Artificial Intelligence (CS F407)
Assignment I

GENETIC ALGORITHM

Devanshu Shah
2018A7PS0240G
February 14, 2021

1 Introduction

Genetic Algorithm relies heavily on inspiration from nature, they have been around since the 1960s and have been an area of research since then. The work by Geoffrey E. Hinton and Steven J. Nowlan in 1987 gives great insight in the implications of Baldwin Effect in Genetic Algorithms, a central concept to evolution! We will also try to gain inspiration from biological phenomena to improve upon our algorithms.

2 Genetic Algorithm on NQueens

2.1 Introduction

The NQueens (8Queens in our case) is a problem in which we have an NxN chessboard where we have to place N Queens in a way so that no pair of queens are in attacking positions (attacking is defined based on the conventional rules of chess). This is a hard problem if it is taken to be a normal search algorithm as there are $N!$ states to explore (this is *after* we prune the search space by restricting a single queen per column). So conventional search algorithms like BFS, DFS won't be much helpful as they work in $O(num\_states)$.

2.2 State Representation

We could have represented the state as an NxN matrix but since we are familiar with the rules of chess, we can say with certainty that it'll never be the case that two queens are in the same row, hence we make a one dimensional array of length N as the state representation where the entry at index i contains the row number of the queen in the column i.

2.3 Fitness Function

The fitness function is used by the algorithm to decide which individuals in the population are ideal for reproduction and should carry the generation forward. The fitness function for NQueens is defined as follows

$$\text{Fitness}(state) = 1 + num(\text{ConflictingQueens}(state))$$

2.4 Improvements in the Algorithm

To improve the algorithm we first need to understand the original algorithm. The Original Algorithm already performs really well on this problem, mainly because of the relatively small NQueens(8Queens) and the inter-relations of the states. However there are a few areas of improvement. Let us consider the main training loop of the algorithm

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
      new_population ← empty set
      for i = 1 to SIZE(population) do
          x ← RANDOM-SELECTION(population, FITNESS-FN)
          y ← RANDOM-SELECTION(population, FITNESS-FN)
          child ← REPRODUCE(x, y)
          if (small random probability) then child ← MUTATE(child)
          add child to new_population
      population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Figure 2.1: Genetic Algorithm (Naive)

The potential functions that can have improvements are the following:

1. *RANDOM-SELECTION(population, FITNESS-FN)*
2. *REPRODUCE(x, y)*

3. *MUTATE(child)*

We will look at betterment in each of them

2.4.1 Improving MUTATE

The way we mutate does not have much improvement to be done, primarily because mutation should *not* be the driving force of the future population and hence the actual mutation of an individual should be minimal (trying to mutate an individual too much led to detrimental results) Now coming to the frequency at which mutation occurs in the population, inspiring ourselves from how evolution works, when the generations are just starting, mutating too much is going to lead to poor generations however once the generations have evolved a fair bit, mutations can give great results that aren't even expected. Hence after experimenting, it was found that when the generations are still young we let it mutate with a small probability of 10% once it is old enough we force mutation on the offsprings hoping that if the mutation is bad it won't propagate to the next generations however if it is good, it will lead to the betterment of the population! (Inspiration from Biological Mutations)

2.4.2 Improving RANDOM-SELECTION

This is critical to decide how the new offsprings will be produced, as after all most of the traits of the offspring will be determined by their parents. One option is to of-course pick the best parents available and make them reproduce, surprisingly(or unsurprisingly) it fails very badly, interestingly neither is this approach followed by actual evolution! To speak in mathematical terms we most of the times end up getting stuck in a local maxima, on a philosophical note, you never know what great things can come when two not so great things combine! So we will *not* do a greedy selection of parents, but there is one small improvement that lead to some improvement in the stability of the algorithm, instead of forming probability by doing $(fitness)/sum$(fitnesses) we will do a softmax based probability calculation, which is

$$p_i = \frac{e^{fitness(i)}}{\sum e^{fitness(i)}}$$

2.4.3 Improving REPRODUCE

Arguably the most important part of the algorithm, it is responsible for deciding which traits to pick from the parents to produce the best possible offspring. There are several options for this. Some of the famous ones are PMX, OMX, UPMX, CX, CX2, and a lot more. We will look at the ones mentioned

PMX: Partially Mapped Crossover was introduced by Goldberg and Lingel, in "Alleles, Loci and the Traveling Salesman Problem", The crossover is made by randomly choosing

two crossover points x_1 and x_2 which break the two parents in three sections S1 and S3 the sequences of Parent1 are copied to the Child1, the sequence S2 of the Child1 is formed by the genes of Parent2, beginning with the start of its part S2 and leaping the genes that are already established.

UPMX: It is the same as PMX however it does the crossover with a predefined probability, and does not need start and end indices

OMX: The Ordered Crossover method was also presented by Goldberg, it is used when the problem is of order based, like assembly lines and of course *TSP*. Given two parents, two random crossover points are selected partitioning them into a left, middle and right portion. Then it behaves the following way, child1 inherits its left and right section from parent1, and its middle section is determined.

CX: CX is cyclic crossover, it performs very badly on both the problems and hasn't been plotted. CX2 is just a variant of CX
For more reference on crossovers please refer to Hands on Explanation and this Arxiv paper

Here is the comparative plot of various crossovers algorithms on NQueens

The following figure shows the Outcome of the algorithm averaged over 100 runs, primarily to reinforce the point that the algorithm doesn't just perform well due to luck. If other average runs are needed please do let me know.

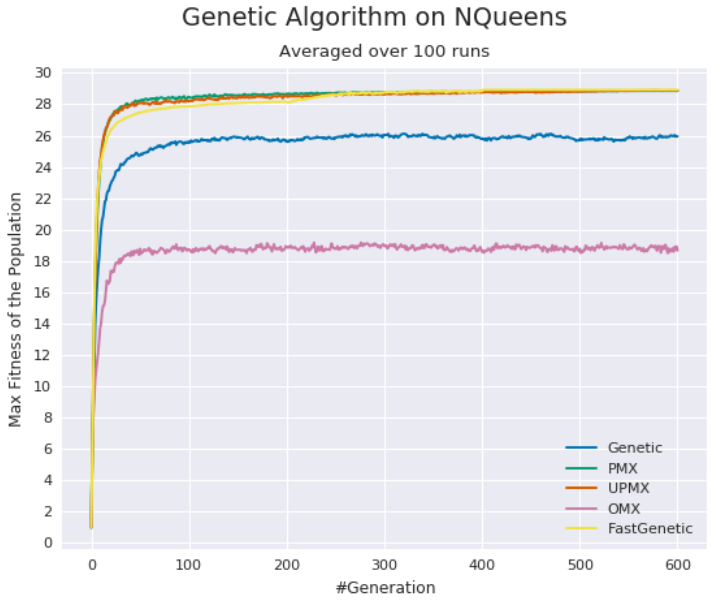

Figure 2.2: Comparison of various crossovers on NQueens

It is clear that PMX, and UPMX, and Fast Genetic Algorithm (which is a custom flavour of algorithm crossover where we iterate all possible indices to create children instead of choosing just one in the naive algorithm, and returning the fittest child) OX fails by a lot, because OX is meant to exploit ordered data, which NQueens clearly is not. We will see how it performs on TSP!

2.5 Performance Compared to Baseline

Here the optimized algorithm refers to the algorithm with PMX Crossover on NQueens *The following figure shows the Outcome of the algorithm averaged over 100 runs, primarily to reinforce the point that the algorithm doesn't just perform well due to luck. If other average runs are needed please do let me know.*

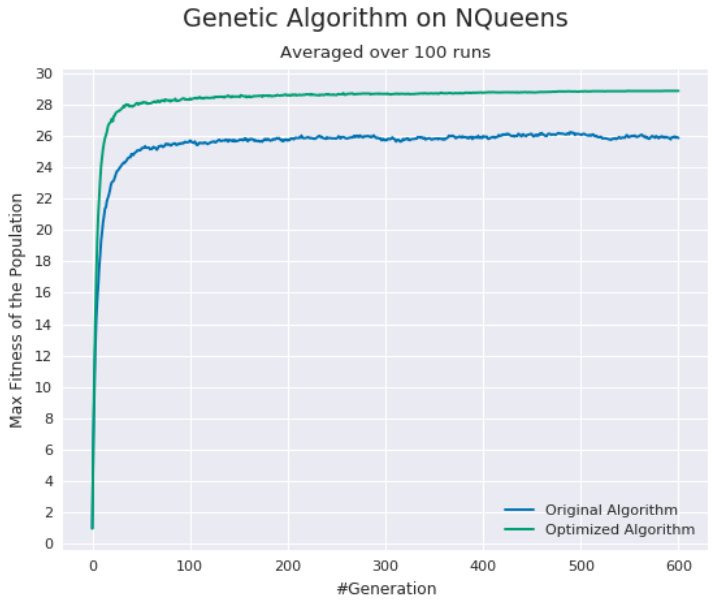

Figure 2.3: Comparison between original and optimized Genetic Algo on NQueens

3 Genetic Algorithm on TSP

3.1 Introduction

The TSP or Travelling Salesman Problem, is a widely famous NP-Complete Problem, meaning we do not know any algorithm that can solve this in polynomial time of the number of cities. First let us look at what the problem is, there is a salesman who intends to visit a certain bunch of cities, and you have to give an the salesman the order in which she/he should visit the cities so that she/he has to travel the least distance, you are given the pair-wise distance between every city.

3.2 State Representation

The obvious representation would be to just list the cities in a list in the order you wish to visit them, and that is precisely what we will do. There is however a small caveat, if we use this state representation then we have to make sure while reproduction and mutation we do not end up creating an offspring which has copies of the same city more than once. Also the cities that do not have any roads between them have been given a high value INF as their mutual distance.

3.3 Fitness Function

We need to define a fitness function based on the path cost as the genetic algorithm we have relies on fitness functions, so one thing we know for certain is that the fitness function should decrease as the path cost increases, one clear solution is to have $fitness(route) = 1/cost(route)$ and this works really well. Some other options could be $fitness(route) = 14 * INF - cost(route)$ and $fitness(route) = e^{-cost(route)}$ but these don't work so well in practice.

3.4 Improvements in the Algorithm

We will proceed in the same way as we did in NQueens, let us look at the constituent functions of the training loop

3.4.1 Improving MUTATE

As seen in NQueens messing with the actual mutation is not a very good idea, however here we have a great option, the path cost and hence the fitness of a route is same as any other cyclic permutation of the route, so after mutating it we will rotate the child by some random number, to provide diversity in the population.

3.4.2 Improving RANDOM-SELECTION

As discussed earlier greedy selection of the parents is not such a good idea, and in this case softmax was little unstable primarily due to technical reasons as eponentiaiting

floats is not so perfect. Hence we will stick to the original version

$$p(route) = \frac{fitness(route)}{\sum_{route \in population} fitness(route)}$$

3.4.3 Improving REPRODUCE

Coming back to the heart of the algorithm, this particular step has a lot of research done on it! We will use the same ones as shown in NQueens and as was done in NQueens also a custom FastTSPAlgo which instead of randomly selecting both start and end index of the parent to pass to the child it iterates over all start indices and randomly selects the end index and returns the best child among them.

Here is the comparative plot of the same

The following figure shows the Outcome of the algorithm averaged over 100 runs, primarily to reinforce the point that the algorithm doesn't just perform well due to luck. If other average runs are needed please do let me know.

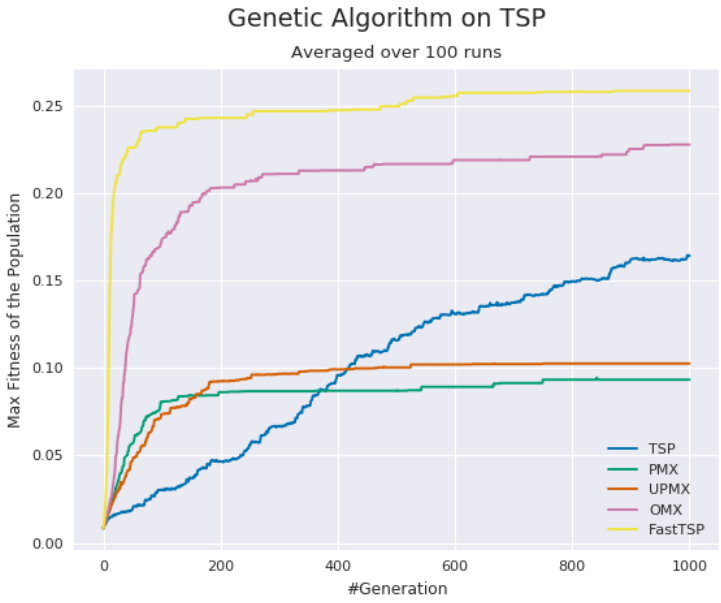

Figure 3.1: Comparison of crossovers on TSP

As expected OX obliterates even all the other crossovers (except FastTSPAlgo) mainly because TSP has order to its state representation and OX exploits it really well! PMX and UPMX don't work so well as they do partial selection of items, and as expected it won't be good for routes where items are cities.

3.5 Performance Compared to Baseline

Here the optimized version is FastTSPAlgo explained in the previous section

The following figure shows the Outcome of the algorithm averaged over 100 runs, primarily to reinforce the point that the algorithm doesn't just perform well due to luck. If other average runs are needed please do let me know.

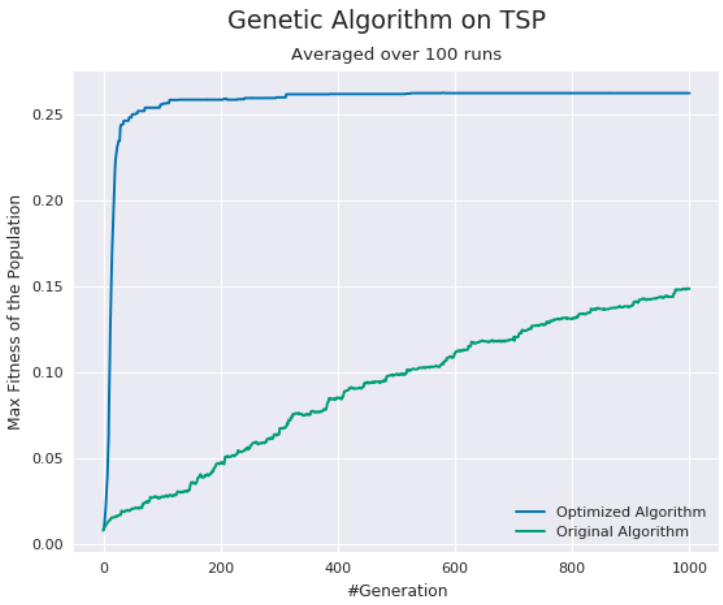

Figure 3.2: TSP Comparison to Baseline

The actual optimal fitness value is 0.267 which is reached by our optimized algorithm

4 Conclusion

Genetic Algorithms have huge inspirations from biology and evolution all the way from the crossover methods to the very core idea of improvement over generations. It was a really interesting assignment to work on opening wide possibilities in future applications to such problems!