Artificial Intelligence 1
Lab 1

Name1 (student number 1) & Name2 (student number 2)
Group name

day-month-year

Theory

Exercise 1

Exercise 2

Programming

Program description

The program also implements a genetic algorithm approach for solving the problem. This takes a set of solution alternatives, and generates new generations of alternatives with breeding and mutating the previous generation until a solution is found.

Problem analysis

Program design

The genetic algorithm introduces new variables: an individual looks like the global nqueens array, but this time we need several of this kind. Therefore we introduce two arrays for storing the current and the next generation, both containing 100 individuals. We also introduced a simple int array for storing the individual fitness values for inserting them in order. This could have been done with defining a generation struct as well. We also introduced son1 and son2 variables, to use them when generating new individuals.
The algorithm works as follows: it generates two sons from the current generation, and inserts them in order of their fitness (number of conflicts) to the next generation array, until it is filled with 100 individual, which seemed an optimal number in our runs. For that an insertInOrder function was defined, that inserts the son to the correct place and shifts everything after that (a heap could be used, but not crucial in this task), and a conflict counter, which is the

same as the one used for the other solutions, but it takes an individual as an argument, instead of the global nqueens individual. Then the current array is updated to the next, the global individual is updated to the best from the next generation, and the loop starts again, until a solution is found, or the generation limit is extended. Generating the new individualsrelies on the function newIndividual. It takes the current generation, and the son arrays, because the generated individuals will be stored in them. Since the genom of an indiviual is a column number for each row, the genom could be easily cut an any point, and stuck together for the new individuals to create a valid son. This is the method we used for crossover. The parents are chosen randomly, with the ones in the beginning having a greater chance to be chosen: the index of the individual is subtracted from the population size, divided by 3, and then checked if it is greater than a random number between 0 and the population size divided by 2. This method gives a linearly decreasing probability, that is not too steeply decreasing, and does not start from a too high chance for the best individiuals. The two parents must be different individuals. We also implemented a simple mutation method: since swapping the value of a row creates a valid individual (if the new value is within the size of the table), we could use this method for more variability and to avoid local optima.

Program evaluation

The genetic algorithm however works very efficiently for this problem: even for bigger table sizes, like 20 - 40 it is able to find the solution normally in a couple hundred generations. For smaller sizes it usually find in less than 100 generations. Even for states between 40 and 65 the solution is found in a couple of thousand generations. Above that, the solution becomes really complex, and the program seems to get stuck with local optima: this could be prevented by higher mutation ratio, or more chance for the less fit individuals to breed, but that could lower the effectivness for smaller tables.

Program output

Program files

Main.c

```
1 Your code here
```

SomeFile.c

```
1 Some other code here
```