Analysis of Genetic Algorithm to solve the 8-Queens Problem

Robert Goes

Abstract— **Analysis of two different genetic algorithms being used to solve the 8-Queens problem, one using permutation to represent the chess board, and the other using a set. The algorithm using the permutation converged on a correct solution the fastest.**

I. INTRODUCTION

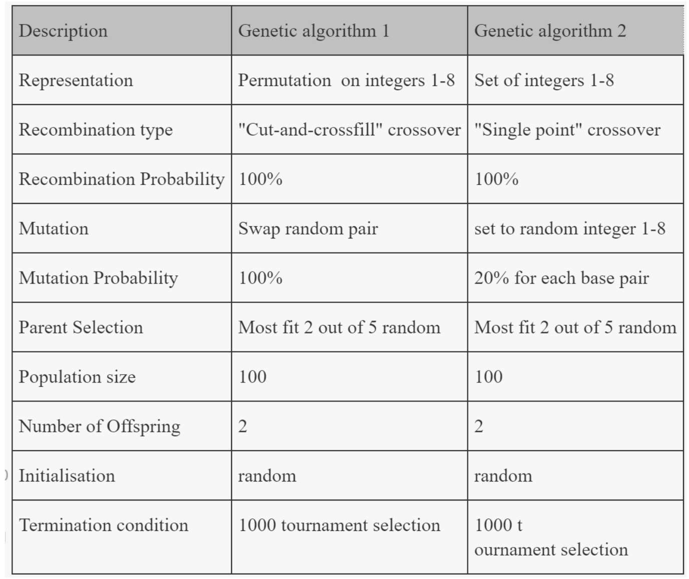
The problem considered here is known as the 8-Queens Problem. It can be stated as follows: You have an 8x8 chessboard, and 8 queens that can attack all pieces in the same row, column, and up and down diagonal as them. The problem consists of trying to place these 8 queens onto a chessboard in a configuration such that no two queens are attacking each other. Without simplification, there are 64 choose 8 ways (4,426,165,368) to place the 8 queens on the board, and only 92 solutions. In order to simplify this problem more, because no two queens can share a column, we can represent the positions as a set of 8 numbers, between 1-8, each number signifying the position of the queen in a given column. This simplification leads 8 to the 8th power possible combinations, 16,777,216. Furthermore, because no two queens can share a row, we can represent the positions of the queens on a given board as a permutation of the set $\{1, 2, , 8\}$. This leads to only 8 factorial boards to check, 40,320.

We will solve the problem by using two different genetic algorithms, briefly stated as follows. The genome will be our chosen representation of the board, different for each algorithm, and each individual will represent a configuration of the board. A fitness function will evaluate how many queens are attacking each other on a given board, and assign a numerical score to each individual. A population of individuals will be created. Then, for a certain number of interactions, two individuals will be selected out of the population, that are determined to have a higher fitness to be parents, and then, recombined with each other two form two more individual that are the children of the parents. These children will be mutated in some way, then replaced into the population. Our methods will ensure the overall fitness of the population increases over time, in hopes that certain individuals will have boards with no queens attacking each other.

II. METHODS

A. *Overview of methodology*

Two different genetic algorithms will be used, labeled Ga1 and GA2 hereafter. Both will be coded in the high level programming language Python. Ga1 will use the permutation representation of the board, and Ga2 will use the set representation. For each trial, 100 individuals will be created, and



Description	Genetic algorithm 1	Genetic algorithm 2
Representation	Permutation on integers 1-8	Set of integers 1-8
Recombination type	"Cut-and-crossfill" crossover	"Single point" crossover
Recombination Probability	100%	100%
Mutation	Swap random pair	set to random integer 1-8
Mutation Probability	100%	20% for each base pair
Parent Selection	Most fit 2 out of 5 random	Most fit 2 out of 5 random
Population size	100	100
Number of Offspring	2	2
Initialisation	random	random
Termination condition	1000 tournament selection	1000 t ournament selection

Fig. 1. Comparison of GA's

run for 1000 selections for a given instance of a genetic algorithm. In order to study Ga1's and Ga2's respective effects on solving the 8-Queens problem, 30 trials will be conducted with each algorithm, and their best, average, and worst fitness will be recorded. A comparison of their differences can be found in Figure 1.

B. *Genetic Algorithm Construction*

When the Genetic Algorithms differ, it will be noted. Both use a Class to represent the individual, population, and the genetic algorithm being run. The individual Class contains a gene, that is a list of numbers, which can either be a set(ga2) or a permutation(ga1). It also contains a fitness function that determines how many queens are in a position to attack another, regardless if another queen is in the way. This is implement by calculating the intercept points for the diagonals created by the queens (and the rows in the case of ga2) and determining if each point is contained in those sets. The individual also contains a recombination function. In the case of ga1, this function picks a point, and copies the first part to the children up until to the crossover point. Then it creates the second part by inserting values from parents, in the order the appear, skipping the ones that are already present, and looping back to the beginning. In the case of ga2, the recombination simply randomly chooses a crossover point, and takes the part up to the crossover point and puts it into the respective child, then puts the rest in the opposite child. The individual class also contains a function to mutate.

In ga1, this function chooses two random base pairs, and swaps them. In ga2, this function, with a 0.2 probability for each base pairs, changes them to a random integer.

The population Class initializes 100 random individuals with the given representation when created. It also contains a function to take sample of n random individuals, and also return the fitness of the best, worst, and also take the average fitness of all individuals.

The genetic algorithm class takes a population class, and a number of iterations, set to a 1000. When the function called run is called, first it will record, the best, worst, and average of the population, then it will call sample on the population, determine 5 random individuals, it will then select the most fit two individuals, and perform recombination on them. Next it will take the two children produced, mutate them, then replace the two least fit individuals of the population with them. This will repeat for the specified number of individuals, 1000.

C. Genetic Algorithm Usage

30 instances of each genetic algorithm were created, each with random populations, and run. Their resulting data, was averaged together for each generation, and a variance was also calculated for each generation.

III. Results

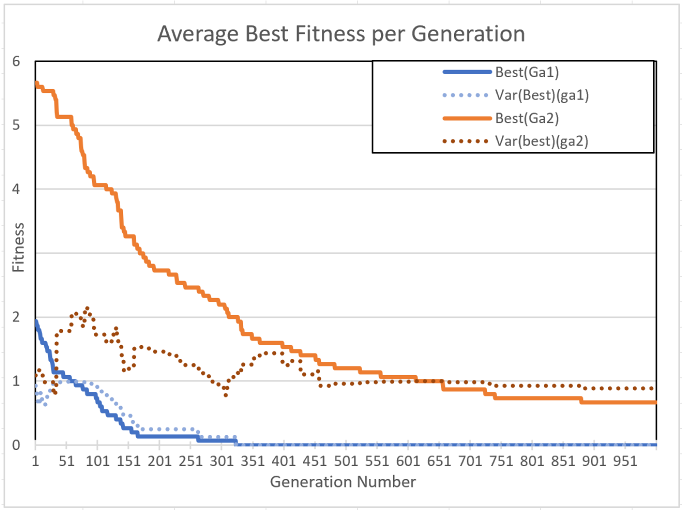


Fig. 2. Fitness of best individual in each generation, averaged over 30 trials. Variance calculated per generation.

The best case fitness in ga1 on average, started off much lower then ga2. This is most likely because the problem space of ga2 is much larger do to the set representation, so more individuals have opportunity for conflict. Ga1 proceed to quickly find a solution after only 351 generations on average, with very little variance among individual trials, as seen in Figure 2. The best individual in Ga2, on the other hand, seems to proceed a a similar rate to ga1, but does not on average achieve a solution in a 1000 iterations, and retains a variance of 1. The average of case of fitness, averaged over all trials for ga2 initially is worse the ga1, but becomes

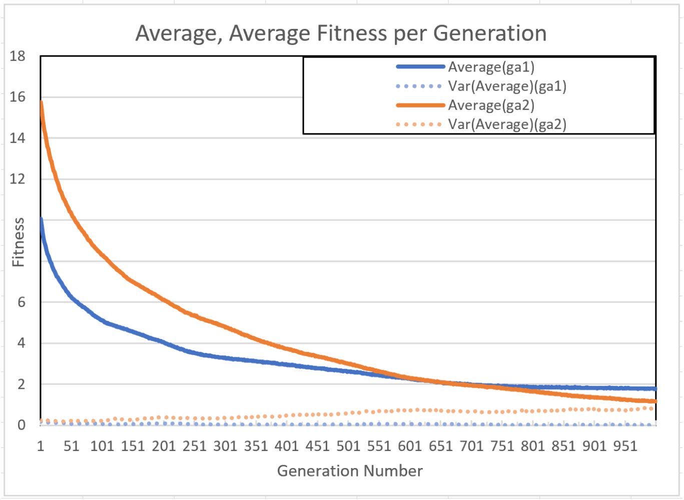


Fig. 3. The average fitness of every individual in a generation averaged over 30 trials, with variance calculated per trial

better after around 750 iterations, with slightly more variance between trials as seen in Figure 3. This could be because ga2 becomes less diverse quickly, as it's crossover method preserves most of the parents genes intact.

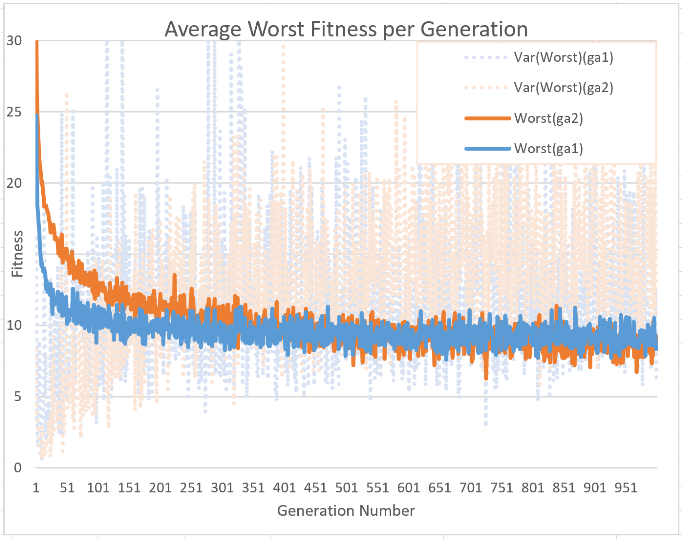


Fig. 4. The worst fitness of a individual in a generation averaged over 30 trials, variance calculated per generation.

The worst case of fitness, averaged over generations, initially starts high for both ga1 and ga2, but decreases rapidly to around 10, staying there for the rest of the generations. The variance remains extremely high for all generations, as seen in Figure 4. The large variations could be caused by the fact single mutation could add a lot of new conflicts to a single individual, and thus the worst individual could become much worse in the space of one generation.

IV. Conclusions

We can see ga1 is superior as it is able to converge on a solution much quicker then ga2. The number of solutions it produced is less then ga2, since it’s average flattens out at 2, and does not keep decreasing. This may not be bad though since ga2 most likely has mostly the same solutions.

1 Genetic algorithm 1

```
"""
begining of Robert Goes's 8-queens problem genetic algrithom 1.
Queen postion is represented as a 8 number long list, for example,
[1,2,3,4,5,6,7,8] reprsents queens going down a diagnal, from the top
left to the bottom right.
 Fitness is calculated as if queens can check through eachother.
 individual works on larger genes, minimal 1 gene
 """

#probabality of mutation occuring
mutation_rate = 1
import csv

import random
def mean(numbers):
    return float(sum(numbers)/len(numbers))

class Individual:
    def __init__(self, gene):
        #forgot to make a copy here, created many issues
        self.gene = gene[:]

    def __repr__(self):
        output = str()
        for value in self.gene:
            output += str(value) + ','
        output += ' - '
        output += str(self.fitness())
        return output

    #swap random pair
    def mutate(self):
        #occurs mutation rate percent of the time
        if random.random() <= mutation_rate:
            #choose to random points, not the same
            pos_a = random.randint(0,len(self.gene)-1)
            #modulus point so it can be anywhere, and make range 1 less then size needed to
            pos_b = (pos_a + random.randint(1,len(self.gene)-2))%len(self.gene)

            temp = self.gene[pos_a]
            self.gene[pos_a] = self.gene[pos_b]
            self.gene[pos_b] = temp
```

```
#ussses cut a cross to make child
def recombination(self, parent_2):
    #select point to do cross over 1-(length - 1)
    crossover_point = random.randint(1, len(self.gene) - 1)

    child_a = self.gene[:]
    child_b = parent_2.gene[:]

    #fill in child a
    offset = 0
    for pos in range(crossover_point,len(self.gene)):
        while parent_2.gene[(pos+offset)%len(self.gene)] in child_a[0:crossover_point]:
            offset += 1
        child_a[pos] = parent_2.gene[(pos+offset)%len(self.gene)]

    #fill in child b

    offset = 0
    for pos in range(crossover_point,len(self.gene)):
        while self.gene[(pos+offset)%len(self.gene)] in child_b[0:crossover_point]:
            offset += 1
        child_b[pos] = self.gene[(pos+offset)%len(self.gene)]

    return (Individual(child_a),Individual(child_b))




def fitness(self):
        fitness_value = 0
        #holds left hand endpoints of up and down diagnols
        down_list = []
        up_list = []

        #calculate the endpoints
        for pos,value in enumerate(self.gene):
                down_list.append(value-pos)
                up_list.append(value+pos)
        #check where every queen falls along the line
        for value in self.gene:
                #use count necause may be in check by more then 1 queen
                fitness_value += down_list.count(value)
                fitness_value += up_list.count(value)

                down_list = list(map(lambda x: x+1,down_list))
                up_list = list(map(lambda x: x-1, up_list))
            #subtract two times the number of queens since each queen will be counted as pu
            return fitness_value - len(self.gene)*2


class Population:
    def __init__(self, size):

        self.members = []
        #base gene to be shuffled
        base_gene = [1,2,3,4,5,6,7,8]

        for n in range(size):
            random.shuffle(base_gene)
            self.members.append(Individual(base_gene))

    #=sample
    def sample(self, size):
        return random.sample(self.members,size)
    def print(self):
        for member in self.members:
            print(member)

    def average(self):
        return sum(member.fitness() for member in self.members)/len(self.members)
    def best(self):
        return min(self.members, key = lambda x: x.fitness()).fitness()
    def worst(self):
        return max(self.members, key = lambda x: x.fitness()).fitness()

class Genetic_Algrithom:
    def __init__(self, population, generations):
        self.population = population
        self.current_gen = 0
        self.max_gen = generations
        self.worsts = []
        self.bests = []
        self.averages = []
    #runs the genetic algrithom
    def run(self):
        for i in range(0,self.max_gen):
            if i%200 == 0:
                print("On generation {}.".format(i))

            self.worsts.append(self.population.worst())
            self.bests.append(self.population.best())
            self.averages.append(self.population.average())
            self.evaluate()
    #Return the best two parents of a population
    def selection(self):
        group = sorted(self.population.sample(5),key = lambda x: x.fitness())

        return group[0:2]

    #gets two valid offspring
    def get_offspring(self, parents):
        child_a, child_b = parents[0].recombination(parents[1])
        child_a.mutate()
        child_b.mutate()
        return [child_a,child_b]

    #replace the worse two in the population
    def survivor_selection(self, childern):
        self.population.members.sort(key = lambda x: x.fitness(), reverse = True)
        self.population.members[0:2] = childern

    #runs one cycle
    def evaluate(self):
        parents = self.selection()
        childern = self.get_offspring(parents)
        self.survivor_selection(childern)




#########################
#run for 30 trials, with 30 populations
Populations = [Population(100) for a in range (0,30)]

Ga_list = [Genetic_Algrithom(Populations[i],1000) for i in range(0,30)]
counter = 0
for Ga in Ga_list:
    print("GA number {}-------".format(counter))
    Ga.run()
    counter = counter + 1

#a.print()

#print results
with open('data_ga1_raw.csv', 'w', newline='') as csvfile:
    datawriter = csv.writer(csvfile, delimiter=',',
    quotechar='|', quoting=csv.QUOTE_MINIMAL)
    for i in range(0,1000):
        bests = [Ga.bests[i] for Ga in Ga_list]
        averages = [Ga.averages[i] for Ga in Ga_list]
        worsts = [Ga.worsts[i] for Ga in Ga_list]
        datawriter.writerow(bests+averages+worsts)
with open('data_ga1.csv', 'w', newline='') as csvfile:
    datawriter = csv.writer(csvfile, delimiter=',',
    quotechar='|', quoting=csv.QUOTE_MINIMAL)
    for i in range(0,1000):
        bests = [Ga.bests[i] for Ga in Ga_list]
        averages = [Ga.averages[i] for Ga in Ga_list]
        worsts = [Ga.worsts[i] for Ga in Ga_list]

        bests_avg = mean(bests)
        averages_avg = mean(averages)
        worsts_avg = mean(worsts)

        bests_var = mean([(i - bests_avg)**2 for i in bests])
        worsts_var = mean([(i - worsts_avg)**2 for i in worsts])
        averages_var = mean([(i - averages_avg)**2 for i in averages])

        datawriter.writerow([bests_avg,bests_var,averages_avg,averages_var,worsts_avg,worsts
```

2 Genetic algorithm 2

```
#probabality of mutation occuring
mutation_rate = 0.2
import csv

import random
def mean(numbers):
    return float(sum(numbers)/len(numbers))

class Individual:
    def __init__(self, gene):
        #forgot to make a copy here, created many issues
        self.gene = gene[:]

    def __repr__(self):
        output = str()
        for value in self.gene:
            output += str(value) + ','
        output += ' - '
        output += str(self.fitness())
        return output
```

```
#mutate random gene
def mutate(self):
    #occurs mutation rate percent of the time
    for i in range(0,len(self.gene)):
        if random.random() <= mutation_rate:
            self.gene[i] = random.randint(1,8)


#crosses over genes at random point to make child
def recombination(self, parent_2):
    #select point to do cross over 1 through (length - 1)
    crossover_point = random.randint(1, len(self.gene) - 1)

    child_a = self.gene[:]
    child_b = parent_2.gene[:]

    child_a[crossover_point:] = parent_2.gene[crossover_point:]
    child_a[crossover_point:] = self.gene[crossover_point:]
    return (Individual(child_a),Individual(child_b))




def fitness(self):
        fitness_value = 0
        #holds left hand endpoints of up and down diagnols
        down_list = []
        up_list = []
        side_list = []

        #calculate the endpoints
        for pos,value in enumerate(self.gene):
                down_list.append(value-pos)
                up_list.append(value+pos)
                side_list.append(value)
        #check where every queen falls along the line

        for value in self.gene:
                #use count necause may be in check by more then 1 queen
                fitness_value += down_list.count(value)
                fitness_value += up_list.count(value)
                fitness_value += side_list.count(value)
                down_list = list(map(lambda x: x+1,down_list))
                up_list = list(map(lambda x: x-1, up_list))
        #subtract three the number of queens since each queen will be counted as puttin
```

```
            return fitness_value - len(self.gene)*3


class Population:
    def __init__(self, size):

        self.members = []

        #make random numers 1-8 for genes, and make in hundreat times.
        for n in range(size):
            self.members.append(Individual([random.randint(1,8) for i in range(0,8)]))

    #=sample
    def sample(self, size):
        return random.sample(self.members,size)
    def print(self):
        for member in self.members:
            print(member)

    def average(self):
        return sum(member.fitness() for member in self.members)/len(self.members)
    def best(self):
        return min(self.members, key = lambda x: x.fitness()).fitness()
    def worst(self):
        return max(self.members, key = lambda x: x.fitness()).fitness()

class Genetic_Algrithom:
    def __init__(self, population, generations):
        self.population = population
        self.current_gen = 0
        self.max_gen = generations
        self.worsts = []
        self.bests = []
        self.averages = []
    #runs the genetic algrithom
    def run(self):
        for i in range(0,self.max_gen):
            if i%200 == 0:
                print("On generation {}.".format(i))

            self.worsts.append(self.population.worst())
            self.bests.append(self.population.best())
            self.averages.append(self.population.average())
            self.evaluate()
    #Return the best two parents of a population
    def selection(self):
        group = sorted(self.population.sample(5),key = lambda x: x.fitness())

        return group[0:2]

    #gets two valid offspring
    def get_offspring(self, parents):
        child_a, child_b = parents[0].recombination(parents[1])
        child_a.mutate()
        child_b.mutate()
        return [child_a,child_b]

    #replace the worse two in the population
    def survivor_selection(self, childern):
        self.population.members.sort(key = lambda x: x.fitness(), reverse = True)
        self.population.members[0:2] = childern

    #runs one cycle
    def evaluate(self):
        parents = self.selection()
        childern = self.get_offspring(parents)
        self.survivor_selection(childern)




#########################
#run for 30 trials, with 30 populations
Populations = [Population(100) for a in range (0,30)]

Ga_list = [Genetic_Algrithom(Populations[i],1000) for i in range(0,30)]
counter = 0
for Ga in Ga_list:
    print("GA number {}-------".format(counter))
    Ga.run()
    counter = counter + 1

#a.print()

#print results
with open('data_ga2_raw.csv', 'w', newline='') as csvfile:
    datawriter = csv.writer(csvfile, delimiter=',',
    quotechar='|', quoting=csv.QUOTE_MINIMAL)
    for i in range(0,1000):
        bests = [Ga.bests[i] for Ga in Ga_list]
        averages = [Ga.averages[i] for Ga in Ga_list]
        worsts = [Ga.worsts[i] for Ga in Ga_list]
        datawriter.writerow(bests+averages+worsts)
with open('data_ga2.csv', 'w', newline='') as csvfile:
    datawriter = csv.writer(csvfile, delimiter=',',
    quotechar='|', quoting=csv.QUOTE_MINIMAL)
    for i in range(0,1000):
        bests = [Ga.bests[i] for Ga in Ga_list]
        averages = [Ga.averages[i] for Ga in Ga_list]
        worsts = [Ga.worsts[i] for Ga in Ga_list]

        bests_avg = mean(bests)
        averages_avg = mean(averages)
        worsts_avg = mean(worsts)

        bests_var = mean([(i - bests_avg)**2 for i in bests])
        worsts_var = mean([(i - worsts_avg)**2 for i in worsts])
        averages_var = mean([(i - averages_avg)**2 for i in averages])

        datawriter.writerow([bests_avg,bests_var,averages_avg,averages_var,worsts_avg,worsts
```