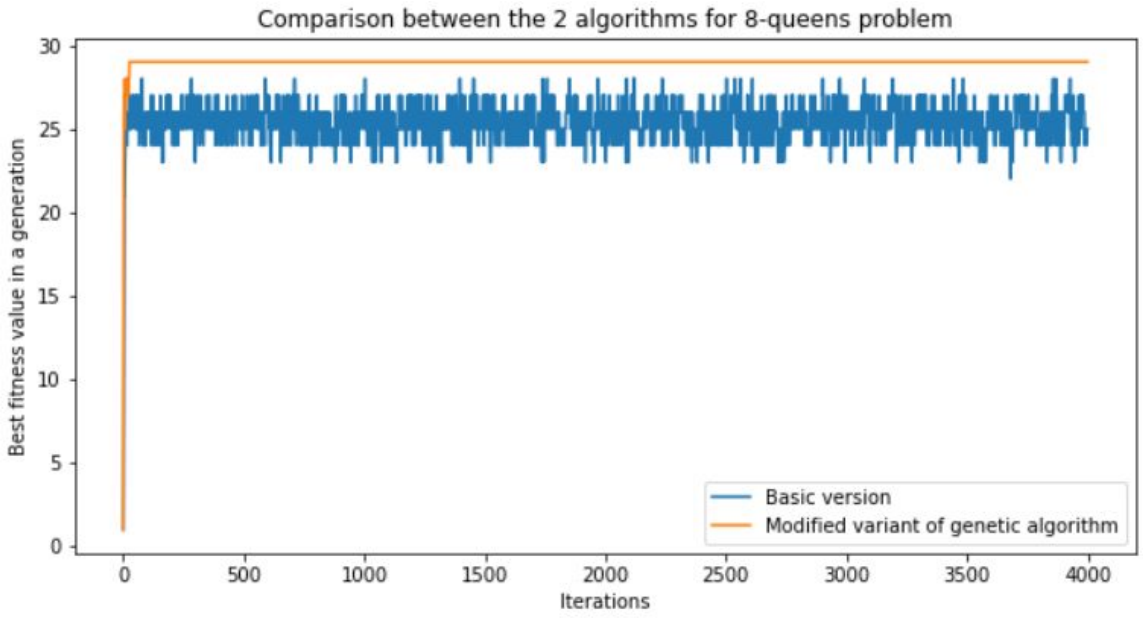
AI Programming Assignment 1 Report

- by Shreyansh Joshi
2018A7PS0097G

8 QUEENS PROBLEM -



Note: Since the modified variant finishes execution normally within 50 iterations, whereas the basic version needs upto 5000 or even more at times, so to plot this graph I ran both algorithms for a stipulated number of iterations (4000). This shouldn't be a problem since the purpose of the graph is to compare the progress made in both the versions.

Fitness function = 1 + No. of pairs of queens not in attacking position

To improve the basic version of 8-queens problem, I did the following -

1. **Increased the initial population size to 200.** Increasing the population size ensures that in each iteration, there are more children generated and more number of mutations (since we iterate over the population size). As a result, the chances of finding the best solution in a lesser number of iterations are much more (at the expense of running time however). I wrote a program that took different population sizes, ran each for 10-15 times, and found the average number of iterations required to converge. Although increasing population over 200 would maybe reduce the iterations even more (by not much though), the execution time shoots up a lot then. Hence, I decided to fix it to 200.

2. **Crossover function -** I manually tried a plethora of crossover functions, by dividing the parents into multiple fragments (say k), and then combining them to produce a kid. However, I got the best results when k=2. So, I produced 2 children there of the form A1B2 and A2B1 (if parents were A1B1 & A2B2). The kid with higher fitness value was selected for further mutation.

3. **Mutation function -** Since the goal of mutation is to bring randomness into the population, mutation in my code works by randomly choosing a column whose queen should be mutated and then choosing a row randomly, where that queen would be moved. I avoided the approach of looping over all 64 possibilities to figure out which mutation gives the best fitness value and choosing it, as that is a greedy approach (can lead to algorithm getting stuck in local maxima). Moreover, it defeats the purpose of mutation - *bringing randomness.*

   I experimented a lot with the probability of mutation. A value of about 0.4-0.5 seemed to work pretty well. Then, I mutate the chosen kid twice and choose the kid that has a higher fitness value between the two after mutation. I did not compare the mutated kids with the kids before mutation and decide whether to choose the mutated one or not (based on the fitness value), as doing this means we are always choosing the better option available. This again might lead to getting stuck in local maximum, as a slightly worse seeming option now, might get us to the solution later on. In short, a greedy approach would be bad.

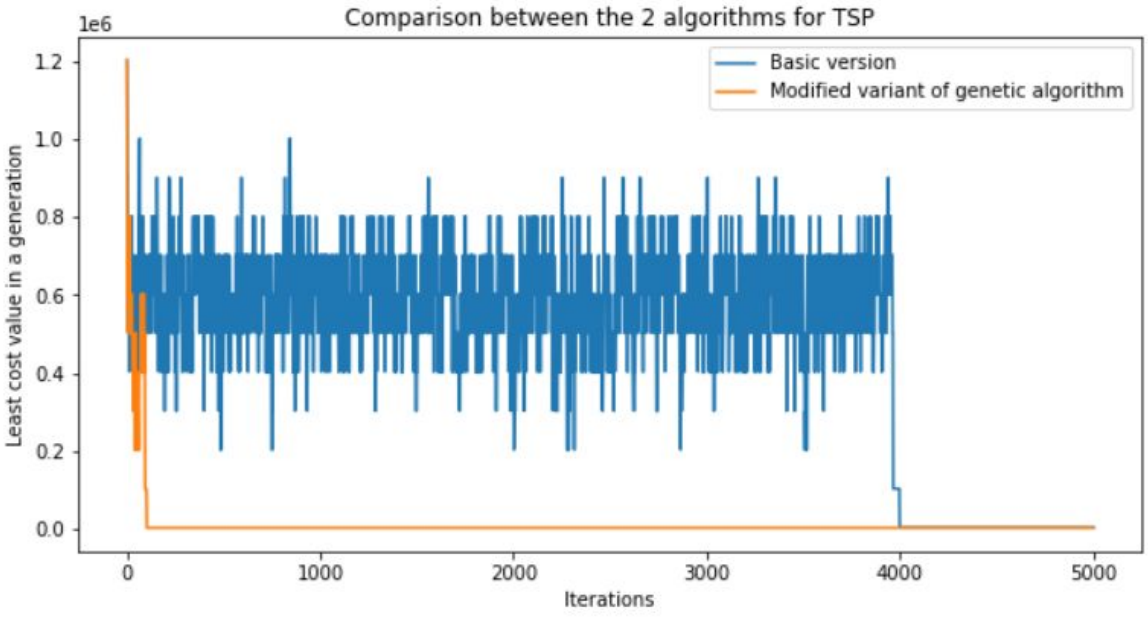
Note: Since the maximum fitness value (29) for 8-queens problem can be found manually quite easily, my code for this question stops running when the maximum fitness value is attained.

TRAVELLING SALESMAN PROBLEM (TSP)-

Note: In my code for TSP, I have used 'cost' function instead of 'fitness' function, as the cost function (the actual path cost) is much more intuitive here. The only place where a sort of fitness function was used was during crossover - choosing 2 parents with weights inversely proportional to their cost values. There I divided 1000000 by the path cost, to get the corresponding fitness function and used that as weights to choose the 2 parents. This ensures that states with higher fitness values (and hence lower cost values) are more likely to become parents for mating.

Cost function = Round trip cost of a particular path

So, the graph below has the best (least) cost value obtained in a generation on the y axis.



It's quite clear from the graph that as in the case of the 8-queens problem, the basic version here is extremely noisy and takes a very long time to converge as compared to the improved and modified version.

To improve the basic version, I did the following -

1. **Increased the initial population size to 1000.** As explained before, increasing initial population size ensures that the algorithm converges to optimum in lesser iterations. Since, the number of paths (possibilities) is huge in TSP (14! for our problem), it behoves to keep the size somewhat larger (1000) than what was in the 8-queens problem (200).

2. **Crossover function -** Here I use the method described in the question. I tried fragmenting the initial parents more than 2 times, but the results were suboptimal. So, I stuck with 2. So, 2 children are produced by crossover. The kid with higher fitness value was selected for further mutation.

3. **Mutation function -** As said previously, to enable mutation to perform its work to the maximum by bringing randomness, I mutate by randomly swapping 2 elements in the list. This is done with a probability of about 0.47 (even this parameter was tweaked to find the best). The chosen child from crossover undergoes mutation twice in each iteration (to increase the randomization so that convergence happens more quickly). And the mutated path with lower cost is chosen and appended to the new population list. Like in 8-queens problem, I avoid swapping all pairs (n*(n-1)/2) in the list and choosing the mutation which has the lowest cost value, as this is a greedy approach and has a good chance of getting stuck in local optimum.

   Initially, I had also tried to mutate by reversing multiple subarrays (>=1) randomly. However, none of them seemed to work well - they added much more noise instead. The results fluctuated wildly, requiring from 100 iterations at times to even more than 80,000 iterations.

Note: Since the minimum cost path for TSP is not so easy to find, and hence, to avoid hardcoding, my code for this question stops running when there is no improvement in the minimum cost value of a generation for 100 iterations.