Introduction to Computational Intelligence - Assignment 1; 8-Queens problem

Albert-Jan Yzelman 0216313
anyzelma@students.math.uu.nl

This assignment consists of thinking of an evolutionairy algorithm to solve the eight queens problem. This problem involves placing 8 queens on a regular chessboard in such a way that neither queen can hit any other. An algorithm to solve this shall be discussed in four paragraphs, which will cover representation, operators, fitnessfunction and competition, respectively.

Representation

A Queen can move diagonally, horizontally and vertically. So, in advance we can exclude that more than one queen is placed on the same row or column. With this in mind, a very suitible representation of this problem is to convert it into an permutation of the array A=(1,2,3,4,5,6,7,8). One could interpret this array as follows; the number in cell A[i] is the column number of the i'th queen on the i'th row of the chessboard. Alternatively, you could ofcourse say that the number in cell A[i] is the row number of the i'th queen on the i'th column of the chessboard. For example, the array (6,3,2,7,5,1,4,8) can be represented by the following situation:

1	2	3	4	5	6	7	8	
					X			
		X						
	X							
						X		
				X				
X								
			X					<==> 63275148
							X	

(where an 'X' denotes the position of a queen)

Operators

By using only an array as a means to represent any situation within this problem, it becomes easy to think of operators for the representation. An operator could be, for example, swapping the location of two randomly chosen numbers within the array. In terms of evolutionairy algorithm, this is called a mutation. An example would be: 15423678 > 15823674. I have chosen for only one transposition; swapping any more colums would mutate the initial problem too strongly.

A second operator is a recombination. Recombination works on two different arrays and returns one array which is somehow similar to both initial arrays. Firstly I was thinking of using a cycle recombination, because that method ensures that the order of the numbers in both input arrays are kept as intacts as possible. This seemed to be a useful property for this problem. However, in practice, almost all solutions proposed by this operator scored signifantly worse than simple mutation. Eventually I came up with a variant on positional recombination. This works as follows. Firstly, we randomly select numbers from the first input array. These will be copied to the exactly the same index of the new solution:

array 1: 15**2**4**38**7**6**
array 2: 51427386
new: --2-38-6

After that, all empty spaces in the new array become occopied by the corresponding numbers from the second array. However, this could cause a number to appear more than one time in the new array, and this was not allowed. Should that happen, we keep the empty space empty.

array 1: 15243876
array 2: **51**4*27*3*8*6
new: 512-38-6

In the above example, the '2' and the '8' from array 2 could not be copied, because then they would occur multiple times in the new array. The remaining empty spaces are now filled in with the missing numbers, from left to right in an ascending order:

array 1: 15243876
array 2: 51427386
new: 512**4**38**7**6

By use of this method, the new array still retains a lot of simelarity with their parent arrays. This recombination performed somewhat better than cycle recombination, but still generally worse than mutation. This is probably because in this problem, the use of recombination changes an array stronger than recombination. Thus, an almost perfect solution may be scrambled beyond recognition by a recombination, whereas mutation always will stick close to that almost perfect solution.

Fitness-function:
A fitnessfunction is a means of rating a solution (represented by an array in this case). A fitness function is not hard to come up with in this case: the score is the sum of the number of other queens each queen can diagonally hit. Thus our algorithm should try to minimize this fitness function, and reaches a perfect solution when the score equals zero. As a matter of optimalization, we program the function to only look down when searching on diagonals. In this way, every conflict is counted only once, and the score calculation works twice as fast, while still ensuring to detect a conflict between queens, if one exists.

Competition:
I have chosen for a population consisting of ten solutions, of which the four best solutions (the ones with the lowest fitness score), live to become part of the next generation. This new generation consist of four individual mutations of these four best solutions, in addition to the best solutions themselves. To get to a total of ten solutions again, the remaining two members of the population are constructed by using the recombination operator twice on two randomly chosen pairs of the best solutions.

Remarks:
It sometimes occurs that the population converges to a local minimum, that never quite reaches zero. This algorithm will then 'hang' indefinitly. This is a problem recurrent in all local search algorithms. In this case it implicates that after playing lots of rounds, the population is very similar. By using a larger population, the chance that such a failed convergence occurs could be minimized.
However, by means of the chosen representation, the search space has become quite small. If we use a large population in a small search space, chances are that one or more of the randomly instantiated initial solutions, already is a perfect solution. It is not odd to find even in a population of only ten, a perfect initial solution.

Improvements:
The above remarks in mind, I increased the population to twenty (and also increased the surviving number of solutions to 8, with the new generation consisting of 8 permutations and 4 recombinations). After a bunch of tests, it was apparent that the algorithm indeed did not hang as often as it did. The amount of generations needed to reach a solution was an avarage twenty (maximum number of rounds was set at one hundred, as such a population probably never would reach a solution). In the implementation of this algorithm I did not implement random selection of pairs for use in recombination; this struck me as too much work at the time. Instead, I assumed the best pairs would recombinate together (the first two recombinate, the second two recombinate, et cetera). All things put together, this seems like a satisfactory algorithm for solving the eight queens problem.