جامعة الإسكندرية

ALEXANDRIA UNIVERSITY

كلية الحاسبات وعلوم البيانات

Computing and Data Science

2th Year

Dr. Yasser fouad

Name: Ali Mohamed Sayed Ahmed

ID: 20221449583

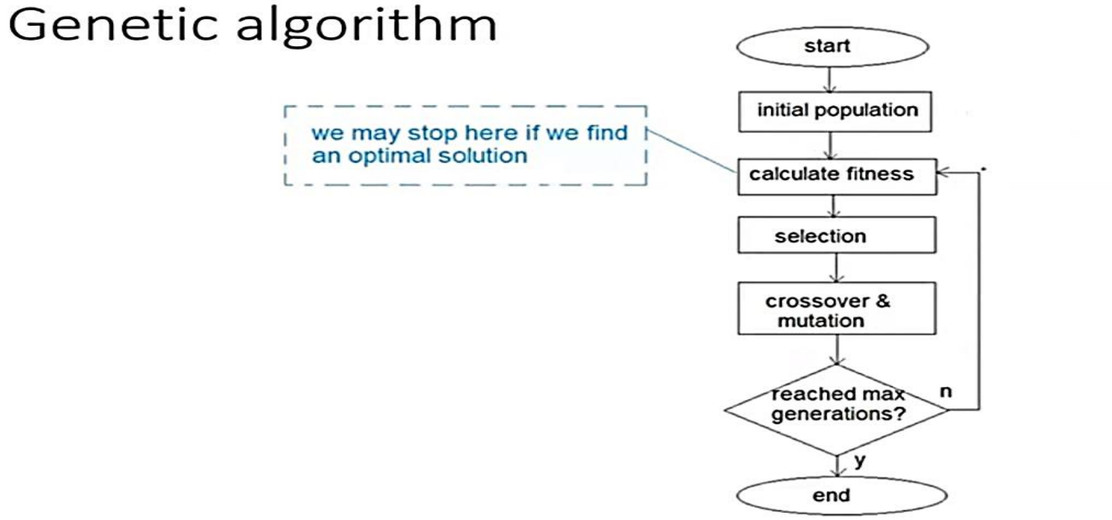
ABSTRACT

A genetic algorithm is one of a class of algorithms that searches a solution space for the optimal solution to a problem. This search is done in a fashion that mimics the operation of evolution – a "population" of possible solutions is formed, and new solutions are formed by "breeding" the best solutions from the population's members to form a new generation. The population evolves for many generations; when the algorithm finishes the best solution is returned. Genetic algorithms are particularly useful for problems where it is extremely difficult or impossible to get an exact solution, or for difficult problems where an exact solution may not be required. They offer an interesting alternative to the typical algorithmic solution methods, and are highly customizable.

How algorithm do?

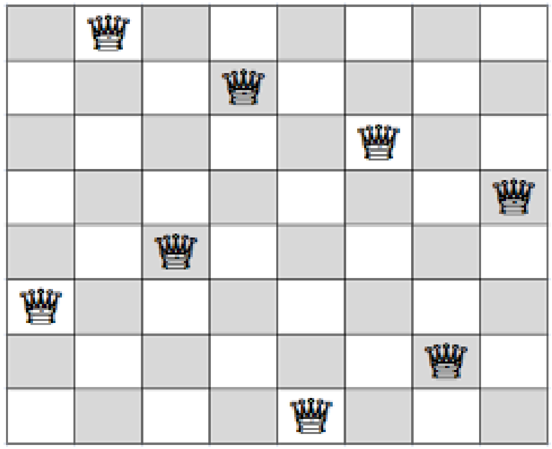
- **Five phases are considered in a genetic algorithm**
  - **Initial population.**
  - **Fitness function.**
  - **Selection.**
  - **Crossover.**
  - **Mutation**

Flow chart:



<u>Problem:</u> 8 queens

- **Placing 8 chess queens on an 8x8 chessboard so that no two queens threaten each other**



To solve these problem:

- **Individual (solution) representation**
  - We can represent a single solution as array of 8 values

1) Initial population:

- By initializing random N individuals we obtain matrix (N x 8) where each row represents an individual

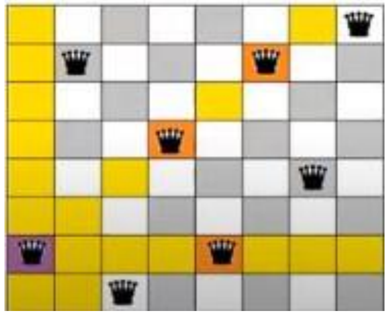
- The code :

create population

```
In [26]: def init_pop(pop_size):
             return np.random.randint(8,size=(pop_size,8))    ##put random value from 0 to 7
                                                               ##size 4 row and 8 column
                                                               ##EX [4,1,2,3,4,5,7,6]
```

2) Fitness function:

- **Penalty of single queen = number of queens she can see**



- We want to maximize fitness (minimize penalty)
  - Fitness = -penalty
- For queen(i) : we check 3 location at column (j)
  1. r
  2. r-d
  3. r+d

  where r is the row of queen and d is distance between column (i) and column (j)

- The code :

to calc fitness of pop

```
In [27]: def calc_fitness(population):
             fitness_vals= []
             for x in population:
                 penalty=0    ##number of queen she can see
                         ## we want high fitness (low penalty)
                 for i in range(8):
                     r=x[i]
                     for j in range (8):
                         if i==j:     ##do not to check the column that it is in this column
                             continue
                         d= abs (i-j)
                         if x[j] in [r,r-d,r+d]:  ##check 3 location (note r is row of queen and d is distance between 2 column)
                             penalty+=1
                 fitness_vals.append(penalty)
             return -1*np.array(fitness_vals)   ## fitness(solution) = -penalty
```

3) Selection

- Randomly select N parents such thart parents with higher fitness are more likely to be selected than parents with lower fitness

Note: that selection can selected parent multiple times
So we convert the fitness value to probabilities

- The code:

to selection pop

select high fitness ..> convert fitness to probability and select high probability

```
In [28]: def selection (population, fitness_vals):
             probs=fitness_vals.copy()
             probs+=abs(probs.min())+1  ##select min value and +1 so we not want any zeros
             probs=probs/probs.sum()    ##divided each value by sum all value to convert to probability
             N=len(population)
             indices=np.arange(N)
             selected_indices=np.random.choice(indices,size=N,p=probs)
             selected_population=population[selected_indices]
             return selected_population
```

4) crossover

- We apply crossover between each 2 parents
- Probability of crossover: pc
- The code:

to crossover every 1 pop with other pop ¶

```
In [29]: def crossover(parent1,parent2,pc):
             r=np.random.random()
             if r < pc:
                 m=np.random.randint(1,8)   ##select index from 1 to 7 to make crossover
                 child1=np.concatenate([parent1[:m],parent2[m:]])
                 child2=np.concatenate([parent2[:m],parent1[m:]])
             else:
                 child1=parent1.copy()
                 child2=parent2.copy()
             return child1,child2
```

5) Mutation

- With very small probability, randomly modify one value of an individual
- The code :

method to mutation every pop

```
In [30]: def mutation(individual, pm):
                 r=np.random.random()   ##return value in range (0 to 1 )
                 if r < pm:    ##check if r less than prob_mutation ..> make mutation
                     m=np.random.randint(8)  ##return value in range from 0 to
                     individual[m]=np.random.randint(8)     ##return value in range from 0 to 7
                 return individual       ##if r not less than prob_mutation...> return pop copy
```

- Conclude we put it all together

```
In [32]: def eight_queens (pop_size,max_generations,pc=0.7,pm=0.01):   ##stop when fitness equal 0
             population=init_pop(pop_size)      ##call function of make pop
             best_fitness_overall = None    #create variable to store best fitness
             for i_gen in range (max_generations):   ##run to max_generation or best_fitness =0
                 fitness_vals=calc_fitness(population)
                 best_i=fitness_vals.argmax()   ##argmax()...>return the index of max value
                 best_fitness=fitness_vals[best_i]  ##return value of index of max value
                 if best_fitness_overall is None or best_fitness > best_fitness_overall:
                     best_fitness_overall=best_fitness
                     best_solution= population[best_i]
                 print (f'\ri_gen={i_gen+1:05}   -f={-best_fitness_overall:03}',end="" )  ##best_fitness_overall return negative value
                 if best_fitness == 0:    ##if fitness equal 000 it is optimal solution
                         print("\nFound the optimal solution")
                         break
                 selected_pop=selection(population,fitness_vals)
                 population = crossover_mutation(selected_pop,pc,pm)
             print(best_solution)
```