes just under .5 seconds to run. At 11 queens, the program jumps up to taking 379 seconds to run. The formula that was given through excel to estimate the time it would take to run is , Where x is the number of queens. This is for multiple reasons. First, the program is not running in a fast environment. It is being ran on a virtual machine running Xubuntu, so it is not getting the full processing power it can. Also, the time for this program is going to exponentially grow due to the number of states it has to generate and explore growing. At 8 queens, there are only 2057 states to generate, but at 11 queens, that number is 166,926. Excel gave the formula to estimate the number of nodes that needs to be generated, with x being the number of queens.



The Average Branching factor for this problem is going to always be just over 1 when you go from initial state to the solution due to using breadth-first search. Since breadth-first search is searching all solutions the same depth as each other when the solution has to be at a depth of N, not only are all nodes generated but close to all nodes are explored too. For example, the average branching factor for 8 queens is 1.04 since 2057 nodes were generated and 1966 were explored, and the average branching factor for 11 queens is 1.02 since 166926 nodes were generated and 164247 nodes were explored. Since breadth-first search is being used, finding the average branching factor is not very important due to it always being just over 1.

It won’t take long for the program to take too much time to be reasonably ran. Based on the exponential function from excel, 15 queens would take about 1 day to run. At 17 queens, it will take over 25 days to run, and at 20 queens it would take over 9 years. Not only would time be an issue, but so would space. To keep things simple, let’s assume each node takes 40 bytes (1 byte per coordinate, so 40 bytes for a 20 queen problem). At 17 queens, 13 gigabytes of memory would be needed, and at 20 queens, about 679 gigabytes of memory would be needed based on the excel function. This shows how complex this problem can expand.

One easy improvement we could make to the program is using depth-first search instead of breadth-first search. Since the n-queens problem has all the solutions at the bottom of the tree, depth-first would more quickly find a solution and use much less memory, mainly due to not having to explore the right side of the tree. This could be easily done by appending the results of EXPAND to the front of the NEXT array instead of the back.

The n-queens problem is a fascinating toy problem that helps teach algorithms and think in unique ways. Common LISP proved to be a challenging but rewarding language to program in. The project had downfalls, mainly due to using breadth-first search at the searching algorithm, but was still useful for learning about problem-solving and applying algorithms to problems.

I Didn’t know if you wanted the answers to the problems from the Project or if they were just to test our functions, but here they are

Problems:

1. ((B (C (W W)) D) X (W W) ((W W) (U V)))
2. ((A (U V)) (V) (Y V V))
3. ((1 8) (2 4) (3 (7)) (4 3) (5 6) (6 2) (7 5) (8 1))
4. Create two of my own Problems:

* #1
  + S1: (Q)
  + S2: (A 6 3 (Q 2) (Q) (4 2)) ‘
  + S3: (D B)
  + Result: (A 6 3 ( Q 2) (D B) (4 2))
* #2
  + S1: (4 1 2)
  + S2: (4 1 2 (8 5) (6 (4 1 2)) (412))
  + S3: (DAN THE MAN)
  + Result: (4 1 2 (8 5) (6 (DAN THE MAN)) (412))

VALID-BOARD? Problems:

1. NIL
2. T
3. NIL
4. NIL

Final Code Test:

Input: ((1 0)(2 0)(3 0)(4 0)(5 0)(6 0)(7 0)(8 0))

A Solution was found!

Number of Nodes Generated: 2056

Number of Nodes Explored: 1966

Solution: ((1 1) (2 5) (3 8) (4 6) (5 3) (6 7) (7 2) (8 4))

Real time: 0.492931 sec.

Input: ((1 1)(2 4)(3 0)(4 0)(5 0)(6 0)(7 0)(8 0))

No Solution was found!

Number of Nodes Explored/Generated: 40

Real time: 0.020322 sec.

Input: ((1 4)(2 0)(3 0)(4 0)(5 0)(6 0)(7 0)(8 0))

Number of Nodes Generated: 270

Number of Nodes Explored: 254

Solution: ((1 4) (2 1) (3 5) (4 8) (5 2) (6 7) (7 3) (8 6))

Real time: 0.128647 sec.

Input: ((1 8)(2 1)(3 0)(4 0)(5 0)(6 0)(7 0)(8 0))

No Solution was found!

Number of Nodes Explored/Generated: 33

Real time: 0.023558 sec.