

EVCO Open Assessment

Y6375144

December 28, 2013

Chapter 1

Introduction

1.1 Genetic Algorithms

For some problems it can be difficult to find an optimal solution. This is because the search space grows at a much faster rate (exponentially, for example) than the input space. The Travelling Salesperson problem is a perfect example of this - for a search space with n cities, there are $n!$ paths that the salesperson could take.

Evolutionary algorithms do not present a solution to this problem. However, by taking inspiration from biological evolution, they do offer an alternative approach that will often lead to a ‘good-enough’ solution in a much more reasonable number of calculations.

Genetic algorithms are a type of evolutionary algorithm that typically represent population members (i.e. candidate solutions) in bitstrings. The population of strings is repeatedly updated through the following process:

1. Select - A number of the existing population members are selected to continue through to the next round. Typically these would be the fittest members in the current population.
2. Crossover - To generate new population members, existing ones are created by crossing two (or more) population members over. See figure 1.1 for a visual representation.
3. Mutate - The population is mutated at random. This typically involves passing over every bit in a string, and with a random probability flipping the bit from a 0 to a 1 (or vice versa).
4. Evaluate - A fitness function is run on each member of the population, giving an indicator of how ‘good’ a solution is.

Genetic algorithms hold much potential, but in order to get the most from them you must carefully choose your representation and parameters. The representation is the bit string typically, but it can also take other forms. There are many parameters that can be fine tuned, such as the number of offspring to breed (at the crossover stage), or the type of crossover to perform.

1.2 The Problem

Morpion Solitaire is a one player game that is typically played on grid paper. To start, a cross shape is drawn in the centre of the page (as shown in figure 1.2). On each turn the player must add one cross that creates a line of five crosses either vertically, horizontally or diagonally. The lines may intersect, and in this particular variant the lines may also overlap (it could be argued that this makes the game easier). Figure 1.2 shows

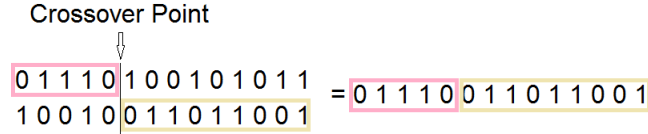


Figure 1.1: The child is made up of the top row (which represents the first parent) until the crossover point, where it then takes from the bottom row (the second parent).

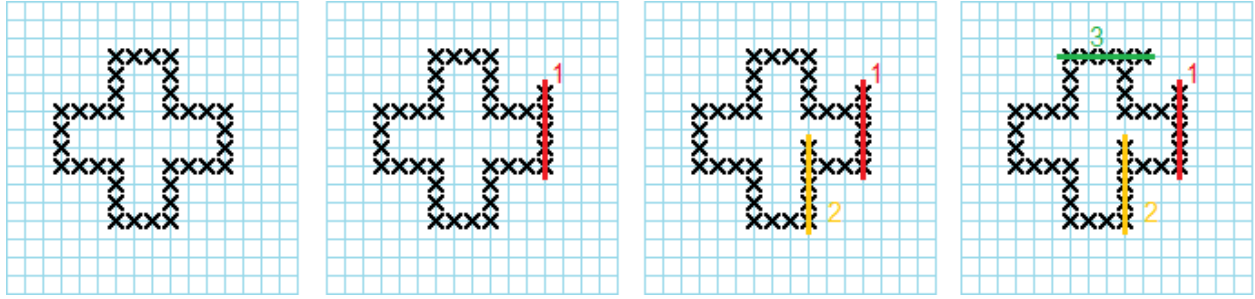


Figure 1.2: The initial state of the game on the left, and from left to right the first three moves are shown. The lines don't have to be numbered, but for clarity they are here.

a possible first three moves. Finishing at this stage would give an overall score of 3, as 3 lines of 5 crosses have been formed.

In this report I will detail my investigation into obtaining a high score through the use of a genetic algorithm. Working out the fitness of a population would be a tedious process if I have to take each population member and draw out its game from scratch. Thankfully that is not necessary as I have been provided with a computerised version of the game that is slightly faster.

When designing my genetic algorithm I will have to consider the format that the digital version takes its input by. The program represents positions on the grid by their (x, y) co-ordinate positions, where $(0, 0)$ is the bottom left of the grid. For each move, four parameters must be passed into the program in the format $x_1y_1x_2y_2$, where (x_1, y_1) is the position of one end of the line to draw, and (x_2, y_2) is the other end of that line. For the game shown in figure 1.2, the input would be:

```
linuxMsolitare.exe 12 11 12 7 9 9 9 4 6 13 10 13
```

The program accepts a maximum of 2000 inputs, or 500 moves. Given that the current best known solution is 178 [?], this is unlikely going to be a problem. One potential problem is that to evaluate a population of size n , the application will need to be called n times. Even with a decent processor, this could be a slow task for any sizeable population. Also, if a representation is chosen that doesn't directly map to the program's inputs is chosen, then each population member's representation will have to be converted to the correct format before the program can even be called. Again, this could be time consuming, but solutions to these problems will be considered in the next section.

Chapter 2

The Solution

2.1 Representation

The standard genotype in a genetic algorithm is a bit-string. Studies have shown that in a lot of cases the algorithm will perform better when a bit string is used [?], although other studies suggest that this is not necessarily the case [?].

I will now discuss possible representations, and then explain my choice of representation.

2.1.1 Gray Code Bit-string

Binary bit-strings are not ideal for a genetic algorithm as a single bit flip during the mutation stage can cause the number to change a lot. Consider the bit string 10010110 that represents the 8 bit unsigned integer 150. During mutation, if just the first bit is flipped, the string becomes 00010110 and now represents 22. If the search space is a 256x256 grid, and that integer represents one co-ordinate, then 150 to 22 is a large jump. It would be more useful if the co-ordinate changed by just a small amount. If we wanted to change to 149 (10010101), two bit flips are required.

One solution to this is to use Gray code. This is a bit-string where each consecutive value differs by only one bit. In Gray code, 11011101 represents 150, and 11011111 represents 149. That's just a single bit flip, making Gray code ideal for genetic algorithms.

There are now a few possible options for how Gray code could be used to represent an individual move, and then a series of moves. The first option is to represent the co-ordinates as they are passed into the program in Gray code (remember that a move is passed in to the program as $x_1y_1x_2y_2$). The grid 40x40, so ($\log_2 40 = 5.32$) 6 bits would be required to represent each co-ordinate. Each move would use 24 bits, and if the maximum number of moves is played each time (500), then 12000 bits or 1500 bytes will be required for each population member. If we have a population size of 5000, then the total required space will be 7.15MB.

The second option is to store each move as a combination of (co-ordinate, direction), where co-ordinate is the new cross to draw on the board, and direction is the direction of the line. So ((20, 20), left) would mean draw a cross at position (20, 20), and draw left. This would then be translated into the program's input format $x_1y_1x_2y_2$ to be 20 20 15 20. Again, 6 bits would be required to represent each co-ordinate. As there are 8 possible directions, ($\log_2 8$) 3 bits are required to represent the direction that the line should be drawn in. Therefore each move would use 15 bits, giving a maximum of 7500 bits or 938 bytes for each population member. With a population size of 5000, the total required space will be 4.47MB. This is a good saving on the last representation described. However, computers nowadays have a lot of memory, and the

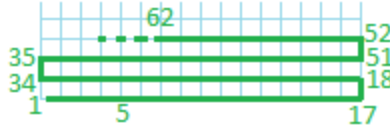


Figure 2.1: An alternative co-ordinate representation that starts at the bottom left of the grid numbering each position as shown.

relative difference between 4.5MB and 7.2MB is not huge. There is also the overhead to consider when the representation has to be converted into program input. The first described representation would be the fairly simple task of converting Gray code to an integer. With this representation, the overhead would be slightly higher, as the start position and direction would need to be considered in order to calculate the program input.

The third and final possible representation that uses Gray code does not use the co-ordinate system as it has been previously described. Starting at the bottom left and moving right, each position will be assigned a single integer. When the end of the row is reached, the above position will be numbered, and then on that row moving left the positions will be numbered. See Figure 2.1 for a visual explanation. As the grid is 40x40, 1600 positions will be numbered. Describing a move will be done as above, with (position, line direction). The amount of storage space required will be $(\log_2 1600)$ 11 bits for the position, and 3 again for the direction. This uses 14 bits per move in total, and sticking with the already used example of 500 moves per population member and 5000 population members will use a total of 4.2MB. A slight saving on the previous representation, but again the overhead increases as we now also have to convert between the new position representation as well as the direction representation. Another disadvantage that comes with this representation is that it is more difficult during the mutation stage to move a move in a particular direction. If you wanted to mutate the population member by changing one of its moves to start two rows above, you have to perform a relatively complex calculation in order to work out the number assigned to the position two rows above. With the above two representations it is easy to convert the Gray code to an integer, add 2 to the y co-ordinate, and then convert back to Gray code. Even better (although not necessarily easier to code) would be to perform the arithmetic in Gray code.

2.1.2 Integer Representation

The natural input to the program is a set of integers. Using any other representation internally requires that a conversion takes places before the fitness of the population can be calculated. Because of this, it's worth weighing up the pro's and con's of using an integer representation.

To understand a big reason why an integer representation might not be an awful idea, consider the reason for using binary / Gray code. Consider the move 20 20 24 24 that draws a diagonal line from the centre of the grid to the position (24, 24). If we were using a binary representation with 6 bits for each co-ordinate, the move would look like:

```
010100 010100 01100 011000
```

We then perform a mutation operator on the representation that has a 10 % probability of flipping each bit that it comes across. It outputs the following:

```
110100 010101 01100 011100
```

This is the same as 56 21 24 28, an obviously invalid move because the board is limited to 40x40, and also because no the co-ordinates don't represent a lines that is 5 positions long.

Using an integer representation then the mutation operator would know what changes could be made to a group of integers that would still be valid, and only perform these. For example, with 20 20 24 24, it might decide to shift down by 2 rows, and so it would subtract 2 from the y co-ordinates. Of course there is no reason why this can't be done with a binary representation - using binary arithmetic 000010 could be subtracted from each x co-ordinate, but using binary provides no advantage over using an integer representation, but does include the disadvantage that when used in the 'pure' genetic algorithm sense (i.e. flipping bits at random), then there is a high probability of creating an invalid move.

To confirm that using an integer representation doesn't disadvantage the algorithm, we must also consider crossover. Consider the two

2.2 Selection

2.3 Crossover

2.4 Mutation

Chapter 3

The Results

3.1 Conclusion