ECM2423

Evolutionary Algorithm Sudoku Report

660037119

Design

Solution Space and Representation

In order to represent solutions to this sudoku grid, I have gone with the approach of using a two dimensional list to model an individual sudoku grid. Each sub-list represents one row of the sudoku grid, as such there will be 9 sub lists in total. A visual example is provided below:

```
[
[4, 9, 5, 1, 8, 2, 6, 3, 7]
[5, 2, 7, 6, 4, 3, 8, 1, 9]
[7, 6, 9, 5, 3, 8, 4, 1, 2]
[6, 7, 5, 1, 2, 9, 3, 4, 8]
[2, 4, 7, 8, 3, 1, 9, 5, 6]
[8, 3, 9, 4, 1, 5, 6, 7, 2]
[3, 7, 2, 9, 8, 4, 5, 6, 1]
[6, 5, 4, 2, 3, 1, 7, 8, 9]
[9, 8, 6, 3, 5, 7, 1, 2, 4]
]
```

1.a

=

```
495|182|637
527|643|819
769|538|412
---|---|---
675|129|348
247|831|956
839|415|672
---|---|---
372|984|561
654|231|789
986|357|124
```

1.b

Storing the sudoku grid data in this form simplifies the process of reading a base sudoku grid from a file. It also makes checking for conflicts in rows and columns a quick process.

The solution space will therefore be a number of these representations with randomly generated numbers, of which will then be operated on to attempt to produce a correct solution.

Fitness Function

In order to evaluate an individual, I decided to sum the number of conflicts present across the whole grid. By conflicts I refer to a row, column or 3x3 sub-grid containing more than one number of the same value.

To do this I obtain all rows, columns and sub-grids from the individual and count the number of duplicate numbers in any of these to produce a final count of conflicts. I went with this approach since this fitness function is changed by the smallest of differences between grids, allowing differentiation of potential between even very similar grids. Upon the running of the evolutionary algorithm, this fitness function will be aimed to be minimised, 0 being the ideal value.

Crossover Operator

In terms of crossing two solutions together, taking into account my current method of solution representation and initial solution space. I came up with the decision to take a certain horizontal slices of two parents and recombine them to produce a new child grid. I choose a random slice location between grid rows to be used for each different crossover. To clarify, say the random slice was between row 2 and 3. The first parent would assume the first two rows of the child, the third to ninth rows of the child would be from the second parent.

Therefore by looking at figure **1.b** we can see there would be 8 different possible crossover scenarios, I use a random roll to determine which one. By keeping the integrity of rows during crossovers, we do not undo the work done during initial population creation of avoiding row conflicts.

Mutation Operator

In order to mutate an individual, I would go through each row and use mutation rate modifier compared with a random roll to determine whether this row should be mutated. If the row is to be mutated, I would swap the location of two non-fixed numbers within that row. Thus, similar to crossover, keeping the integrity of the row.

Therefore the maximum an individual could be mutated is 9 times, with 2 numbers swapped on every row, and the minimum being none at all. In the case a row only has 1 unfixed number, the row is never mutated, since this one unfixed number must already be in the correct location.

Population Initialisation

In terms of creating an initial population, after reading in the base sudoku grid and keeping record of the already fixed grid locations. I would then generate a complete random individual by going though all unfixed grid locations and filling them in with numbers (1-9) that are not already present in that row, ensuring row conflicts never occur upon an individual's creation. By doing this and leaving the fixed grid entries untouched, the effectiveness of the initial population is greatly increased in comparison to blind random generation of initial numbers.

Selection and Replacement

In order to transition from one generation to the next, I disregard a certain percentage of the current generation population. This percentage being the individuals with the least amount of potential determined by sorting of their performance with the fitness function. The exact percentage of individuals selected is defined by the user determined by the 'truncation rate'.

Alongside selection of the fittest for crossing, I also implement the method of replacement. This carries over some of the best performing parents in their exact form to the next generation, in order to attempt to avoid the disruption of a parent with very high potential when crossing over. The number of parents to be carried over is a percentage of the top performing individuals after evaluation and selection. This percentage, known as the 'replacement rate'.

Termination Criteria

In order to determine when the algorithm should stop transitioning to future generations, I do two main criterion checks. The first is whether the fitness function of the best individual in an offspring population is 0, meaning a fully correct solution has been found. In this case, stop the algorithm and print out the solution as a grid.

The second is in place to deal with scenarios where solutions are impractical to be reached, due to taking too long to solve or being continually stuck in local minima. Therefore I also have a limit in place to cap the maximum number of generations to pass through before terminating, and printing the best performing individual's grid (even though this would not be 100% correct). However I have also implemented a last resort guard against local minima, if the fitness function of the best individual is relatively constant over many generations (indicating local minima) I drastically increase the mutation rate temporarily to cause a 'super mutation', in an effort to escape this local minima. With this tactic, it's possible the algorithm may end up in a worse position that it had once had been. Therefore I keep track of the best solution ever discovered in any generation, and provide this upon unsuccessful termination.

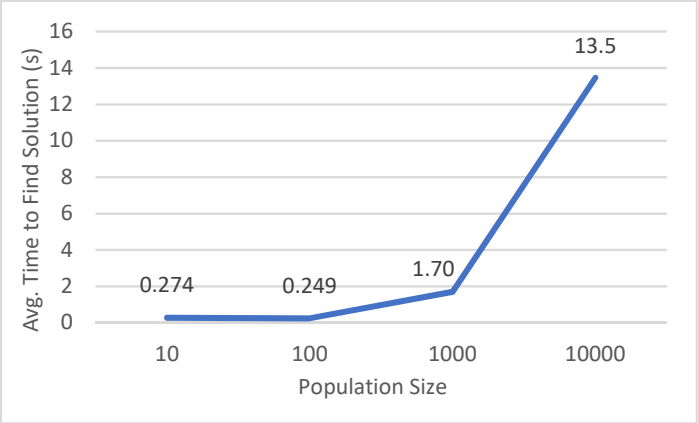
Experiment Results

Regarding the constant parameters for the experiments, detailed in the README file for reproducing results: I chose these parameters since they allow the algorithm to perform competently, while also adapting to local minima detection and attempting to relocate the global minima using 'super mutations'. Having 'max generations' as 1000000 ensures is not a limiting factor, to make the experiments fair.

I believe the most effective algorithms are able to retrieve a correct solution in the shortest amount of time, especially in an industrial setting. As such, I have evaluated time in my experiments as the comparative dependent variable. As another option, generation number on the other hand, does not correlate directly with effectiveness of an algorithm, this is because it would be bias towards the larger populations since they do more computation per generation. Smaller populations can traverse generations faster, so a hard cap limit would not evaluate performance evenly.

Grid 1

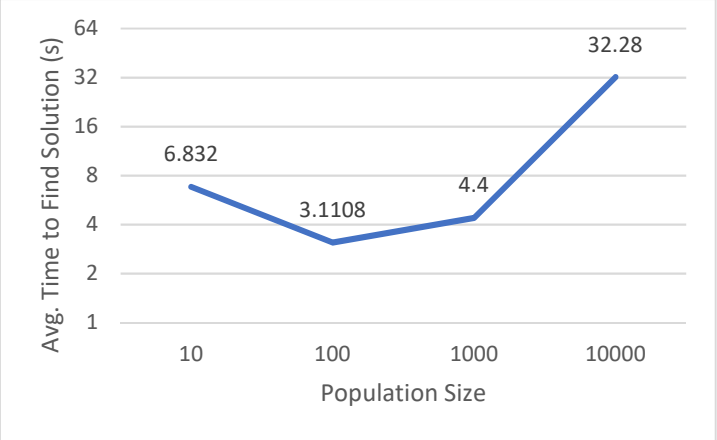
Since this grid is so simple to solve for an evolutionary algorithm, in order to measure performance I will compare the average time needed until the solution grid was reached. I also record the number of generations needed traversed to reach the final solution in the data table as extra information in the table:



Pop. Size and data type	Individual Run Data				
	1	**2**	**3**	**4**	**5**
10 Time (s)	0.493	0.522	0.138	0.124	0.0948
10 Gen	434	450	119	104	81
100 Time (s)	0.187	0.240	0.344	0.164	0.308
100 Gen	14	19	28	12	25
1000 Time (s)	1.647	1.663	1.872	1.892	1.434
1000 Gen	12	12	14	14	10
10000 Time (s)	13.5	13.4	14.7	13.4	12.3
10000 Gen	9	9	10	9	8

Grid 2

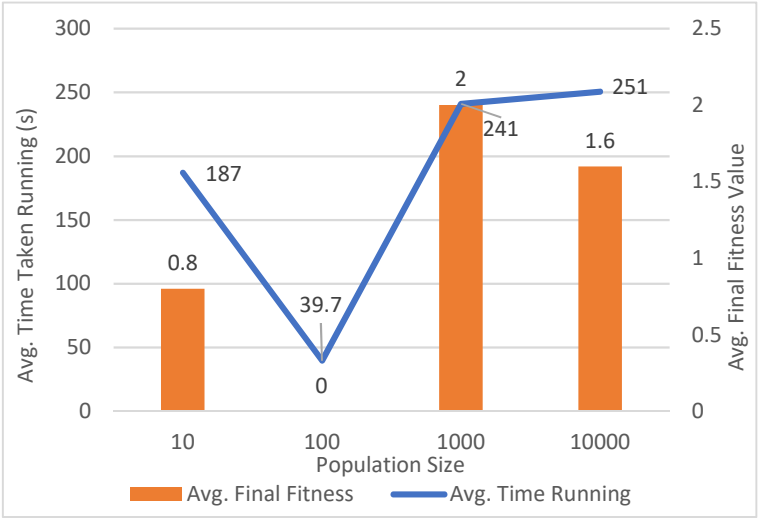
This grid is a little trickier, however I have never had an unsuccessful run under 100000 generations, and so will use the same approach as grid 1 to measure performance.



Pop. Size and data type	Individual Run Data				
	1	**2**	**3**	**4**	**5**
10 Time (s)	1.13	8.92	3.36	4.25	16.5
10 Gen	998	7854	2979	3600	14720
100 Time (s)	2.34	0.744	8.82	2.63	1.02
100 Gen	210	63	784	230	89
1000 Time (s)	4.84	3.57	4.83	4.92	3.84
1000 Gen	40	28	39	41	31
10000 Time (s)	35.5	34.4	30.0	27.9	33.6
10000 Gen	27	26	23	21	26

Grid 3

This grid is difficult to solve. Some experiments take a very long time, especially with larger populations. Therefore to practically compare different population size performance on this grid, I will run the algorithm for 5 minutes maximum, and record the best fitness achieved and at which generation this was (might not be the final generation if the solution was not reached). If a correct solution was found within this limit, I will also record the time, otherwise it will be 300s (MAX).



Pop. Size and data type	Individual Run Data				
	1	**2**	**3**	**4**	**5**
10 Time (s)	97.1	82.3	MAX	MAX	156
10 Gen	83805	72015	91865	253224	131320
10 Fitness	0	0	2	2	0
100 Time (s)	9.23	3.22	62.3	56.4	67.4
100 Gen	790	277	5402	4720	5964
100 Fitness	0	0	0	0	0
1000 Time (s)	MAX	MAX	MAX	5.25	MAX
1000 Gen	445	47	43	43	78
1000 Fitness	2	2	4	0	2
10000 Time (s)	MAX	53.24	MAX	MAX	MAX
10000 Gen	50	42	48	39	45
10000 Fitness	2	0	2	2	2

Questions

Which population size was the best?

By taking into account all 3 grids, it is clear that 100 was the most effective population size. It performed the fastest in all cases to reach the solution grid. As well as this, it never failed to reach the full correct solution in under 5 mins when I was testing it on grid 3.

Why?

I believe this is because any lower than 100, the generations are too volatile, the rely too much on luck of the stochastic behaviour generating a random individual with good potential. While this works quite well for easy problems (such as grid 1), it will struggle more in direct proportion to the problem complexity. Because this small population size is so volatile, it is also difficult to detect when it has entered a local minima in order to apply a 'super mutation'.

Populations much larger than 100 on the other hand, most likely will almost always have good potential individuals, but take too long to traverse generations and thus makes them inefficient in finding the solution in time. The approximate population of 100 is therefore closest to the optimum population which takes the middle ground between these two extremes.

Which grid was the easiest and hardest to solve?

The grids increased in difficulty from 1 to 3. I can tell this because the time spent running was larger for every population size when going from grid 1 to 2 or grid 2 to 3. Grid 3 was so difficult, some sizes could not resolve a result in under 5 mins. Therefore I can conclude grid 1 to be the easiest and 3 the hardest.

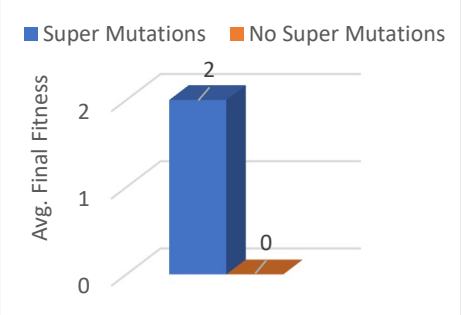
Why?

This difficulty curve directly corresponds to the amount of numbers pre-set in each grid, the more pre-set numbers, the easier the grid. This is because with fewer pre-set numbers, there are more possibilities when generating, mutating or crossing individuals. This means that there is a smaller chance of one of these operations producing an individual with a large amount of potential (close to the actual solution).

Further Experiments and why?

If I were to spend more time researching this algorithm there are a quite a few things I believe would also be useful to experiment. These include: the impact of changing the truncation rate, I did some small tests to pick a good number for my tests, but proper experiments could uncover the optimal truncation rate (harshness of selection) for this algorithm. Another possibility would be the same approach but with replacement rate, Is there an optimum percentage of top performing parents to replace in the new offspring population? Finally, do these 'super mutations' that I introduced as a guard against local minima actually have an effect, and what values for the tolerance on the detection of local minima are the best? All these further experiments would be useful as they may lead to uncovering optimum parameters that make the algorithm much more efficient.

As I was interested to see If my 'super mutations' really did make a difference, I performed a quick and small extra experiment using grid 3, with population size 100 recording average final fitness with and without super mutations over 5 runs running for a maximum of 5 minutes:



Run Type	Individual Run Final Fitness				
	1	**2**	**3**	**4**	**5**
No Super Mutations	2	2	0	3	3
Super Mutations	0	0	0	0	0

Here we can clearly see the benefit of super mutations, succeeding to find the solution every time (as before). In comparison to not using super mutations, where it only succeeded once, with an average final fitness of 2. Showing the advantage of being able to escape local minima.