Knight’s Tour Report

Programming Assignment #1

Uladzimir Kasacheuski

2/21/16

Intro

The Knight’s Tour problem is a very interesting and enriching programming problem to solve. From demonstrating how effective heuristic rules are, to implementing your own data structures, to debugging mass amounts of fast paced data, there is a lot to gain from it and a great way to develop programming confidence and competency. The project is pretty straightforward, but the problem and implementation are intricate and at times challenging. It’s amazing how much a little logic can affect the speed of a program and how even though two methods may both solve a problem - digging with a twig and digging with a excavator are extremely different in their level of effectivity. The implementation I used to solve this problem is organized simply and effectively uses both Warnsdorff's rule as well as backtracking.

Background

The Knight's Tour problem is an interesting problem that demonstrates the value of heuristic rules when approaching problems. You are given a chessboard of size 8 by 8, resulting in a total of 64 positions. You are told you have a knight, it is sat at position (x,y), and you have to get it to each position on the board once and only once. At first sight, this is a tricky problem. Through a bit of googling you find that to a person, this problem is not as difficult as you may imagine. In fact, many people have solved the problem systematically from any point on the board, some even as early the 9th century AD. However, to a computer it’s a different story. This rock needs to be coded to do every step - and if the best you can do is brute force you’re going to be waiting a while. Even at 32 steps to go, depending on how your algorithm is set up some positions may take up a reasonable chunk of time. (Don't test 5,6 on my program unless you want to sit for half an hour - most of them work pretty speedily though). Starting from the start with 64 positions to cover? You may aswell wait until they come out with quantum computers and just write your program then.

Have no fear though, H. C. von Warnsdorf solved this problem with a little simple logic. He first recorded his heuristic rule in 1823 - long before the first computers were even considered. (At that time, they were only just starting playing with electricity). His rule states that given multiple positions the knight can move to, to solve the whole system, the knight should move to the position to which there is the smallest amount of available moves /from/. In other words, given multiple positions - the knight should move to the position that is hardest to get to. Which makes sense, if it's so hard to get to, you may not ever get another chance again. Simple, right? The beauty of it is in how effectively it works. With Warnsdorff's rule you can actually expect to solve the system every time from the first time - no backtracking required. That's a constant time solution for every initial position. No need to wait for your quantum computer after all. It's incredibly simple to implement as well. After finding all the positions the knight can move to, just calculate how many positions you can move to from each of those and pick the lowest one.

Project Description

This project required the solution of the knight's tour problem implementing a linked list, a stack, and a 2D array of size 8 by 8. The linked list was to be used in storing the user's input for initial conditions to be used to solve the problem. The stack was to be used for actually solving the system, meaning that the positions that the knight visited would be stored in the stack resulting in a chronological order that would be very conducive to backtracking when required. The 2D array was a graphical representation of the chessboard, which would hold numbers in every spot representing the order of the knights movement. For example if the knight visited spot (1,2) second, the number stored in that element of the matrix would be 2. (Initial condition is Zero).

When running the program, the first thing that is necessary is for the user to input initial conditions. The user must be able to input as many initial conditions as they desire. After they say that they have no more conditions they wish to add, they must be given the opportunity to remove entries, modify entries, and add more entries. Only after they have responded that they are done and are satisfied with the initial conditions can the software be allowed to solve each position. That is to all be done using a linked list you create.

The next step is then to solve each initial condition for the Knight's Tour problem. You are to use two methods to solve the problem. For the first 32 moves the movements are to be decided by Warnsdorff's heuristic rule. This is due to the fact that if you decided all 63 moves with the backtracking system, you would likely never finish seeing if your program even works. Recall that the knight has 63 moves to make and at each position he will have at worst 8 positions he can move to. A really rough estimate would put that at 63^8 - which does not account for positions that can not be moved to - and equals 2.5 * 10^14 possible paths the system can be tested for and of which only one will lead to a solution. For the second half of the moves the knight makes, you are to use the backtracking method. In this method you will try each route until you find the solution or that there are no more moves to make, then back out, and try another route. At 31 moves left to make, this problem is solvable at a workable time frame.

Project Implementation

The implementation I created works simply. The main.cpp file pulls together getting the user's input, solving every initial condition, and outputting the chessboard as the result. Getting the user's input is handled by the aptly named cUserInput class. It uses the dsLinkedList class as the linked list and stores every initial condition in it. The user input class runs loops when getting the users input so that if the user inputs something that was not expected they will be asked to try again. (EG “x” when expecting (y/n)). Then it allows the user to modify, remove, or add more initial positions after they say they have no more to add. After the user has responded that they are satisfied with their initial conditions, the cUserInput class method that was called on returns the dsLinkedList containing the initial positions back to the main() function. The main function then passes the dsLinkedList with initial positions to a function that then loops through every initial position and passes it to the cKnightClass, which then solves the system for the initial condition and returns a filled out cChessBoard object, which is then used to output the chessboard matrix containing the solution.

The cKnightClass is the class that contains all the logic behind solving the system given any initial condition. Upon initializing the object, it expects the initial position to be passed as arguments to the constructor. At this time, it creates a new chessboard (cChessBoard) for itself, sets 0 as the initial position, and adds the initial position to the dsStack. The dsStack is built off of the dsLinkedList, with only minor extensions. Popping, Pushing, and returning int* position from the top value of the stack were added to the methods that dsLinkedList has. Each of those methods implement methods that dsLInkedList already has. After the cKnight object is initialized, its method .solveTheSystem() is called and cKnight gets to work. It runs a loop that keeps going while the length of its stack is less than 64. For the first 32 iterations the knight runs Warnsdorff's rule and builds the stack up. It is implemented exactly as was described in the background section of this report. The second half, 31 iterations, is a bit tricky. For this section the knight runs the backtracking method and listens if it returns false. If it does, then it knows that backtracking is required - which triggers a function that removes the last move from the stack and the chess board and returns the position from which the knight back tracked from. This position is then passed to the backtracking method next time it is run, and the backtracking method then picks the next position after that position in the options it has to move to. In this way, we iterate through every position. In debugging I had a function print out every position the stack held after index 32 at each iteration of the loop. On difficult positions, such as (5,6) it ran for a long while and thus created a very interesting and informative visual. It was a very cool experience and felt like I was hacking into the matrix. This function is not turned on in the final code as it would make the output print out extremely long.

Comments

A few parts that made me exceptionally nervous resulted when testing how the backtracking system worked without having a good way of seeing the process as opposed to just seeing the result. This was overcome by playing around and analyzing what exactly I was trying to understand. At one point I analyzed the data and actions on a very microscopic level, outputting the stack level and every change that happened to both the stack and the board to figure out why the backtracking was not adding to the stack. It ended up being a tiny bug that was easy to fix - still back in my linked list structure. Later I had to take the exact opposite approach and analyzed the activity at a macroscopic approach. At this point I had the system print out the list of positions moved to in order starting at move index 32. This was to analyze which path of the many available the knight was on to make sure that the backtracking was looking through every available path and not skipping. This was a very cool look. On the right end of the screen were root nodes that changes once every 5 minutes, and on the left it was almost like counting microseconds, changing faster than the eye can understand. In summary, after taking the time to develop the proper methods of analyzing the system, everything became easy to understand. That is the power of debugging.

Besides this, creating my own linked list and stack was a very cool experience. I’ve been programming for a few years in php and javascript - making some cool web applications - and never even thought about how the data structures I was using were implemented or even worked. By creating my own data structures not only am I becoming free to create data structures in the future that will optimize my software but I will also be able to gain a deeper understanding in what is actually happening behind the code I write in my software - enabling me to more effectively leverage the implementations for speed and reduced complexity.

Conclusion

The main take aways from this project are include having a deeper and enriched understanding of Stacks, Linked lists, and Arrays and how they interface with the computer's memory as well as understanding the value of heuristic rules and how a bit of critical thinking and surprisingly simple logic can make an incredible impact. This project was a great test of current knowledge and exercise in data structures, logic, and debugging.