INFR11161 Natural Computing - Main Assignment

B179419

November 16, 2023

1 Problem 1

We Investigate the effect of population size in the Particle Swarm Optimization Algorithm using the Rastrigin function as the objective function.

$$f(\mathbf{x}) = An + \sum_{i=1}^{n} \left[x_i^2 - A\cos(2\pi x_i)\right]$$

where n is the dimension of the function, $\mathbf{x}$ is a point in the search space, and A is a constant typically set to 10.

The algorithm allows us to minimise the objective function using a swarm of particles, which we initialise at random positions and having random velocity. After each iteration, the particle's position and velocity are updated based on the best position of the particle found so far and the global best particle position found so far. A position is better or worse according to the objective function. We use the canonical version of the PSO algorithm(as discussed in lectures) to minimise two dimensional rastrigin function.

1.1 Part a)

To determine the optimal population size for PSO algorithm, we fix all algorithm parameters except the population size. We use the 3D Rastrigin function with the search bounds set from -100 to 100, and the algorithm was run for 1000 iterations. Inertia weight ω was taken as 0.7, with cognitive (α_1) and social (α_2) constants both equal to 2.02. The variable in this experiment was the population size, which was observed from 1 to 200 particles.

It was found (figure 1) that smaller population sizes resulted in poor performance, with the PSO algorithm failing to converge to the optimal solution. The algorithm converged to the optimal solution after a population size of about 15 particles. Population sizes smaller than 15 had cases of the optimal solution but were unreliable. From this experiment, we infer that a population size of approximately 15 particles is optimal.

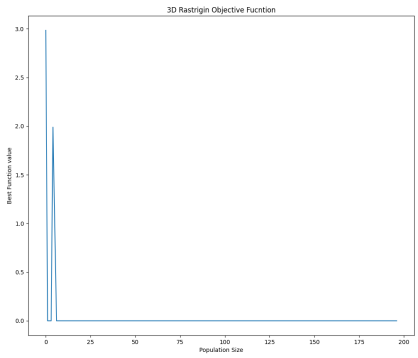

Figure 1: Best values after minimisation of 3D rastrigin function

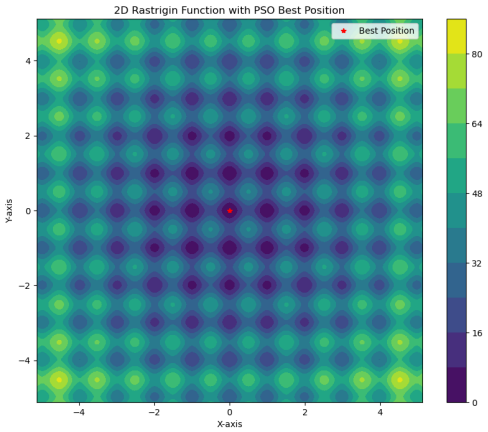

Figure 2: Optimal value obtained for Rastrigin Cost function

1.2 Part b)

We found the optimal number of particles for increasing problem complexity (increasing the dimensions), and it was observed that the number of particles increased linearly. Figure 3 shows the effect of increasing population size on the optimal value of the objective function. We also see in figure **??** that the minimum population size required for dimensions 6 to 12 also increases linearly.

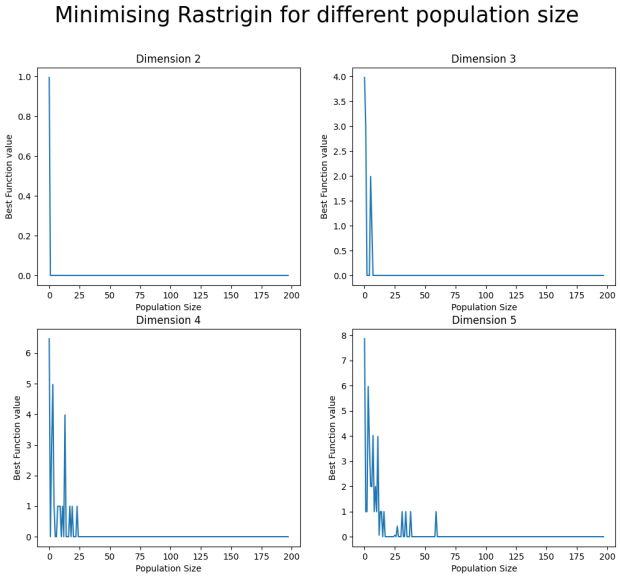

Figure 3: Best Rastrigin Function values optimised from PSO algorithm, with fixed 1000 evaluations of cost function, with variable **population size** and variable **problem difficulty**

2 Problem 2

Implementation of the solver is done in problem2.ipynb, consisting of some helper functions and the GA class, takes an instance of the game and has functions to solve the instance. We generate our own instances of the Sumplete problem through the 'generate_sumplete_instance' function. The generation process takes a deletion rate (as suggested we use deletion rate = 1/3) and a range of positive numbers.

We implement the Canonical Genetic Algorithm, as discussed in the lectures. For a $k - by - k$ variant of the Sumplete problem, the alphabet A is binary, i.e. A = {0,1}. An individual is encoded as a k-by-by matrix consisting of $s_{ij} \in \{0, 1\}$ for $i = 1, ..., k$ and $j = 1, ..., k$, the encoding of '0' deletes the number in the game grid, and encoding of '1' keeps the number.

We use 'generate_initial_population' function in the 'GA' class to generate a population of individuals with random values. This population is put through the Canonical version of GA. Tournament selection was used as it allows good balance between exploration and exploitation. It maintains diversity and does not eliminate population, which was a problem with practical Roulette wheel selection (implemented earlier). The selection process involves running several tournaments from randomly chosen participants and choosing the one with the highest fitness.

After selecting suitable parents, we produce their offspring through 'crossover' function in the GA class. This function deploys a straightforward and effective single-point crossover on the parents with 'p_c' crossover probability set to 0.7 (typical value for binary representation). It chooses a crossover point on the grid and swaps the values after that point between the parents, so the children have the first part from one parent and rest from the other. We run a mutation process on each child obtained from the crossover; we randomly flip the bits (from 0 to 1 or 1 to 0) with mutation probability of 0.01 for our binary representation of individuals.

2.1 Part a)

Figure 4 shows that using a fitness function which assigns a fitness of 1 to correct and 0 otherwise, doesn't scale with $k \geq 3$. The experiment solved ten randomly generated sumplete problems using the GA algorithm with the mentioned fitness function. We found that for k=5,6 sumplete problems, the algorithm failed to provide an optimal solution in 1000 generations, and 7 out of 10 problems were solved for k=4.

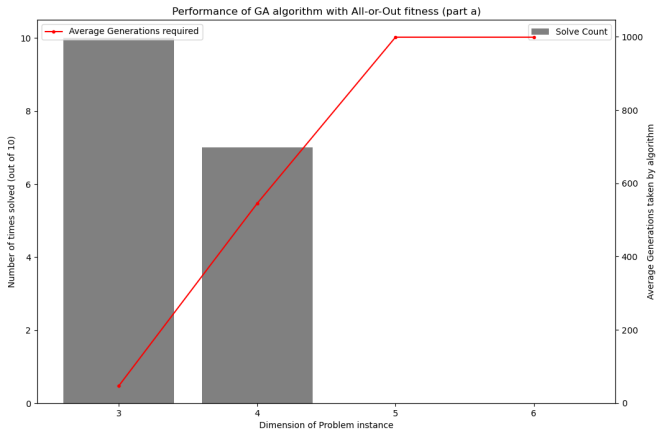

Figure 4: Performance of fitness function giving 1 for correct solution and 0 otherwise.

2.2 Part b)

We used a smooth fitness function to make the algorithm scale more favourably with k, which gives a score between $[0, 1]$. The function gives the inverse of the sum of absolute differences between

the target and evolved sums. The higher the difference, the lower the fitness score, and vice-versa. The same experiment as 2.1 was conducted, but with this smooth fitness function and maximum allowed generations now increased to 2000. Figure 5 shows the performance; the improved GA algorithm now solved 4,3 sumplete problems of k=5 and k=6, respectively.

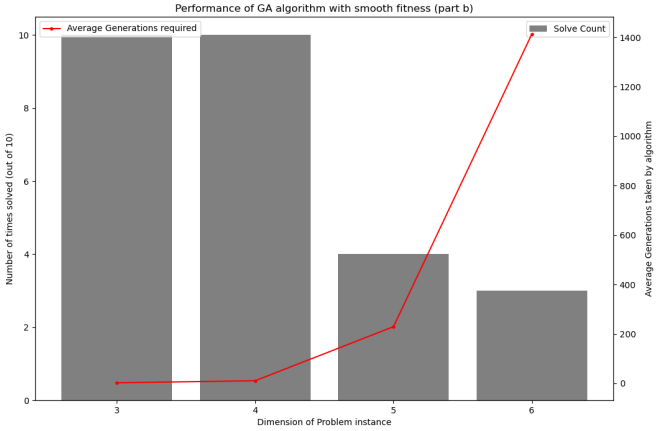

Figure 5: Performance of Smooth Fitness Function

2.3 Part c)

In solving a problem using GA algorithm, parameters play a crucial role. First, the population size matters since more individuals can explore more of the search space but would require higher computing power. This tradeoff was already seen in the above two parts. Second there is crossover probability, which gives how often we breed the selected parents; too little breeding can limit the exploration, but too much breeding can prevent the best answers. Third, Mutation probability with which we introduce random changes; this helps avoid getting stuck in local minima but too much randomness can be counterproductive. We ran an experiment on performance of different Mutatation and Crossover probability (see Figure 6 7).

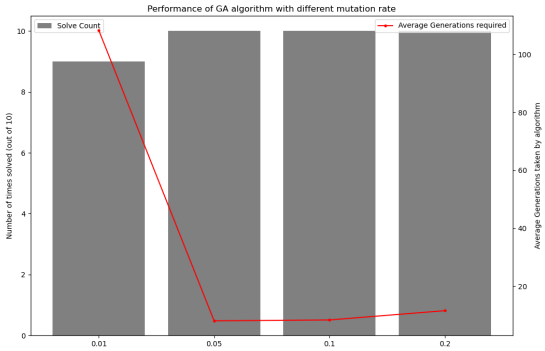

Figure 6: Mutation Rate

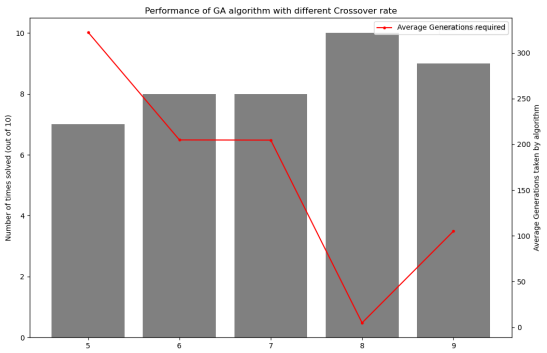

Figure 7: Crossover Rate

3 Problem 3

3.1 Part 3a)

We have chosen the Fibonacci sequence (A000045) from the OEIS [1]. The Fibonacci series is a classic example in mathematics, where each number is the sum of previous two numbers starting from 0,1 as the first two elements of the sequence. It was a good choice for GP since it has a well defined recursive function. we use the first 10 numbers of this sequence as our target dataset. Trying to find a rule that closely follows this sequence can help find the next numbers in this sequence.

Since we only know the first 30 numbers in the sequence, we take these values to calculate the fitness of a tree. A Genetic Programming tree consists of operators (like addition and subtraction) as pre-terminals and constants as terminals. We include the number n in the terminal set.

The algorithm initialises by generating a population of random trees, where the trees can have depths between 4 and 6 (inclusive). The population size is crucial for exploration; we use a 60 tree population size. The algorithm selects individuals from the roulette wheel selection after two parents are selected they are bred together. The 'crossover' function of GPTree class randomly attaches a subtree from one to another with crossover probability of 0.8, we mutate it with a mutation probability of 0.1. This is run for almost 300 generations, and the best-performing tree is chosen.

3.2 Part 3b)

The fitness function we use calculates inverse mean of absolute differences, so the maximum fitness we can get is 1 and minimum is 0. The GPTree class is able to construct a rule for Fibonacci sequences with a fitness value of 0.304. The Evolution of the best fitness score is shown in figure 8.

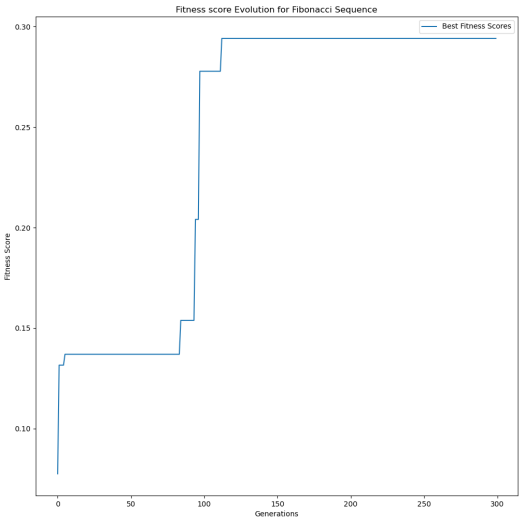

Figure 8: Performance of GP on Fibonacci Numbers - A000045

3.3 Part 3c)

After testing the GP algorithm on 3 distinct sequences from the OEIS- A005875 (Theta series of simple cubic lattice. fig 9), A250000 (Peaceable coexisting armies of queens: fig 11), and A004215 (sum of 4 but no fewer nonzero squares. fig 10) - it's evident that the GP approach exhibits versatility and adaptability. However, its feasibility depends on the complexity of the sequence. It

performed well on evolving to more minor sequences, but further fine-tuning needs to be done for larger sequence datasets (we only used 10 to 30 numbers from each sequence).

It Performed well on fibonacci

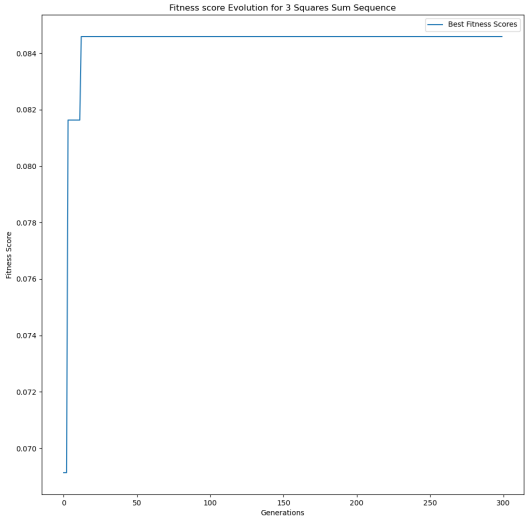

Figure 9: Performance of GP on 3 Square Sum - A005875

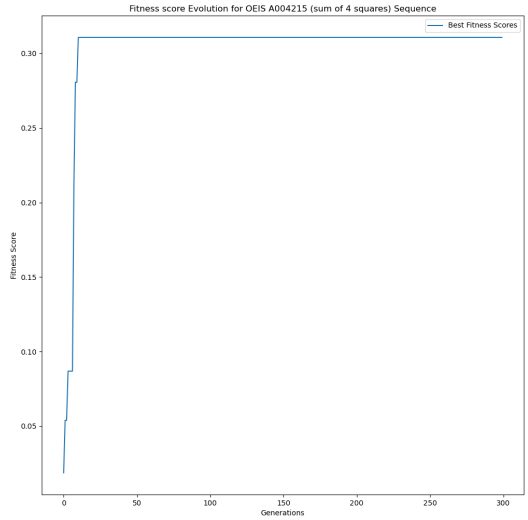

Figure 10: Performance of GP on 4 Square Sum - A004215

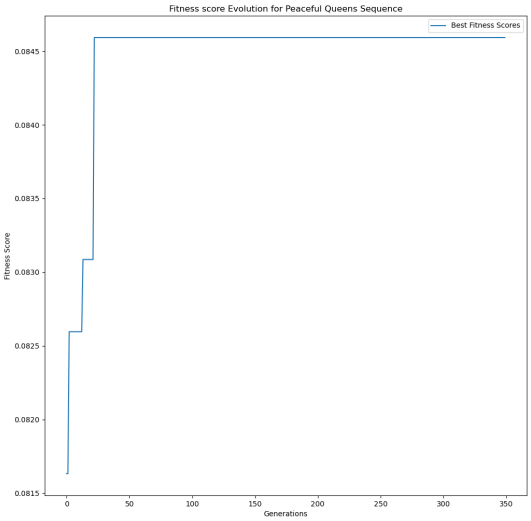

Figure 11: Performance of GP on Peaceful Queens

References

[1] On-line encyclopedia of integer sequences (oeis) wiki.