COSC343 Assignment 2: n-Queens-Problem

Cameron Milsom

May 14, 2014

1 Introduction

To start off with, I decided to program it in python. This choice was purely because I was already learning ruby, but it wasn't specified in the assignment so I went with python instead. In my assignment I came across a few issues. The main issue I had was due to poor lack of planning and foresight on my part.

2 Planning

I huge mistake that went into the assignment was lack of planning. In my head I had an idea of exactly what I wanted to do, but didn't put much thought into the consequences. I originally spent a large amount of time designing a N-queens solution that focused on the positions of the individual queens rather than the position of all the queens on the board. In my algorithm I was finding the perfect fitness of the queens position based on how many collisions it has with other queens. This was leading to a program that was finding queens in the same vertical position. This was the opposite of what I wanted. The code I had written was vaguely correct and a long the right lines, however the general approach was a bit convulted (more than my current code) so I decided to re-approach the assignment from the beginning.

Algorithms

Initializing

I start creating my boards by making a list of random board (where each queen is on a seperate x-axis. These boards all contain a board key which represents all the positions of the queens on that particular board.

Repopulating

This works by taking the list of board, making an intermediate generation of the list by using selection. Then using a crossover mechanism to create the new generation. Its worth noting that there is also a mutation mechanism built into the crossover functions. Finally the population is checked for a perfect solution. This process is repeated until a perfect solution is found and then it is completed.

Selection

I decided to go with Roulette Wheel selection. The roulette selection is designed so that it favours boards that have lower fitness values. The variation in fitness of the boards is not so significant that a particular board will be selected for the majority of the population. This makes it a decent fit for my genetic algorithm.

Crossover

For my crossover function between parents I tried multiple functions to see if there would be any significant effect. However, the program doesn't run at a speed fast enough to easily record a large quantity of values to find what algorithm is best. To solve this, I tested these algorithms on a slightly lower n-value which can find solutions at a significantly faster rate.

What I found was that single crossover was the code that took the least amount of generations. Using the 5-queen problem, I found that over 1000 solutions found, I was averaging 18 epochs for single, 26 epochs for double, 19 epochs for uniform. The results are interesting as I wouldn't expect such a significant difference between single and double. It is dissapointing however that uniform was almost the same as single.

Single Point Crossover

The single point crossover takes a random point from the two parents and makes a child that is a combination of both.

Double Point Crossover

The double point crossover takes a random section out of one of the parents and replaces the middle section of the other parents to make a new child.

Uniform Crossover

The uniform crossover goes through each position of the binary board key and chooses randomly which parent it takes that value from.

Mutation

There is also a mutation mechanism that goes through the board key and at each index (representing the x, y value) has a chance of modifying it. After continuously modifying this number, I found that the best result was roughly 1/100 chance.

Restart

While my code never needs to restart as it seems to always find a solution. I decided to implement a restart algorithm. This required some playing around with to test which values result in better performance. I started with a value of around 300 generations of the population before killing and making a new population. This gave me better results than when I was leaving it indefinitely. After some further testing I found out that it's best to have a value of around about 500. Often it would complete in less than 500 epochs, so reseting it around this point enabled to skip the long times being stuck on poor board choices

Other Performance Enhancers

There were some other techniques that I used to try increase the performance of my n queen solution finder.

Graph

I wasn't able to insert the graph into the pdf. But its there! mygraph.png The graph that was produced wasn't particularly interesting. The graph shows that the most optimal solution rapidly changes over the first 50 generations and then it doesn't change much till the perfect solution is found. This is because of my fitness having only so many possible 'fitness' values rather than truly being [0,1].

Future Improvements

Original Design Mistakes

There are definitely improvements that I would like to have tried. One of the issues with my board implementation is that it will only function up to and including a 10-queen problem. This is because I represent my boards as strings. I chose this way because thats how I had previously approached this task in the lab. However if I used a list, I feel there would have been some performance increases as I had to often convert the string into a list to mutate/modify and then back again to a string to recreate the board object.

Selection Process

One approach I was interested in trying but never completed was the Tournament Selection process. This would have been an interesting selection choice as it's possible to change the size of the tournament. This allows for a 'selection pressure' where we can change how likely it is for the less fit boards to be choosen.