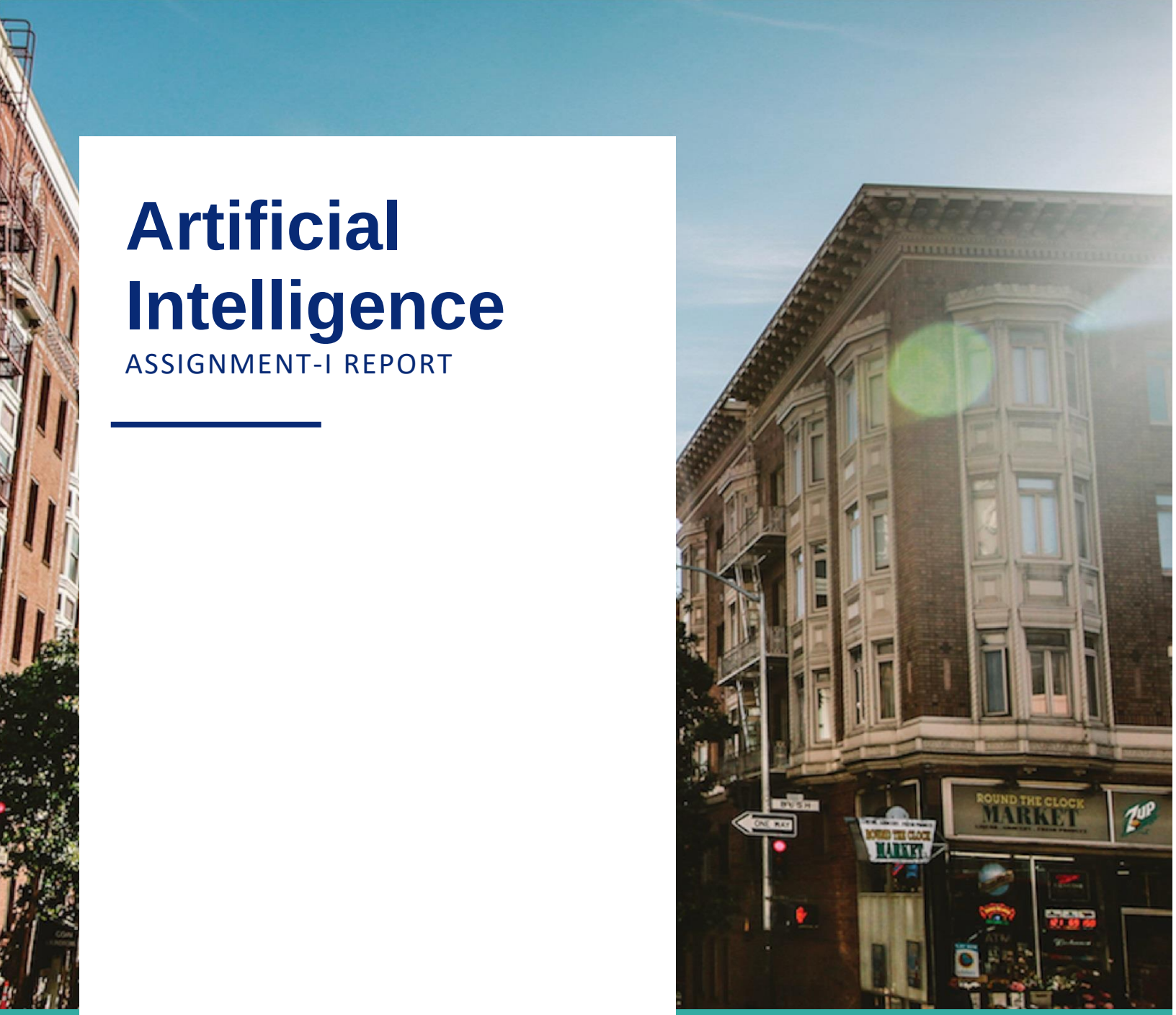
Artificial Intelligence

ASSIGNMENT-I REPORT



FEBRUARY 14

Task 1

Using Modified Genetic Algorithm to solve the 8 Queens Problem.

Aim: Modify the Textbook Genetic Algorithm such that the best fitness value in a population improves in a faster manner over successive generations.

Pre-Conditions Given in question:

- Fitness function must be 1 + *number of pairs of queens not in attacking position.*
- All the Queens in the initial population are in the same row.
- Initial Population size is limited to 20.
- State representation is an array where array[**i**] is the row number of Queen in column **i**.

Modifiable Paradigms and Functions

- Population size
- Reproduce Function
- Mutate Function

Modifications Done:

Population size

The population size was increased by 2 from one generation to another starting from 20. A population limit of 150 was kept as an upper bound.

After running several experiments, I would like to lay emphasis that increasing population at a faster rate across generations and increasing the upper bound on population size leads to a faster solution on average. For example, after some experiments I realized increasing population size by 5 from one generation to another with population cap of 500 resulted in an optimal solution 35% faster, in terms of number of generations, than the above setting.

Pseudo Code:
Size of new population = max (size of old population + k, n), where k and n are some constants.

Intuitive Reasoning of why this works:

- Increasing population increases the probability of an optimal solution as the prospective candidates for an optimal solution increases with population.
- In other words, there is a better probability of an optimal solution in a population of size 150 than a population of size 20.
- Hence the number of generations needed for getting an optimal solution reduces significantly.

What didn't work - Cons of Increasing population size

- The reduced generations needed for Optimal solution doesn't translate into lower run-time of the algorithm each time.
- While we get optimal solution in an earlier generation in this setting, the run time of the algorithm increases if we don't control the population rate as it takes longer to run the one iteration(generation) as population increases.
- Hence, the rate of change of population size was kept at 2 and an upper bound of 150 was put in place to balance the time taken and minimize number of generations.

Reproduce Function

In the textbook version, the probabilities of selection for the parent was linearly proportional to the fitness values. I modified this to quadratic by raising the weights by a power of two. This ensures that the probability of selection of states with higher fitness values is exponentially more than the states with lower fitness value.

Weight of the state is directly proportional to the square of fitness value.
Pseudo code:

```
function make_weights(population, fitness_fn):
    fitness_values = [power(fitness_fn(state),2) for state in population]
    return fitness_values
```

Intuitive Reasoning of why this works:

By increasing the probabilities of states with higher fitness values, we give more incentive for the algorithm to pick better states as parents.

What didn't work

- I also conducted experiments on higher powers like 5 and 10 instead of 2. But the algorithm failed miserably at such high exponents as higher fitness value states dominated the population and there was no scope for any variation in the population.
- Hence I finally choose the exponent as 2.

Mutate Function

The original version has a constant mutation probability across all generations. In the modified version I exponentially decay the mutation probability with a decay factor of 0.95 starting from an initial value of 0.9 and a lower bound value of 0.2.

Pseudo code:

*Mutation Probability = 0.95*mutation Probability, if (0.95*mutation Probability) > 0.2 else 0.2*

Intuitive Reasoning of why this works:

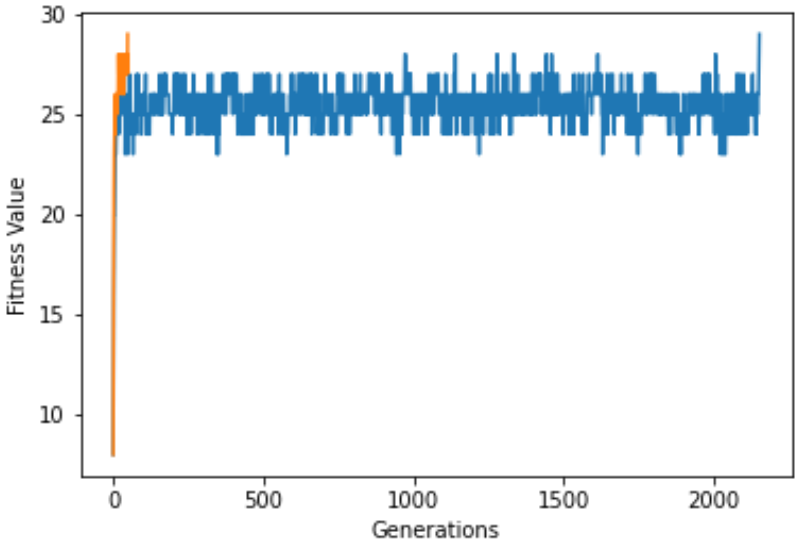
- The initial population has individuals/states of very low fitness values. Hence mutations can bring more variation in population in earlier generations.
    - This ensures that states with higher fitness values are attained faster and show up in earlier generations.
- But at later stages, large amount of mutations can decrease the fitness values of states and produce unnecessary noise in the state encoding. Hence the maximum fitness in a generation falls if mutation probability is too high
    - Do note that, mutation probability below 0.1 lead to the algorithm being stuck in local maximum.
    - Hence a minimum lower bound of 0.2 was kept.

What didn't work

- I tried increasing the number of mutations in a child from one to many (2, 3 etc) for a very low probability.
    - But this generated too much noise in the state and lead to the algorithm under-performing.
- I tried changing the mutation algorithm similar to what's done in mutation algorithm of Travelling salesman problem. But after experimenting I realized it generates too much noise and the variance of fitness values among the states in a generation increases drastically.
    - Too much noise is generated in N-Queens by using the TSP mutation algorithm.

Graph for N-Queens

Clearly the modifications (plotted in orange) have resulted in reaching an optimal solution much faster than textbook version (plotted in Blue).



Other Modifications which may have led to better results

The below couldn't be implemented due to constraints in question.

- Having variation in initial population.
    - We can start with random states with arbitrary start encodings.
    - We can start with a larger initial population.
- Changing fitness functions
    - Use some exponent to exponentially increase fitness values. This gives better probabilities for states with higher fitness.

Task 2

Using Modified Genetic Algorithm to solve the Travelling Salesman Problem

Aim: Modify the Textbook Genetic Algorithm such that the best fitness value in a population improves in a faster manner over successive generations.

Pre-Conditions Given in question:

- All the States in the initial population have the path '*ABCDEFGHIJKLMN'.*
- Initial Population size is limited to 20.
- Distance Matrix is already given.
- State Path encoding is fixed.

Modifiable Paradigms and Functions

- Population size
- Mathematical Value for infinity
- Fitness function
- Reproduce Function
- Mutate Function

Modifications Done:

Population size

The adjustments done are similar to the N-queens modifications. Just with minor variation.

The population size was increased by 5 from one generation to another starting from 20. A population limit of 300 was kept as an upper bound.

For intuition on why this works and for modifications which didn't work, please refer N queens' version.

Value of Infinity

I tried various methods to represent infinity. But it's important to note that we need to be able to differentiate between path with say, one infinity and path with say, 4 infinities. Hence it was necessary for the infinity to be a large but finite value. Hence I chose an arbitrary value of 1000 which is larger than values of distances in matrix but small enough

What didn't work

- I tried math.inf from python math library.
    - But this couldn't differentiate between one infinity and many infinities.
        - Because math.inf + math.inf = math.inf
- I tried large values like 10^10 etc., but this lead to integer overflow in python.

Fitness Function

Unlike N queens, we were given the liberty to choose our fitness function.
I chose the fitness function,

$$(1000/distance)^{0.8}$$

Intuition of why this works:

- The numerator 1000 ensures that if there is even one infinity in the path, then the fitness value remains below one. Hence the numerator was chosen to be equal to the infinity value we chose earlier.
- The distance in denominator ensures that fitness function is inversely proportional to the distance cost.
- The exponent 0.8 ensures that when there are no infinities in path the fitness values don't shoot up to large values.
    - That is, we ensure that path with one infinity or few infinity has a fitness values close to path with no infinity.
        - This ensures the algorithm isn't stuck at local maximum.
- The exponent of 0.8 also ensures that path with higher infinities have exponentially lower values than paths with fewer infinities.

Reproduce Function and Mutate Function

Implemented exactly similar to N-Queens version.

For intuition on why this works and for modifications which didn't work, please refer N queens' version.

Heuristics for Stopping Condition and Finding Optimal Solution

For stopping condition, I made a heuristic where if the maximum fitness in a generation has occurred earlier in at least 15 previous generations and also the path with this maximum fitness has no infinity value in distance then stop the algorithm.

Why this works

- The number of optimal solutions are far fewer than 14! Permutations possible. Hence the algorithm gets stuck in local maximum and the probability of getting a better solution is very low if the above conditions have already been met.

Graph for Travelling Salesman problem

Clearly the modifications (plotted in orange) have resulted in reaching an optimal solution much faster than textbook version (plotted in Blue). In fact, the textbook version didn't give optimal solution with no infinity, even after 1000 generations, this is clear from fitness value of below 1.