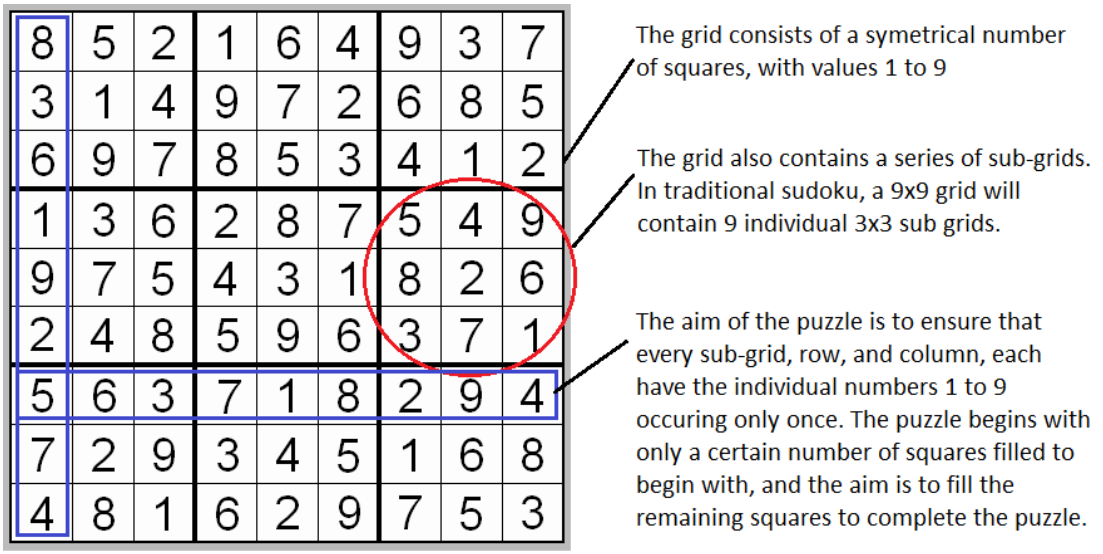
CP3064: AI for Complex Problem Solving
Genetic Algorithms: The Sudoku Puzzle

Introduction

The puzzle which was decided to be solved as part of this assessment was the Sudoku puzzle. In brief, the aim of this puzzle is to successfully fill a grid so that each row, column and containing sub grid each has a single instance of a number. The number range is defined as one through to the maximum dimension of the grid (so a 9x9 grid would be filled with numbers ranging from 1 to 9 inclusive). An example of a filled Sudoku grid is shown below:



A series of different factors should be considered when choosing an appropriate solution using a genetic algorithm. The first is in regards to formatting. Knowing how the grid data will be stored within a chromosome is fundamentally important when considering how the grids can and cannot be manipulated in order to find an optimum solution. Another fundamental consideration is the choice of genetic algorithms available, and the selection of suitable candidates. The usage of a single genetic algorithm or a selection of multiple algorithms needs to be considered. The major two concepts which need to be considered are the usage of crossover, and the usage of mutation.

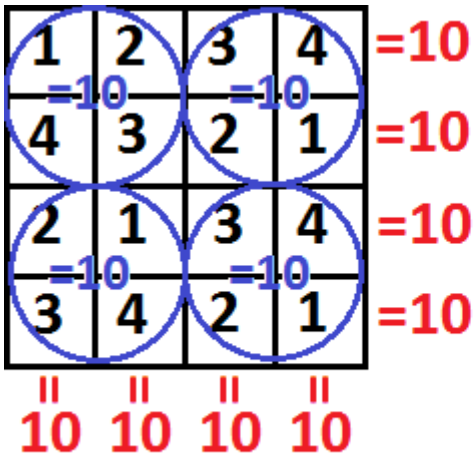
Solving the puzzle will be accomplished incrementally. First an appropriate method will be defined for solving mini Sudoku puzzle. This puzzle will consist of a 4x4 grid, and hence the level of complexity will be lower. This will allow the creation of an adequate solution in order to form a basis for the larger Sudoku puzzle. Once this puzzle has been analysed, the method will be scaled, and any improvements and optimisations will be added to ensure that standard Sudoku puzzles can be solved as efficiently as possible.

Task 1 & 2: Mini Sudoku (4x4)

In order to successfully understand how best to solve the 9x9 Sudoku, It's best to begin with a smaller problem. The first things that need to be defined are the various constraints which exist within the puzzle. Knowing these constraints allows various checks to be made during the fitness calculations. The fitness needs to be determined so that the optimum solution can be found, and in the case of Sudoku this is very important. It's important because in Sudoku there is very rarely any more than one solution per puzzle. Therefore the optimum solution within the search space is much harder to locate, meaning that the fitness function requires the checking of as many constraints as possible in order to find the solution. The potential constraints which have been identified are listed as follows:

- An individual row may only have a single instance of a number between 1 and 4 inclusive.
- An individual column may only have a single instance of a number between 1 and 4 inclusive
- A sub-grid may only have a single instance of a number between 1 and 4 inclusive
- The total of each row should equal the value '10'
- The total of each column should equal the value '10'
- The total of each sub-grid should equal the value '10'

Of the above constraints, the first three are useful constraints, as it's not possible to have a correct solution whereby those constraints are not met. The remaining three constraints, while initially appear be useful, provide an issue. If used in conjunction with the first three, then the there is no immediate problem using these constraints as an additional check, however using those on their own will produce problems regarding duplicate values. This is because it's possible to achieve a total value of 10 using the numbers 1 to 4:



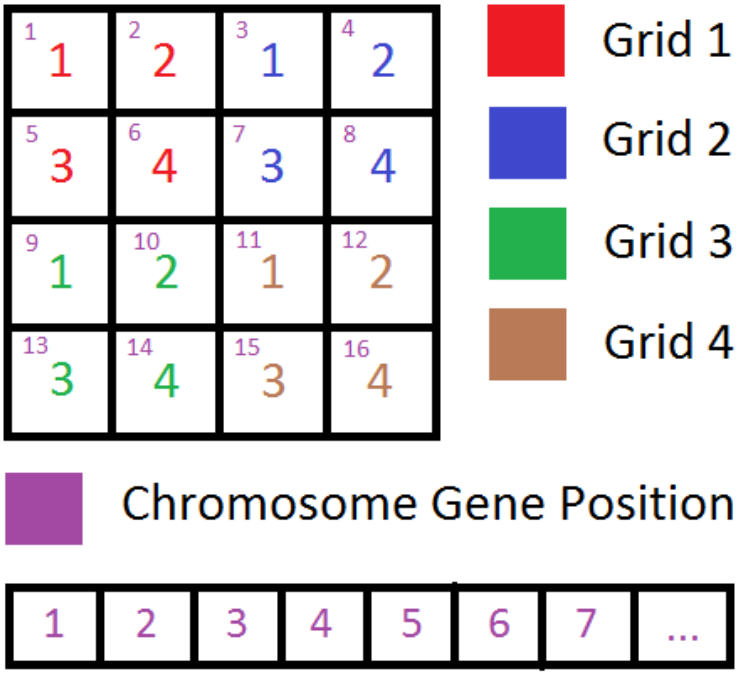
As shown left, it's possible to total 10 in each the rows, columns, and sub-grids while still having an incorrect solution to the mini Sudoku puzzle. Using only the totals as constraints is not adequate on its own.

The creation of the fitness function should therefore take into consideration all constraints mentioned, in order to be effective at generating an optimum output.

Analysis

It was decided that the best way to solve the puzzle was by using an adaption of a process known as simulated annealing. This process is useful for finding a global maximum in a large search space, and although the search space for this smaller Sudoku puzzle is relatively small, it should prove highly effective for the larger 9x9 Sudoku puzzle. The adaption comes in the form of removing the temperature aspect of the annealing process. This means that the genetic process involved is simply pure mutation, and no crossover.

The process is performed by mutating genes on a sub-grid basis. First it's important to explain the encoding of the chromosome, shown in the diagram below:



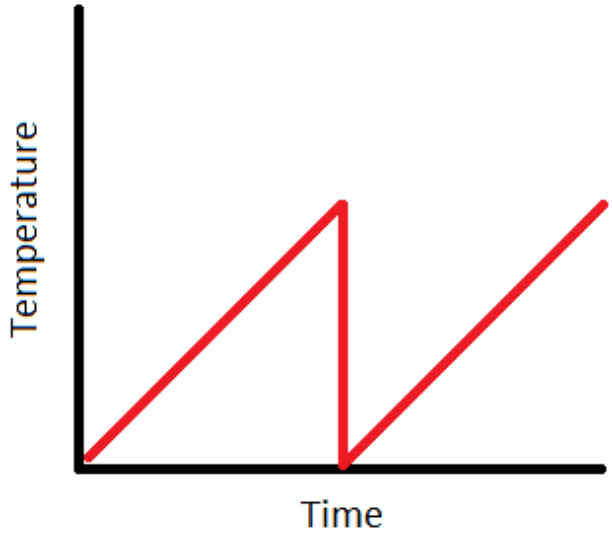
Each sub-grid is indexed using a set of index values attributed to that particular grid. Java starts indexes with the value, so each of the grid order indices , as well as the chromosome gene indices will need the value 1 subtracted in order to find their correct index value. The red sub-grid above can therefore be accessed using the index values {0,1,4,5}, the blue grid using the values {2,3,6,7}, and so on. This allows manipulating the data within these grids to be easier, and allows custom grids to be created if further development is requested.

With the encoding determined, it's necessary to determine how to accomplish mutating the values so that the constraints of the puzzle are left intact. There is an important constraint to consider, that if done so allows the integrity of the puzzle to remain intact. The puzzle can only be complete if each of the sub-grids, rows, and columns all contain unique values. This means that if one of them constraints is kept constant, and the puzzle is manipulated around that constraint, than the puzzle will keep it's structural integrity during the mutation phase. So, if the Sudoku grid is first initialised as shown in the previous diagram, and any fixed values are swapped with other values in the same sub grid, the initial starting grid will already meet the conditions for one of the constraints.

Now that the initial grid is created, mutations can occur for each sub-grid, by swapping the values contained within each sub-grid. This mutation ensures that each sub-grid contains unique numbers, and allows the fitness function to determine after each mutation whether or not there are accurate or inaccurate row and columns containing unique values. The mutation rate for this smaller example is set simply as 1, due to the small size of the search space. Similar to simulated annealing, the process of evolution is simply; if the fitness after mutation is better then the previous population, replacing the current population with the better one. This means that until a better solution is found, the solution will be mutated until a better match is found. Because the search space is relatively small, this is done pretty quickly. Optimisations will need to be made when moving on to the larger Sudoku puzzle.

Task 3 & 4: Sudoku (9x9)

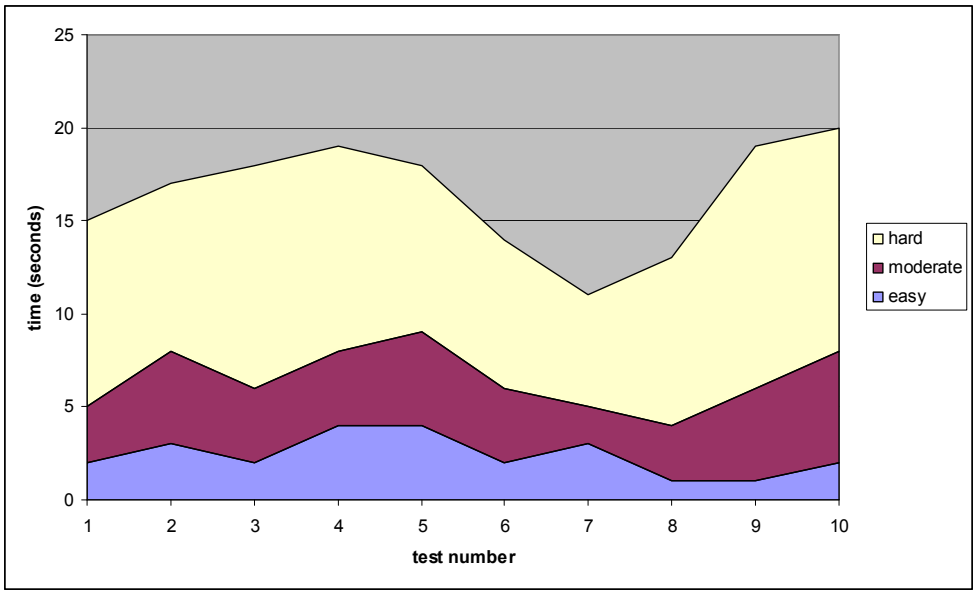
The task of upgrading the mini Sudoku puzzle to work with a full size puzzle took several steps. The first was that clearly the original 'pure mutation' method alone wouldn't be sufficient in pin pointing the correct solution. This seemed to due to the size of the search space, and the number of solutions which existed in the search space that weren't the global optimum. For example, the algorithm would reach near completion (2-3 correct numbers from the optimum), but would be unable to find the sufficient mutations necessary in order to reach the correct solution. Simulated annealing provided better results, by applying a higher rate of mutations to act as the temperature, and then decreasing them at a specified rate until the solution was found. Due to the method I've used regarding random numbers, the temperature is increased as apposed to decreased. This is just a simple reversal of thresholds to allow the effective use of random numbers:



As the temperature increases, the chance of a mutation occuring regardless of fitness is decreased. Once the temperature reaches its maximum, it's reset. This allows the search space to be searched again, regardless of the fitness of the current solution

The reset was necessary due to the number of times that a solution would 'cool' at a global optimum that was not the ideal Sudoku solution. Epistasis was a definite concern when working with Sudoku puzzles, as the solution cannot be correct unless the previous mutations are also correct. The combination of mutations through trail and error was indeed effective at finding a different number of optimum solutions, however because Sudoku puzzles generally only have one solution, the global optimum was very difficult to locate quickly.

The algorithm was successful in locating the solution, given enough time, but for harder solutions the algorithm is unable to locate the desired solution in anything under 8-10 seconds. I tested the algorithm using 3 Sudoku puzzles of 'easy', 'moderate', and 'hard' difficultly level using a Sudoku application generator that I have installed on my PDA:



The results, based on a sample of ten tests for each difficulty level, don't appear to show a linear link between the difficulty of the puzzle and the amount of time the puzzle takes to be solved. Clearly the amount of time increases overall, however the data shows that the likeliness of getting the results desired for the difficulty level are entirely random to a certain degree.

In general, the algorithm is far from optimum for the given situation. The above results show that the process of finding the optimum solution is still too unrefined, and essentially 'random'.

Additional Notes

I conducted a large amount of research into attempting to improve the algorithm, however I did not have the time to implement such findings. Instead I'll attempt to explain my findings, and also the ways I wished to implement them. I'll also mention some information regarding how I thought of ideas regarding my initial design ideas.

Quantum simulated annealing appeared to be a potential possibility in helping to solve my dilemma of premature convergence towards an unwanted and non-perfect global optimum. I read an article (Perez, M) regarding the usage of quantum 'tunnelling' alongside simulated annealing, allowing the search space to be initially searched very quickly by starting with a large number of mutations per iteration, and then decreasing them, similar to how the temperature operated in the annealing process. I attempted briefly to implement this, however found that for one reason or another the search time for finding the solution would increase, rather than decrease. I also attempted to essentially force the concept to work by alternating the number of mutations between low and high values per iteration. Initially I believed to see an improvement in the speed at which the solutions were found. However it appeared that those improvements were clearly flukes in the test data, and were proved as such when further testing was conducted. Another area I was considering looking into was running multiple instances of the same Sudoku grid which needed solving, and then selected the best grid based on the occurring mutation, as apposed to using only a single grid. This idea would theoretically yielded the best results, as I would have

been searching more of the search space per iteration, however once again I was unable to test that theory based on the limited amount of time I had available.

Initial ideas for how to solve the mini Sudoku puzzle came from studying the 8 Queens problem. The problem was successfully solved using simulated annealing in the past, and so I felt that researching into that would be a good place to start. The 8 queens problem shares many of the same constraints as Sudoku, and therefore meant that by changing the constraints and limiting the change of specific constraints, I would be able to create an algorithm which would solve Sudoku to a moderate degree of accuracy.