ROHIT GARG
2018A7PS0193G

Artificial Intelligence CS F407 Assignment-1

Introduction

- Inspired by evolutionary biology, Genetic Algorithm uses selection, crossover, and mutation operators to efficiently traverse the solution search space.
- In this assignment we come up with a variant of the GA algorithm so that the best fitness value in a population improves in a faster manner over successive generations.
- We intend to improve the performance of our Genetic Algorithm implementation for 8-Queens Problem and Travelling Salesman Problem in terms of better convergence.
- Areas for improving genetic algorithm:
  - Choice of Fitness Function
  - Selection of Initial Population
  - Selection of Parents for Cross-Over
  - Cross-over Method
  - Mutation Method
  - Mutation Probability
  - Selection of population for next generation and many more….

Q1. 8-Queens Problem

- The goal of the 8-queens problem is to place eight queens on a chessboard such that no queen attacks any other. A queen attacks any piece in the same row, column or diagonal.
- The successors of the state are all possible states generated by moving a single queen to another square in the same column. So each state has 8× 7 = 56 successors.
- **<u>Fitness Function</u>:**
  => Fitness function = 1 + number of pairs of queens not in attacking position.
  => Fitness function = 1+ ${}^{N}C_{2}$ - Number of attacking pairs of queens

- **State Representation:**
  The state of the chess board is represented by an object of Board class. An instance of Board class has two instance attributes ‘state’ and ‘fitness’ and one class variable ‘max_non_clashes’.

  state is a 8-character string where every value at i^{th} index represents the row number(1-8) placed on i^{th} column.

  fitness represents the current fitness of the state.

  max_non_clashes represents the maximum number of non-attacking pairs of queens. It is $^{N}C_2$ for N-Queens Problem(Choose 2 queens out of N). Since N=8, it comes out to be 28 for our case.

  An instance of class Board also contains a method getFitness() which returns the fitness of state

- **Population:**
  At any generation population is represented by a list of class Board objects. Population size is 20.

- **Selection of Initial Population:**
  Initially, All the states in the population are the same and correspond to the eight queens being on the same row. Every state of the initial population has a fitness value of 1.

- **Selection of Parents for Cross-Over**

  Here Roulette/Wheel based approach is used. In the roulette wheel selection, the probability of choosing an individual for breeding of the next generation is proportional to its fitness, the better the fitness is, the higher chance for that individual to be chosen.
  and it tends to give more optimum results than Tournament and completely greedy-based selection.

  Approach -2
  I also tried a greedy based selection strategy where states are ranked on fitness and best two fit states are chosen for breeding in cross-over method and this

strategy leads the algorithm into unavoidable local optimum since there is no notion of randomness involved.

- <u>**Cross-over Method:**</u>

- For each pair of states to be crossed over, a random crossover point is chosen and the parents are crossed-over from the point to create a new child.
- I have modified crossover() function such that if the child produced has greater fitness than the two parents then the child is passed else more fit parent is passed to the new population.[max(child, parent1, parent2)].
- With this greedy approach, I am able to reach a solution in a lesser number of generations. Although, we may argue that there is a chance of convergence into local maxima but this problem is solved by the mutation function.
  <u>Approach-2 :</u>
- I also tried ordered-crossover but there was no significant improvement. Also schema improves more profoundly when we use random-point for cross-over rather than doing ordered-crossover.

- Instead of producing one child, in order to introduce diversity, I also tried creating 4 children for each pair of parents however, this was computationally expensive and often it got stuck at a local optimum value and did not produce significant change.

- <u>**Mutation Method:**</u>
  Mutation adds a significant additional value to GA by introducing random change which could assist in overcoming local minima in the search exploration. Moreover, in the case of 8 queens problem mutation plays a key role in better convergence, hence finding the global optimum value .

  As an improvement of the basic algorithm, we here perform a local hill climb search with random restart option.

1. We compare fitness values of all the 56 neighbours with the current child state.
2. The best neighbour is chosen among all the neighbours.
3. If it is better than the child then return the best neighbour
4. Else the child to be mutated is already the best among all its neighbours then we randomly move to any neighbour i.e. we randomly select a column and randomly place the queen in any row.

This approach is the key to faster convergence. By performing a local greedy hill climb search, we ensure that we move to the best neighbour and this eventually populates the list with best fit states over time. Apart from this, the latter random movement to the neighbouring state guarantees the algorithm will not get stuck in the local minimum.

Other approaches like swap mutation and double random mutation were also tested but none could converge as good as the above algorithm.

- Since, initially, all the queens were in the same row, swap mutation did not reduce the number of clashes among queens since it merely would swap queens of two randomly selected columns.
- Double random mutation would lead to very high fluctuations in the graph and did not produce a stable algorithm.

- **Mutation Probability:**
  The basic algorithm assumes a random initial population and therefore suggested mutation rates between 3% and 10% hence we need a higher mutation rate for our use case.
  We start with a mutation probability of 5% and increment it by one every time current best fitness value of the population is equal to the previous value else we reset it to 5%. With this, even if the best fitness value of the population does not change we increase the probability of mutation of the state. This will ensure we move to neighboring states and not get stuck on hill climb.

```
if(previousBestFitness == currentBestFitness):
            mutation_probability += 1
        else:
            mutation_probability = 5
```

- **Selection of population for next generation**

Instead of passing on the new population as it is in the next generation as done in the basic algorithm, we create a mixture of population inspired by elitism and random selection.
We pick the top 80% of the population from the new population list ranked by their fitness values and the remaining percentage randomly equally likely from the old

population. In this way, a balance between greedy approach and randomness is ensured in successive generations.

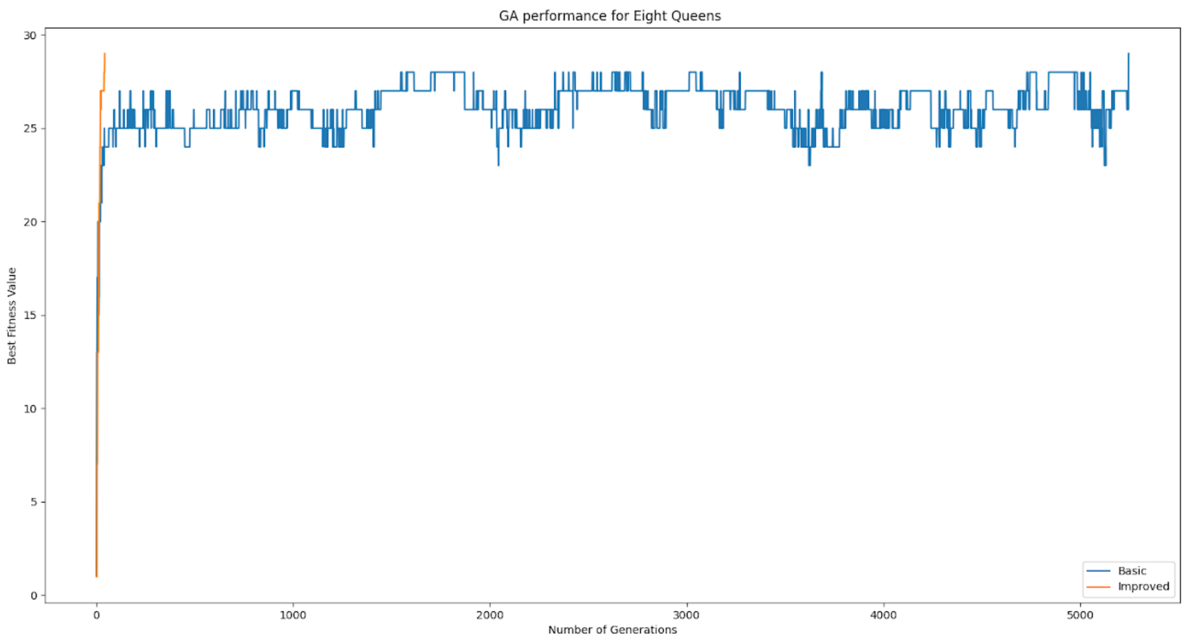
I also tried picking the best(k) states among the new(k) and old population(k) list. However, this greedy approach again got stuck in local optimum.

- **Stopping Criterion**

The algorithm terminates when we find a population with desired best fitness value of 29.

- **Result**

As a part of the question we compare the two algorithms (best and textbook) graphically for a typical run. After running the two algorithms repeatedly for various seed values, it was concluded that the target fitness value (= 29) was reached after 5000+ generations for the book implementation of the Genetic Algorithm , while the Improved/Optimised version reached the same in under 100 generations on average.



Q2. Travelling Salesman Problem

- In a symmetric TSP problem, a salesman has to visit a number of cities and return back to the original (first) city with the shortest route.
- **Population Size** = 20 [same for both basic and improved versions]
- Distance between the cities is maintained in the form of 14X14 Pandas distance dataframe.Non-neighbouring cities had infinite distance between them and was denoted using sys.maxsize
- **Fitness Function Used**:
  => Fitness function = 1/Total path length
  Note: This was kept same for both basic and improved version.

- **State**:
  The tour(complete round trip) is represented by an object of Tour class. An instance of Tour class has two instance attributes 'state' and 'fitness'.

- **Initial Population** : List of states each of form "ABCDEFGHIJKLMN"

- **Selection of Parents for Cross-Over:**
  Both tournament based and wheel based selection was applied. However tournament based approach dominated over wheel based selection.In tournament based selection, some n states were randomly selected out of k total states and the best 2 states among these contestant states were picked as the parent. TOURNAMENT_SIZE = 2 gave the best convergence.

- **Cross-over Method:**
  Ordered crossover was performed instead of basic random crossover method since the latter would produce an invalid state.

- **Mutation Method**:
  As an improvement of the basic algorithm, we here perform a local hill climb search with random restart option.

1. A neighbour is represented by swapping two cities in the tour.
2. We compare fitness values of all the neighbours with the current child state.
3. The best neighbour is chosen among all the neighbours.
4. If it is better than the child then return the best neighbour
5. Else the child to be mutated is already the best among all its neighbours then we randomly move to any neighbour i.e. we randomly select two cities and swap them.

This approach is the key to faster convergence. By performing a local greedy hill climb search, we ensure that we move to the best neighbour and this eventually populates the list with best fit states over time. Apart from this, the latter random movement to the neighbouring state guarantees the algorithm will not get stuck in the local minimum.

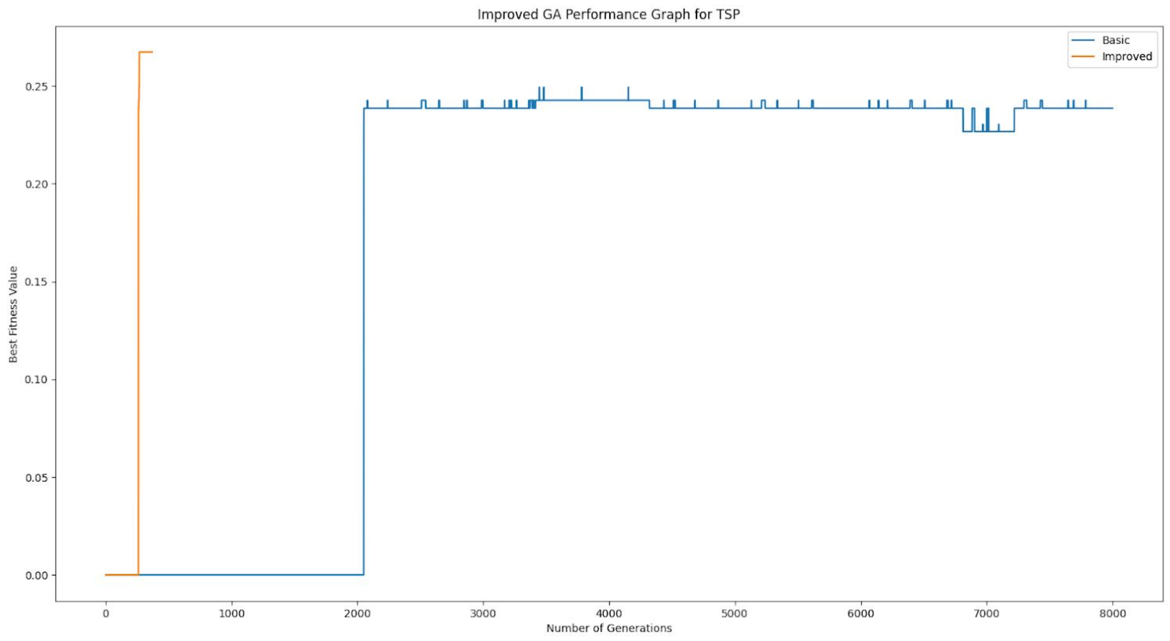
- **Mutation Probability:**
  Mutation was adjusted according to the best fitness value of the population. Initially mutation probability is 5. If the best fitness value did not cross the limit 10^(-6), it is clear that there is no finite length correct route or tour discovered yet so mutation probability is increased by 1 weight.
  Also, if best fitness is not changing, it means that we have hit a local minima and thus we need to randomly move to a neighbour and thus mutate. Hence in this case also, mutation probability is increased by 1 weight. Else, it is reset to 5.

- **Selection of new population for next generation:**
  Instead of passing on the new population as it is in the next generation as done in the basic algorithm, we create a mixture of population inspired by elitism and random selection. We pick the top x% of the population from the new population list ranked by their fitness values and the remaining percentage(100-x%) randomly equally likely from the old population. In this way, a balance between greedy approach and randomness is ensured in successive generations.

- **Stopping criterion** : The algorithm terminates if:
  - maximum number of generations is reached which is kept to be 8000 generations
  - or finite path is discovered and the best fitness value does not change for 100 consecutive generations

- **Result:**

  The best solution had the fitness value of 0.2673 . As evident from the graph the improved algorithm gave the solution 80-90% of times under 200 generations on average. Whereas, the basic version did not even converge sometimes and thus was made run compulsory for 8000 generations.


Improved GA Performance Graph for TSP
Basic
Improved
Best Fitness Value
Number of Generations