CSCU9A3 Practical - Searching for a Solution

Introduction

This practical is aimed at giving you an overview of the following two search methods:

- Greedy Search
- Genetic Algorithms

We will look at two different problems, one of which is amenable to a greedy search/hill climbing approach and one that is not. The first problem is concerned with trying to guess a secret binary pattern where the only information available to the solver is how far the test solution it has generated is from the secret pattern. The pattern is a grid consisting of 1's and 0's so this distance is measured as the number of bits that differ and is often called the Hamming distance between two binary patterns (named after Richard Hamming who noted the use of it as a measure).

The second problem is known as the 'Travelling Salesman Problem' or TSP for short. The challenge here is to work out the shortest route that enables you to travel to all locations on a map while only visiting each location once. There are many variants of this problem that occur in real life, particularly with regard to transportation and logistics.

Before we start, consider which of these two problems is solvable using a greedy approach and which is not.

Source Code & Problems

The source code for this problem is available in the directory:

\\Wide\Groups\CSCU9A3\Practicals\Practical6

Copy all the files from this directory into a suitable location in your own file space. Create a BlueJ or Eclipse based project that contains the Java files in the copied directory.

You should find the Java code for Pattern.java, TSP.java and Place.java. The class Pattern is used to create a target pattern for the search algorithms to look for and is also used to store the 'genes' for each potential solution in our Genetic Algorithm (GA). *Read through the comments and code for this class and try to get an idea of what it allows you to do.*

TSP.java and Place.java provide code to store a set of locations and work out a score for trying to travel between them. The score includes a penalty for not visiting all the places and also for visiting a place more than once. *Look at Place.java first then step through TSP.java and try to get a reasonable idea of what the methods that it provides can do*. You do not have to understand in detail how they do what they do but at least read the comments so you can see what they are doing.

Pattern Matching

We now have enough pieces to try and solve each of our problems with different algorithms. The first thing we will look at is trying to solve the pattern-matching problem using a Greedy approach (this will

also confirm if your above answer was correct). The code for calculating a Greedy solution is in Greedy.java and the methods that call this code are in the solvePattern method in Solver.java.

Start by looking at the code in Greedy.java and in particular, the greedyPattern method (you can ignore the other methods for now). Now run the greedyPattern method via the solvePattern test method in Solver.java and see how long it took to work out what the pattern was. Make sure you understand why this is a greedy approach and why it found an answer in this case. What does it tell you about the solution space of this pattern-matching problem?

We will now compare the greedy approach with a Genetic Algorithm and see how it would try to solve this problem. The code for this is in the evolvePattern method of the class GA.java. This method is called via the commented out code in solvePattern in Solver.java. *First read through the code and comments in the evolvePattern method and try to get an understanding of what they are doing, then uncomment the lines in solvePattern and see how long it takes to produce a solution*.

You will probably find that the GA gets down to about a 25% match against the target pattern and then takes longer and longer to find a better solution. See if you can get a better result (a score close to 0) by changing the following parameters that control the GA:

- Number of generations – This controls how long the GA gets to try to solve the problem
- Population size - A larger population samples the solution space better but involves more evaluations
- Tournament size - The larger the tournament, the more chance there will be that you will select the current best solution and just keep exploiting it leading to less exploration.
- The mutation rate – This directly controls exploration but set it to high and you end up just doing a random search.

Each of the parameters are defined at the start of the evolvePattern method with their initial values in comments in case you need to reset them. You may notice that you can get an improvement but there will often be a trade off in the time it takes to find it. To make the GA explore more, push up the mutation level. To make it exploit more, increased the relative size of the tournament. Increasing exploration will come at a cost of taking longer to find a solution in the end whereas increasing exploitation will produce behaviour similar to hill climbing. Ideally you want to find the best solution in the least amount of time.

You will have probably guessed that this is not a particularly good problem for a GA because it is not very quick at producing small refinements (it can do it but it takes time). The GA will often get a rapid initial improvement since it spans the solution space with its population. However, it will then take longer and longer to finish off the puzzle. Strangely enough, a GA is actually better at solving a complex pattern-matching problem where the solution space is not smooth and convex.

The Travelling Salesman Problem

The next problem we are going to look at is the Travelling Salesman Problem (TSP) that will attempt to find an optimal route travelling around some places in Scotland (see overleaf). We will first look at a Greedy attempt at solving this problem and then compare it with our GA. The code for the Greedy solution is in the method greedyRoute in Greedy.java. This code repeatedly tries to find the best local improvement, takes it, and then looks for the next improvement. It stops when it can find no further route swaps in its current solution that produce an improvement.

First comment out the @Test line above the solvePattern method in Solver.java and uncomment the @Test line above the solveRoute method in Solver.java. This will stop the pattern solver code being called in our tests and switch on the solveRoute test. You will notice that most of this method is commented out with just some code at the top available to show you how to test a given route. Look at the positions of the locations in the file locations.pdf and try entering your own route into the array 'route' that is declared in solveRoute. The values in the array refer to the lines in the file locations.txt if you started counting lines at 0. See if you can come up with a solution that is less than 400 in distance for the complete trip and which includes all locations only once.

Once you have tried this, uncomment the first block of code that works out a solution using the greedy approach and examine the solution it comes up with (use locations.pdf to check the route). Is this route any good – did it do better than you?

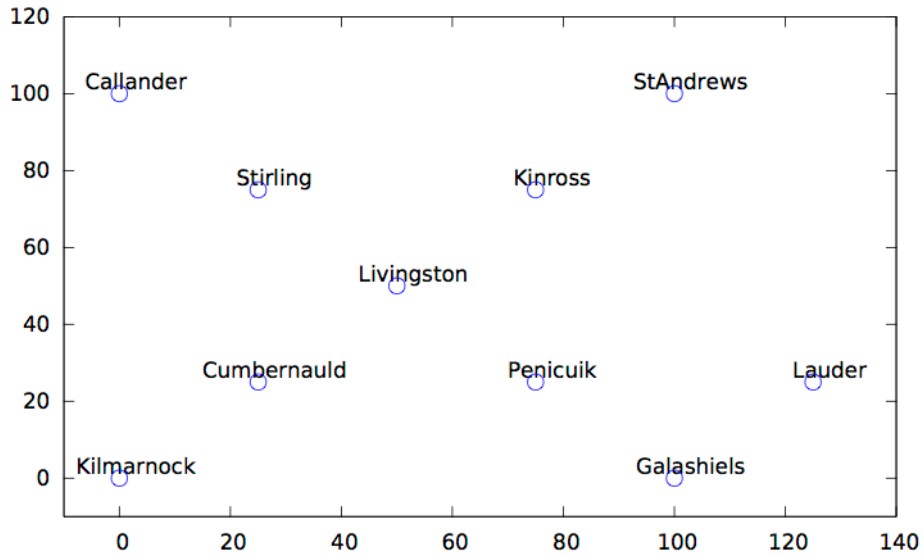
Now lets look at the Gas efforts at solving this problem. The GA code that tries to solve this problem is in the evolveRoute method in GA.java. This is very similar to the previous code except we need a new method for producing parents that needs to take account of the fact we should only visit any location once and that we must visit all locations. Read through this code then go back to solveRoute in Solver.java and uncomment the final block of code that creates a GA and runs it. Try running the solveRoute test a number of times and see if you get better or worse results. Because the GA involves random behaviour, it does not do the same thing on every run. If you run it long enough, however you should find it will produce a solution that in this case cannot be beaten. Try changing the same GA parameters that you did in the pattern matching case and see if you can get the GA to find a good solution in less time.

This class of problem is much better suited to a GA – it involves a deceptive solution landscape where sometimes the solutions that are generated have to get worse before a better solution can be found. A further point to note here is that the code for the GA hardly changed at all. It just needed to know how many genes (problem values) it had to store and what the score was for a given solution. For the greedy approach, we need to write a new algorithm that is specific to our problem. Most of the effort that goes into producing a solution with a GA focuses on correctly scoring solutions and working out what the population, tournament size and mutation rate should be so that we can efficiently search the solution space.

For more information and examples of these techniques, go to the Resources page on the module web site (https://canvas.stir.ac.uk/courses/961/pages/programming). At the top of this page is a reference to a book by Melanie Mitchell which provides a gentle introduction to the area. The Meta-Heuristics book is a lot more in depth and provides an insight into some of the mathematics behind these ideas.

For now, have a look at some of the youTube videos of evolving artificial creatures (some with Neural Network brains). These provide visual examples of GAs and Artificial Intelligence at work.

Appendix: Map of the Locations



Note that locations are only approximate to keep the coordinate positions simple. You could provide your own locations and coordinates if you wish by creating new entries in the locations.txt file (please make a copy of the original locations.txt file). Each entry in the map should be on a single line and should comprise a location name (without spaces) followed by an x and a y coordinate. Each of these items should be separated by a Tab character (not spaces).