ecm2423 - Sudoku as an Evolutionary Algorithm problem

March 2022

1 Design

1.1 Problem representation

The problem is abstracted to a large extent to facilitate efficient computation. A specific solution, Genome in terms of the program, is represented as ordered set (array) of Genes. Genes are atomic objects carrying an integer value.

Organic Genome is a sequence of Genes with capacity to change and/or mutate any of its Genes.

Before being evaluated in a fitness function, each Organic Genome is encapsulated into a Corrected Genome. An object set-up at the beginning of the run with a set of rules, representing the problem itself by redirecting calls to a specific locations to its own record of Genes. That way whenever Organic Genome is evaluated, it does not alter the values set-out in the problem.

Genome is only fitted into a Sudoku 9*9 grid during evaluation to ease writing of the fitness function.

1.2 Design choices

(a) Solution space is G^{9^2} where G is the Gene domain composed by distinct Genes. Each solution is represented by a Genome, a sequence of 81 specific Genes.

(b) Fitness Function $f(g)$ is such function that takes a Genome g and returns a sum of number of Genes spotted after having spotted it at least once within each column, row and the nine 3*3 squares.

(c) Offspring creation mechanism: the Child Genome iterates through each of its Genes and, at random, selects Gene from one of its Parents. This mechanism ensures that $P(G_n|G_{n-1}) = P(G_n)$, meaning that there is no relation between consecutive genes.

(d) Each generation has a constant ratio of mutants created as clones from previous generation. After the genome is selected, its Genes have a constant probability of mutating.

(e) Population is initialised completely randomly. Each Genome is built by randomly selecting 81 Genes from the entire domain with a constant probability.

(f) After a experimentation with different methods of selection, including:

- Selecting from the entire population.
- Ordering the parents according to their fitness, and selecting them linearly descending weight.
- Ordering the parents and selecting adjacent pairs.
- Ordering the parents and selecting from the ones in the first half with constant weight.= at random.
- Ordering the parents and selecting at random with constant weight from all Genomes performing above average.

The last method was chosen as it had the best results. Further practical examination revealed that this approach was prone to getting stuck in local minimums and benefited from significant mutant population and fairly high mutation rate. (For more information see next section on constants.)

(g) Termination criterion is two-pronged. Either a Genome with fitness value of 0 is discovered, or 24 hours will elapse after which the program is manually terminated.

1.3 Constants and default values

- *geneMutationRate* is a constant in the Organic Genome class. This constant defines the probability of every single Gene undergoing mutation and changing into a random different Gene at a probability of $P() = \frac{1}{geneMutationRate}$. The value found working well enough was 10.
- *genomeMutationRate* is a constant in the Organic Genome class. This constant defines the probability that the Genome will mutate at all when undergoing mutation at probability of $P() = \frac{1}{genomeMutationRate}$. It was implemented to allow for different mutation paradigms. The current value is 1 to allow for a constant size of the mutant population.
- *Genes* are a predefined constant. $G \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ They are specific to the problem. If the code was re-purposed, different values would be assigned to them.
- *muRate* Mutant Rate, a number dictating the proportion of the mutant population to the population of sexually produced offspring. The default value found to work well with the other setting is 0.2.

2 Code

Note that the code is not included in this report. The code files are attached in the .zip file in their own folder as the codebase consists of over 700 lines across multiple files.

3 Performance Analysis

Population Size	Avg Time G1	Avg Time G2	Avg Time G3
10	11H 07M 21S	21H 48M 20S	∞
100	08H 28M 48S	09H 03M 55S	∞
1000	∞	∞	∞
10000	00H 21M 13S[1]	∞	∞
Average[2]	09H 48M 05S[3]	10H 17M 14S	∞

Table 1: Average Performance per Grid

Population Size	Average Gens G1	Average Gens G2	Average Gens G3
10	6,775,642	4,306,351	∞
100	2,278,043	2,836,024	∞
1000	∞	∞	∞
10000	1031[1]	∞	∞

Table 2: Average Population per Grid

3.1 Population Size

3.1.1 Possible Explanations

Small population sizes (10) are more susceptible to getting stuck in local minimum, as with smaller mutant population, the chance of a mutated Genome stumbling onto a better value is smaller. We postulate that the reason smaller population fared better, is because their small size allowed to have much smaller time/generation allowing them to refine desirable traits faster while getting similar volume of mutants.

The reason the very large population was not able to fully benefit from this is because the child generation is not multi-threaded and because of the larger memory management overhead, the actual time/mutant was smaller than for slightly smaller populations.

[1] Sample size = 1
[2] Does not include runs that did not finish
[3] Does not include one anomalous result in 1

3.2 Grid difficulty

Grid 1 and Grid 2 had similar times overall and similar generation count. The Grid 3 proved the be the most problematic of the three Grids as the EA could not produce a single result even after a full day of execution.

3.2.1 Possible Explanation

Given that all of the solutions (for a given grid) are identical. There is no evidence suggesting that some grids would be solved faster because there would be multiple possible solutions to a given configuration.

One possible explanation would be that the random distribution is not completely random, making it more likely to arrive to configurations closer to the grids' solution.

3.3 Further experiments

- More runs could be performed to obtain more accurate results.
- The memory usage could be optimised in order to allow comparing the other heuristics with Manhattan distance for all tiles.
- The child generation could be optimised for multi-threaded environment to allow larger population sizes to run faster.
- The results have shown to be highly dependant on mutant population size and gene mutation rate. It would be highly insightful to explore mapping of these two values on execution time.