Assignment 1 - Eight Queens

Joel Doumit
College of Computer Science
University of Idaho
Moscow, ID 83843
Email: doum6708@vandals.uidaho.edu

***Abstract*—This project was to use evolutionary computation concepts and methods to find solutions to the 8 queens problem. Though the 8 queens problem is widely considered to be a solved problem, it makes for the perfect testing ground on which to determine if an evolutionary algorithm can effectively reach a known solution.**

1. Introduction

The Eight Queens Problem is one of the oldest known chess problems. It involves placing eight queen pieces on a standard 8-by-8 chessboard in a configuration such that none of the queens are directly threatening (able to take) any of the others. A standard queen is able to move horizontally, vertically, and diagonally, which is what makes the problem fairly difficult.

1.1. Algorithms and Projected Solutions

The two algorithms follow what is essentially the same method of finding solutions, adapted specifically for each one. The finer details of the algorithms will be discussed in a later section, but the general means of finding the solution are:

- Generating a population
- Assigning each individual in the population a fitness score
- Selecting a subgroup of the population
- Deciding which of the individuals in the subgroup have the best fitness scores
- Creating offspring from selected parents
- Mutating the children
- Removing the two weakest individuals in the population

1.2. Representations

There were two required representations for this assignment: one based on permutations, and one of our choice. I chose combinations as my additional representation, as the method of generating the population using a combination was relatively simple. Additionally, implementing the crossover and mutation functions were actually more straightforward than those for the permutations.

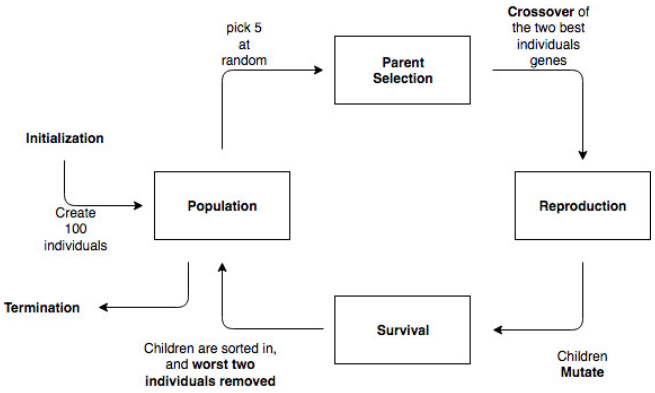

Figure 1.

2. Methods

In this section the algorithms described in section 1 will be detailed further. As previously stated, the algorithms don't differ much in structure. The essential form of each algorithm is shown in Figure 1.

2.1. Permutation

2.1.1. Generation. The permutation version of the algorithm generates the population by calling numpy.random.permutation(8), which generates 8 random numbers without repetition. The algorithm stores them in an array called in an individual, with the numbers inside said array being called genes. It does this 100 times, and stores those arrays in a larger array. This new array is the population. For each individual in the population, the algorithm assigns a score based on how many collisions between queens there are. The lower the score, the closer the individual is to being a solution. It stores this number, called a fitness score, in the population array, next to the individual the score belongs to.

2.1.2. Selection. Five individuals are selected at random and copied into a small array along with their fitness scores. The individuals are organized by said fitness scores, and the two individuals with the best scores are selected for reproduction.

2.1.3. Reproduction. The two best individuals from the selection phase reproduce by means of crossover. Crossover in this case means all of the values in a parent after the chosen crossover point are swapped with all of the values from after the crossover point in the other parent. In the case of permutation, however, we cannot have repeating numbers, so to eliminate repetition we loop back around to the numbers before the crossover to fill in any doubles. The resulting individuals are considered the offspring of the selected parents.

From here, the children mutate. In the permutation version of the algorithm, this is accomplished by swapping two of the genes (numbers) in the individual, again to avoid repetition.

2.1.4. Survival. The freshly mutated children are then assigned a fitness score in the same way as the individuals did during generation. They are then added to the population. At this point, we have two too many individuals in the population, so we sort the population by fitness score, and kill the two with the worst scores, even if those individuals happen to be the children we just created.

2.2. Combination

2.2.1. Generation. Much in the same way as the permutation version, the combination version of the algorithm generates the population using numpy. This time we use numpy.random.choice(8), which generates 8 random numbers, this time allowing repetition. Again, these numbers are saved into an individual array, and the individual is saved into the population array. Once filled, this population is given fitness scores as well.

2.2.2. Selection. This is done in the same way the permutation version. Five individuals are selected and copied at random to a new small array, and the best two as decided by fitness score are chosen to be parents.

2.2.3. Reproduction. Again like the permutation version, the offspring are created via crossover of the parents. This time, however, the method of crossover changes. Because combinations don't care about repeated numbers, the genes swapped from each parent are done one-to-one. No additional looping is required, because any doubles are considered valid.

In the mutation phase, a different approach is taken again. Because repetition is allowed, we simply select one of the genes at random and replace it with a random number (within the allowed range).

2.2.4. Survival. The survival stage is the exact same as the permutation version. The offspring are placed back into the population and the worst two individuals are trimmed away.

2.3. Data Recording

Each of the algorithms is run a total of 30 times for what is considered a full data set. For each of those 30 times, the population is initialized once, and then each phase from selection through and including survival is considered a generation. Effectively, we run 30 tests of 1000 generations to have a complete data set for one of the algorithm variants. This data is written out to a csv file, which is then graphed in Excel. Output from the actual code described in the above sections will be reviewed in a later section.

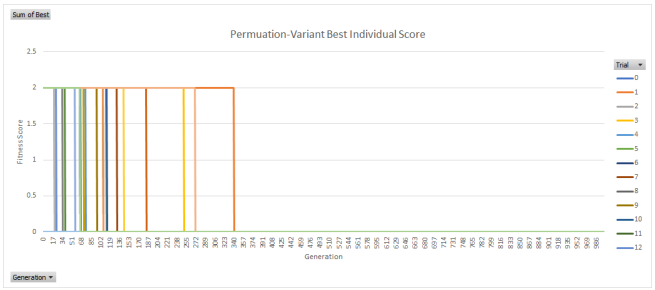

Figure 2.

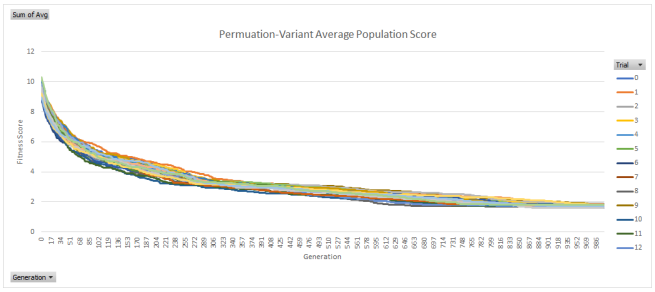

Figure 3.

3. Results

The data from each of the algorithm variants is represented in figures 2 through 7.

The best individual scores for both the permutation and the combination versions of the algorithm effectively show that the algorithm is working. The data represents the number of generations it took for a solution to be found. The range from 0 to 2 represents an individual (namely, the best individual in the population) going from a score of 2 to a score of 0. If a line appears at a particular generation, that

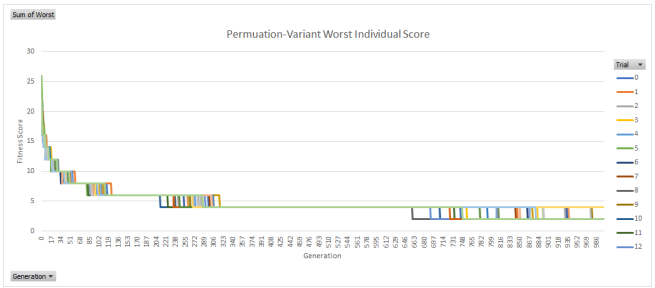

Figure 4.

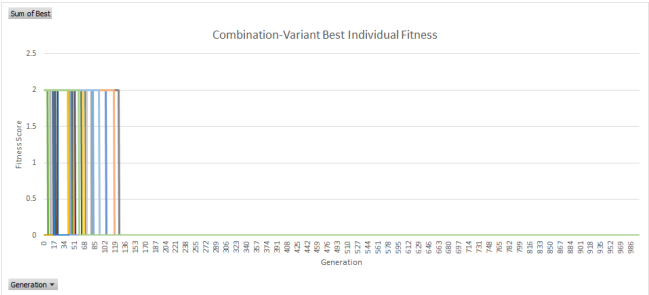

Figure 5.

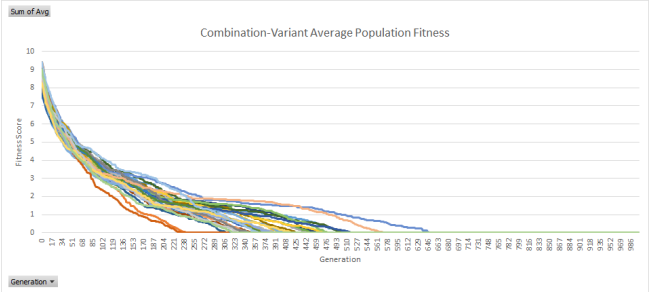

Figure 6.

means it took the population that many generations to get the best individual to become a solution. The fact that so many populations end up with a solution is further evidence that the algorithm works. That being said, it appears that the combination version of the algorithm edges out the permutation version with regard to having the fewest number of generations to get to a solution.

The average population scores support the claim that the algorithm works as well. In both versions of the algorithm the average score shrinks with each passing generation. Again the combination version edges out the permutation version, as every population eventually reaches a point of consisting of all solutions, whereas in the permutation version, the average score approaches 2.

The worst individual scores support the average population scores. Even the worst populations end up reaching a solution in the combination version, and the worst scores still get consistently better in the permutation version.

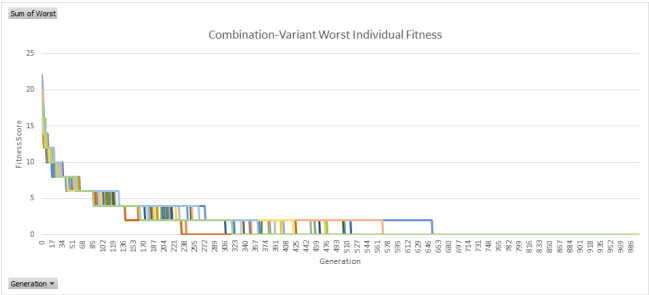

Figure 7.

4. Conclusion

While both versions of the algorithm succeeded in finding a solution to the Eight Queens Problem, the combination version succeeded in finding what looks like a larger number more quickly. In actuality, the combination version ends up finding a very small number of solutions in the beginning with reasonable speed, and then quickly changes every other individual in the population into a copy of those solutions. Despite the fact that the performance looks very good, the actual quality of the solutions doesn't match.

Overall, the algorithm works. It does what it is designed to do: find a solution to the Eight Queens Problem, and record the the way the population got there. In the future, it would be interesting to attempt to tackle trying to find several unique solutions.

5. Code

5.1. Permutation Version

```
#!/Library/Frameworks/Python.framework/Versions/3.4/bin/python3
# Joel Doumit
# CS472 - Evolutionary Algorithms
# Assignment 1 - 8 Queens - Permutation

import numpy as np
import csv

def initializeBoard():
    return np.random.permutation(8)

def initializePopulation(size):
    return [initializeBoard() for i in range(size)]

def checkMatch(pos):
    counts = (np.unique(pos, return_counts=True))[1]
    conflicts = [i * (i-1) for i in counts]
    return np.sum(conflicts)

def fitness(board):
    score = 0
    posdiag = [0, 1, 2, 3, 4, 5, 6, 7]
    negdiag = [0, -1, -2, -3, -4, -5, -6, -7]

    checkpos = [x + y for x, y in zip(board, posdiag)]
    checkneg = [x + y for x, y in zip(board, negdiag)]

    score = (checkMatch(checkpos) + checkMatch(checkneg))

    return score

def pairIndividuals(population):
    fitnessPop = []
    for i in population:
        fitnessPop.append([i, fitness(i)])
    return fitnessPop

def selectionPool(population):
    chosenIndiv = np.random.choice(100, 5, replace=False)
    selectPool = []
    for i in chosenIndiv:
        selectPool.append(population[i])
    return selectPool

def sortPool(to_sort):
    to_sort.sort(key=lambda x: x[1])
    return to_sort

def pickParents(sorted_pool):
```

```
    return sorted_pool[0][0], sorted_pool[1][0]

def crossover(parents):
    kids = [[],[]]
    crossover = np.random.choice(range(1,7))
    for kid in range(2):
        offset = 0
        for i in range(8):
            if i < crossover:
                kids[(0+kid)%2].append(parents[(0+kid)%2][i])
            else:
                while(parents[(1+kid)%2][(i+offset)%8] in kids[(0+kid)%2]):
                    offset += 1
                kids[(0+kid)%2].append(parents[(1+kid)%2][(i+offset)%8])
    return np.asarray(kids)

def mutation(kids):
    #swap two values inside the kid for permutation, pick one
    #value to swap to another random value in combination.
    swapVals = np.random.choice(7, 2, replace=False)
    kids[0][swapVals[0]], kids[0][swapVals[1]] = kids[0][swapVals[1]], kids[0][swapVals[0]]
    swapVals = np.random.choice(7, 2, replace=False)
    kids[1][swapVals[0]], kids[1][swapVals[1]] = kids[1][swapVals[1]], kids[1][swapVals[0]]
    return kids

def survivors(sortedPop):
    sortedPop = sortedPop[:len(sortedPop)-2]
    return sortedPop

def stats(sortedPop):
    best = sortedPop[0][1]
    worst = sortedPop[99][1]
    avg = 0
    for i in sortedPop:
        avg = avg + i[1]
    avg = avg / 100
    return best, worst, avg

if __name__ == "__main__":
    with open('8queens_permutation_JSD.csv', mode='w', newline = '') as outfile:
        writer = csv.writer(outfile)
        writer.writerow(['Trial','Generation','Best','Avg','Worst'])

        for t in range(30):
            initPop = initializePopulation(100)
            population = pairIndividuals(initPop)
            for g in range(1000):
                parents = pickParents(sortPool(selectionPool(population)))
                kids = (crossover(parents))
                mutation(kids)
                population.append([kids[0], fitness(kids[0])])
                population.append([kids[1], fitness(kids[1])])
                sortPool(population)
                population = (survivors(population))
                Best, Worst, Avg = stats(population)
                writer.writerow([t,g,Best,Avg,Worst])
```

5.2. Combination Version

```
#!/Library/Frameworks/Python.framework/Versions/3.4/bin/python3
# Joel Doumit
# CS472 - Evolutionary Algorithms
# Assignment 1 - 8 Queens - Combination

import numpy as np
import csv

def initializeBoard():
    return np.random.choice(8, 8)

def initializePopulation(size):
    return [initializeBoard() for i in range(size)]

def checkMatch(pos):
    counts = (np.unique(pos, return_counts=True))[1]
    conflicts = [i * (i-1) for i in counts]
    return np.sum(conflicts)

def fitness(board):
    score = 0
    posdiag = [0, 1, 2, 3, 4, 5, 6, 7]
    negdiag = [0, -1, -2, -3, -4, -5, -6, -7]

    checkpos = [x + y for x, y in zip(board, posdiag)]
    checkneg = [x + y for x, y in zip(board, negdiag)]

    score = (checkMatch(checkpos) + checkMatch(checkneg))

    return score

def pairIndividuals(population):
    fitnessPop = []
    for i in population:
        fitnessPop.append([i, fitness(i)])
    return fitnessPop

def selectionPool(population):
    chosenIndiv = np.random.choice(100, 5, replace=False)
    selectPool = []
    for i in chosenIndiv:
        selectPool.append(population[i])
    return selectPool

def sortPool(to_sort):
    to_sort.sort(key=lambda x: x[1])
    return to_sort

def pickParents(sorted_pool):
    return sorted_pool[0][0], sorted_pool[1][0]

def crossover(parents):
    kids = [[],[]]
    crossover = np.random.choice(range(1,7))
    for kid in range(2):
```