Evolutionary optimization on the N-Queens problem.

Problem

N-Queens problem

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. Only one queen per column.

System design

The program has been developed in Python. The program consists of one file with two classes: Individual and population. See Figure 1.

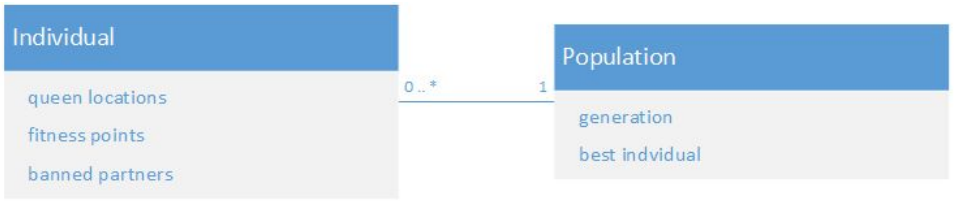

Figure 1. Illustrates the system design of the program.

Representation

The member *queen location* in the class Individual consists of an array of values equal to the number of rows of the board. That is, if the board is of size N x N then the array will be of length N. The indexes represent the queen's row location while the number itself represents the queen's column location. Figure 2 illustrates how the array of numbers can be represented graphically. Where we start counting from 0 to 7 in bottom to top manner.

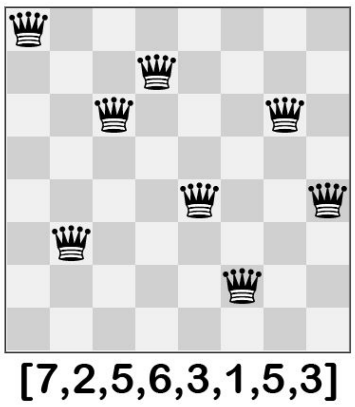

Figure 2. Illustrates how the numbers in the array represent the location of the queen.

The population object contains a list of individuals. The rest of the members in the population class are used to keep track of the current state of the population. The *generation* member is simply a number representing the current generation while the *best individual* member is the individual with the highest points of the current generation.

Fitness function

The fitness function takes in the *queen locations* of an individual and returns a fitness point where a higher value represents a better fitness. The function has a point parameter that are initialized with the maximum number of non-attacking pair of queens, (n * (n-1) / 2). Then the function searches the board for attacking pair of queens and removes one point from the point variable for each pair. The point variable is returned when all locations in the *queen locations* have been checked.

Operations

Crossover - The algorithm uses a crossover method that kills a percentage of the population leaving only the fittest individuals alive. These individuals are then used to reproduce a new population. The individuals will reproduce randomly with each other. The reproduction works by combining the first part of the *queen locations* of the first parent with the second part of the *queen locations* of the other parent. These two parts can then be combined in two ways: Either the part from the first parent is placed at the beginning and the part from the other parent at the end or vice versa. There is an equal probability for both ways.

Mutation - Mutation is implemented by randomly generating a new number from 0 to N-1. The random number is then inserted randomly in an index in the individual *queen locations.* Depending on the mutation factor this is done 0 to N-1 times on the same individual.

Parameter values

Board size - The size of the board. If set to 8 it will generate a N-Queens problem with board size 8x8.

Mutation chance - The chance of a child being mutated in %, 0 - 1.

Mutation factor - To which extent the child is going to be mutated in %. 0 no mutation and 1 is the same as generating a new individual (almost, as the index to mutate is randomized it can happen that the same index is changed more than once under one mutation. If this happens, artifacts from the parents will be left in the child even if mutation factor is 1).

Population size - The number of individuals in the population.

Kill percentage - How many of the least fit individuals in the population that is killed per generation.

Strict mating – Whether an individual can mate with the same individual multiple times as well as if an individual can reproduce asexually (because if strict mating is false no track-keeping of the parents are kept so there is a possibility of an individual mating with themselves).

Timeout generations – The amount of generations that the algorithm will try to find a solution in before returning the best-found solution so far.

Experimental setup

The experimental setup consists of the two code files, the main file, which is the genetic algorithm, see Appendix A, as well as the test file, see Appendix B. In the test-code file one enters the number of times to run the genetic algorithm. It records the time it takes to find a solution for each time and the amount of generations it took to find a solution. In the end of the test the program prints the minimum and maximum time spent on finding a solution as well as the total time and calculates an average to a file. The number of timed out tries and the parameter values used in the test is also printed to the file.

We noticed during development, that the sweet spot of the parameters seemed to be the following:

- Kill percentage = 0.75
- Mutation chance = 0.2 - 0.3
- Mutation factor = 0.5 - 0.7

Therefore, we will do a few tests with the parameters as described above, only chaining the mutation chance as to more easily draw conclusions about just the mutation chance. All these tests will be done with both strict mating turned on and off to see if that will affect the result in any meaningful way. We have also decided that 1000 is a good population size, as that will allow us to run the tests on our computers in a reasonable timeframe while simultaneously giving us satisfactory diversity within the population. We will do this while we increase the board size from 8 up to 25. In the future, it would be interesting to change all the parameters, and draw even more conclusions.

The test-file runs algorithm 10 times and calculates an average value for the number of generations needed for a solution as well as the seconds it took to find a solution. The test also has a roof for how many generations the algorithm tries to find a solution for before stopping, we call this a timeout. The timeout is set to 2000 generations. If the algorithm does not find a solution before the timeout the generations are not counted in the average. However, the number of times a timeout occurred are counted separately. Therefore, we get an overview of when the algorithm starts to timeout as well.

Result from experimentation

The experimentation revealed that after generation 18, the algorithm timeout more than it found solutions. Therefore, we did the testing of different parameters up to generation 18. Then we took the best performing parameters and pushed the algorithm to see how the algorithm did on large board sizes.

The alleged sweet spot we mentioned earlier shown in the graph with blue and green colors (blue with strict mating and green without) does not actually seem to be a sweet spot. The results show that lowering the mutation yields better results. Something the figure 3, figure 4, and figure 5 illustrates.

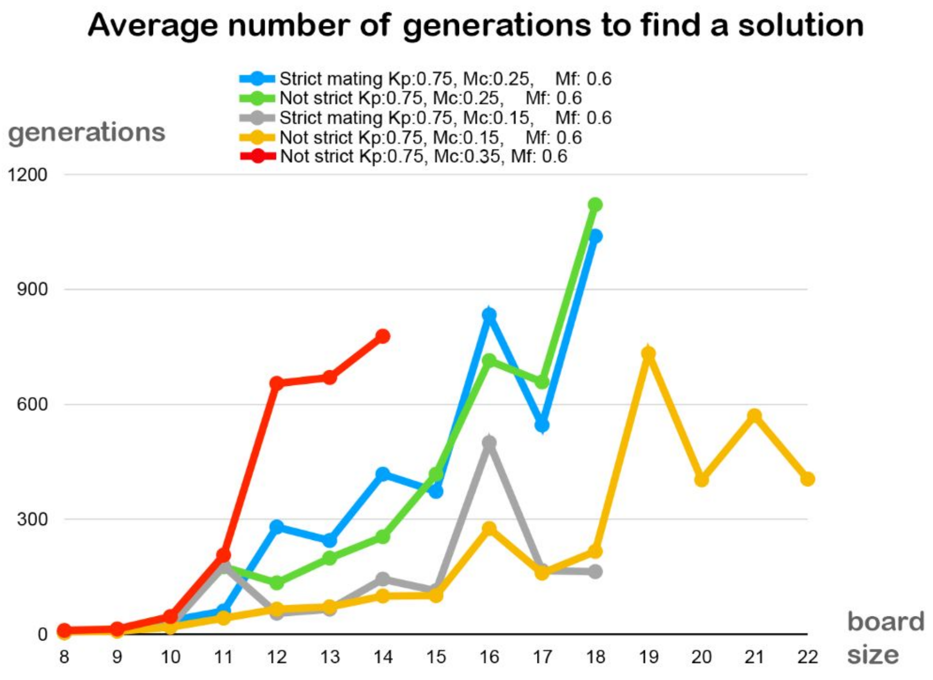

Figure 3. Illustrates average number of generations to find a solution.

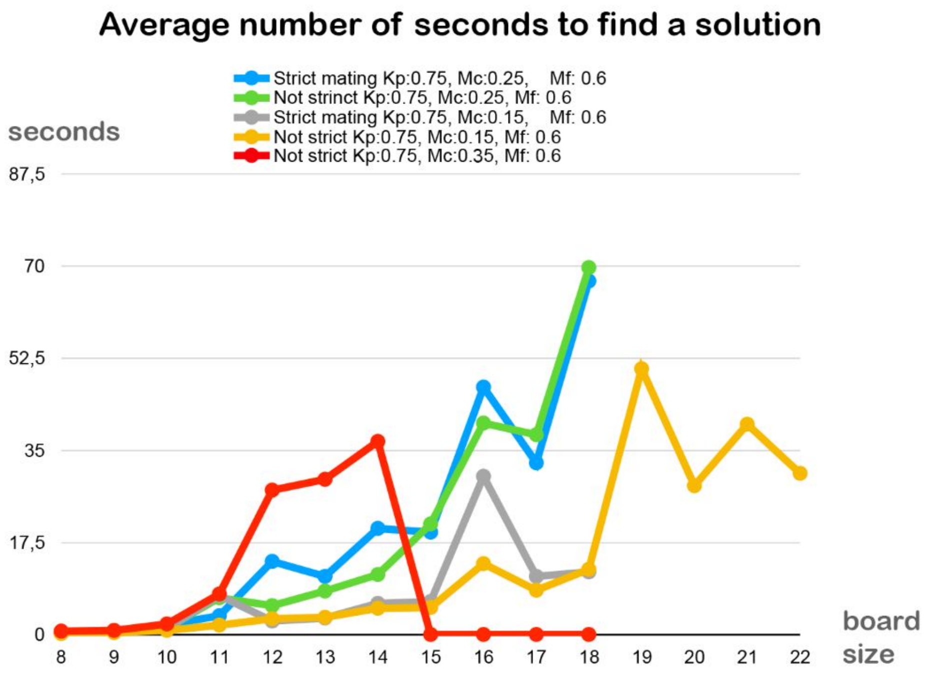

Figure 4. Illustrates average number of seconds to find a solution

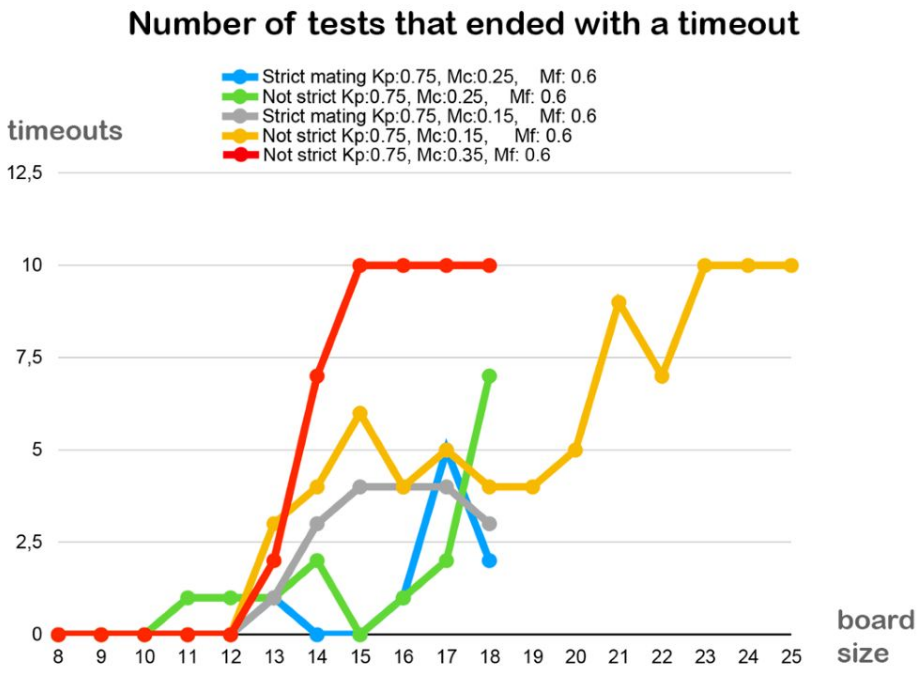

Figure 5. Illustrates number of tests that ended with timeout.

The graphs show that increasing the mutation chance (the red line) worsens the performance of the algorithm in every way. Higher mutation chance increases both the generations and time needed to

find a solution. Worth noting, that with a high mutation chance, no solutions were found after generation 14 with the timeout parameter we had set.
In contrast, decreasing the mutation chance below the initial test mutation chance significantly decreases the amount of generations and time spent on finding a solution (orange line in graph). Furthermore, we start timing out more as the board size increases, but we still occasionally find a solution even at a board size of 22 x 22.

Analys with conclusions

Regarding strict mating, it seems that using strict mating or not does not affect the result too much as the graphs with and without strict mating seem to follow a similar trajectory.
When it comes to mutation, the experiment shows that the best parameter settings were the ones with lower mutation chance, however we only went as low as 15% chance of mutation. We probably could have gone a bit lower and gotten an even better result, but at a certain point the generation would have such a poor diversity that they always get stuck in a local maxima. This is also something we noticed during the experiment. When the algorithm used a low mutation chance and did not find any solution early one, there was a high probability of repeating the same queen locations over and over until the algorithm hit the timeout. This indicates that if a solution is not found in an early generation, the algorithm gets stuck and can’t get anywhere because of low diversity and the mutation can’t get it out of the local maxima.
We also found that our algorithm did not do well on large board size and we expect that to be because of the cross over function. We speculate that the cross over does not have enough variety to create a diverse generation which still produces an overall high fitness score.
In conclusion, a genetic algorithm with low mutation chance is preferable. However, the lower the mutation the higher the chance of hitting a local maxima. To combat this, a well-made cross over function is needed together with the low mutation chance.

Appendix A - Code

```
import random
import operator


boardSize = 8
maxP = int(boardSize * ((boardSize - 1) / 2))
mutationChance = 0.22
mutationFactor = 0.5
numOfIndividuals = 1000
killPrecentage = 0.75
strictMating = True

timeoutGens = 1000

class Population:
    def __init__(self, numOfIndividuals):
        self.indviduals = sorted(self.generateIndviduals(numOfIndividuals),
                                        key=operator.attrgetter("points"))
        self.bestIndividual = self.indviduals[len(self.indviduals) - 1]
        self.generation = 0

    def __str__(self):
        pop = " "
        for invid in self.indviduals:
                        pop += f" {invid} \n"
        return pop

    def generateIndviduals(self, num):
        population = []
        for _ in range(num):
            population.append(Individual())
        return population

    def kill(self, precentage):
        self.indviduals = self.indviduals[int(len(self.indviduals) * precentage):]

    def reproducePopulation(self):
        newPopulation = []
        for i in range(numOfIndividuals):
                        parent1 = self.indviduals[random.randint(0, len(self.indviduals) - 1)]
                        parent2 = self.indviduals[random.randint(0, len(self.indviduals) - 1)]
            while (parent2 in parent1.banedPartners and strictMating):
                        parent1 = self.indviduals[random.randint(0, len(self.indviduals) - 1)]
                        parent2 = self.indviduals[random.randint(0, len(self.indviduals) - 1)]
                        child = parent1.reproduce(parent2)
            newPopulation.append(child)
        self.generation += 1
        self.indviduals = newPopulation
        self.indviduals = sorted(self.indviduals, key=operator.attrgetter("points"))
        self.bestIndividual = self.indviduals[len(self.indviduals) - 1]

    def getPointArr(self):
        points = []
        for indiv in self.indviduals:
            points.append(indiv.points)
        return points
```

```python


class Individual:
    def __init__(self, genetics=None):
        if (genetics != None):
            queenLocations = genetics
        else:
            queenLocations = self.generateQueens()
        self.queenLocations = queenLocations
        self.tryToMutate()
        self.points = self.fitnessPoints()
        self.banedPartners = [self]

    def __str__(self):
        return str(self.queenLocations)

    def generateQueens(self):
        queens = []
        for _ in range(boardSize):
            Queens.appe
nd(random.randint(0, boardSize - 1))
        return queens

    def fitnessPoints(self):
        points = maxP
        for i in range(len(self.queenLocations)):
                        q = self.queenLocations[i]
            for c in range(i + 1, len(self.queenLocations)):
                        q2 = self.queenLocations[c]
                if (abs(q2 - q) is c - i):
                        points = points - 1
                if (q2 is q):
                        points = points - 1
        return points

    def reproduce(self, otherParent):
        invidLen = len(self.queenLocations)
        randCut = random.randint(0, invidLen - 1)
        genetics = self.queenLocations
        if (randCut % 2 is 0):
                        genetics = genetics[:randCut] + otherParent.queenLocations[randCut:]
        else:
                        genetics = genetics[randCut:] + otherParent.queenLocations[:randCut]
        self.banedPartners.append(otherParent)
        otherParent.banedPartners.append(self)
        return Individual(genetics)

    def tryToMutate(self):
        if (random.randint(0, 100) <= mutationChance * 100):
            invidLen = len(self.queenLocations)
            numOfMutations = int(invidLen * mutationFactor)
            for _ in range(numOfMutations):
                randIndex = random.randint(0, invidLen - 1)
                randLocation = random.randint(0, invidLen - 1)
                self.queenLocations[randIndex] = randLocation


```

```python
def geneticAlgorithm(population):
    while (True):

        bestIndividual = population.bestIndividual

        print(
            f"""Generation: {population.generation} has the best child:
                      {bestIndividual.queenLocations}
                      with points {bestIndividual.points} of {maxP}""")

        if (bestIndividual.points is maxP):
            print(bestIndividual.queenLocations)
            return population.generation

        global killPrecentage

        population.kill(killPrecentage)
        population.reproducePopulation()

        if (population.generation >= timeoutGens):
            return population.generation
```

Appendix B - Test Code

```python
from Nqueens import *
import time

def test() :

    totalGen = 0
    totalTime = 0
    minGen = 99999
    maxGen = 0
    minTime = 99999
    maxTime = 0

        timeout = 0

    numOfTests = 10

    for _ in range(numOfTests):

        t0 = time.time()
        gen = geneticAlgorithm(Population(numOfIndividuals))
        t1 = time.time()

        if(gen != timeoutGens):
            timeTaken = t1 - t0

            totalTime += timeTaken
            totalGen += gen

            if timeTaken < minTime:
                minTime = timeTaken
            if timeTaken > maxTime:
                maxTime = timeTaken

            if gen < minGen:
                minGen = gen
            if gen > maxGen:
                maxGen = gen
        else:
                        timeout += 1

    print(
        f"\n\n result printed to file \n")

        with open("testfile.txt", "a") as file:

        if numOfTests-timeout == 0:
            file.write(
                        f"\n\n result \n"
                f"Board size: {boardSize}\n"
                f"Population: {numOfIndividuals}\n"
                f"Kill percentage: {killPrecentage}\n"
                f"Mutation chance: {mutationChance}\n"
                f"Mutation factor: {mutationFactor}\n"
                f"Strict mating: {strictMating}\n"
                f"Nr of tests: {numOfTests}\n"
                f"Nr of gens for timeout: {timeoutGens}\n"
            f"Nr of timeouts: {timeout}.")
        else:

            averageGen = totalGen/(numOfTests-timeout)
            averageTime = totalTime/(numOfTests-timeout)
            file.write(
                    f"\n\n result \n"
                f"Board size: {boardSize}\n"
                f"Population: {numOfIndividuals}\n"
                f"Kill percentage: {killPrecentage}\n"
                f"Mutation chance: {mutationChance}\n"
                f"Mutation factor: {mutationFactor}\n"
                f"Strict mating: {strictMating}\n"
                f"Total Gen: {totalGen}. Average: {averageGen}\n"
                f"Min gen: {minGen}. Max gen: {maxGen}\n"
                f"Total Time: {totalTime} sec. Average: {averageTime} sec\n"
                f"Min time: {minTime} sec. Max time: {maxTime} sec\n"
                f"Nr of tests: {numOfTests}\n"
                f"Nr of gens for timeout: {timeoutGens}\n"
                f"Nr of timeouts: {timeout}.")

test()
```