Using Genetic Programming To Solve The n Queens Problem.

Jesper Goos

November 10, 2005

Jesper Goos

Artificial Intelligence
And Intelligent Systems

Datalogi
Roskilde Universitetscenter

Contents

Abstract

The focus of this assignment is on genetic programming used to solve the 'n Queens Problem'. The first chapter is about the basics of genetic programming, followed by the second chapter about the 'n Queens problem'. In chapter three, I go through the program, see chapter 5, step by step. Finally, in chapter four I elaborate on running the program.

In most cases, the program solves the problem - otherwise it runs out of generations. To be able to give some statistical information about how good the algorithm is, a test program need be written.

1 Genetic Programming

Genetic programming is part of evolutionary programming inspired by phenomena in the natural world. This kind of programming produces genetic algorithms, which are optimization techniques relying on parallels within nature. The optimization technique requires a method to measure how good the outcome is. This measure is defined as the fitness of a solution.

The initial idea of genetic programming originated in the observation of how the evolution of biological creatures derives from their constituent DNA and chromosomes. A simple analogy can be made with a mathematical problem with many parameters. Each parameter represents a chromosome in the mathematical analogy.

In nature, evolution takes place by selection, commonly known as Darwin's 'survival of the fittest'. This means that in genetic programming we need a population of individuals, which have alternating fitness. In practice, this is done by randomly creating the initial population. A selection method, which considers the fitness of the individuals, is then defined. This means that the less fit the individuals/solutions are, the less likely they are to be selected for a successive population. Hereby, we simulate natural selection by favouring the fittest individuals.

Besides selection, crossover and mutation are applied with a specified probability. Crossover occurs when two individuals mate. A randomly selected break point in the two mating chromosomes for splitting is found, and the two parts of each chromosome are then switched around. As a result, the offspring is different from the parents. If crossover does not occur, cloning is applied and the offspring is identical to their parents. In nature, mutation is vary rare, and can have from very small to very big results. The main purpose for applying mutation in a genetic algorithm is to avoid the search algorithm to get trapped in a local maximum. Furthermore, mutation actually can lead to a significant improvement in fitness.

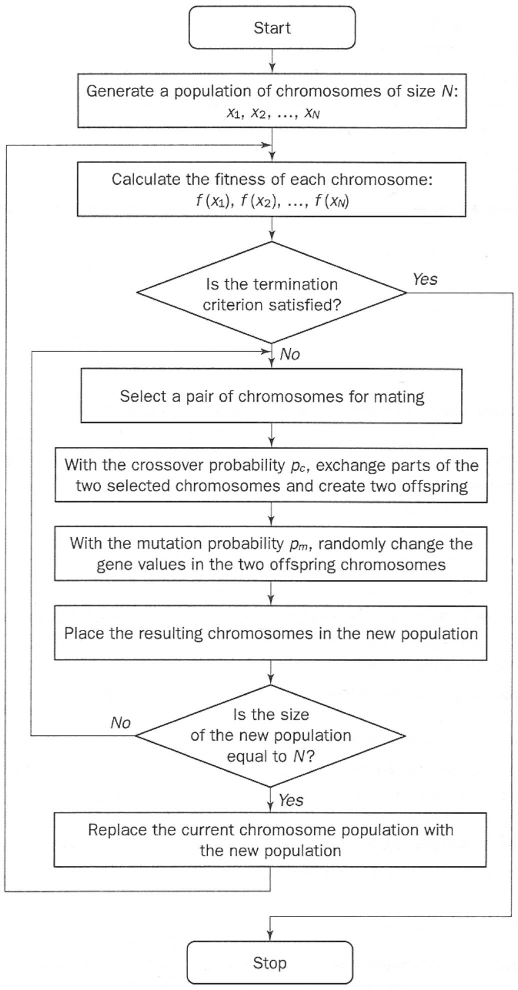
Start
Generate a population of chromosomes of size N: $x_1, x_2, \ldots, x_N$
Calculate the fitness of each chromosome: $f(x_1), f(x_2), \ldots, f(x_N)$
Is the termination criterion satisfied?
Yes
No
Select a pair of chromosomes for mating
With the crossover probability p_c, exchange parts of the two selected chromosomes and create two offspring
With the mutation probability p_m, randomly change the gene values in the two offspring chromosomes
Place the resulting chromosomes in the new population
Is the size of the new population equal to N?
No
Yes
Replace the current chromosome population with the new population
Stop

Figure 1: A path through the steps of a genetic algorithm. The illustration is taken from the courses textbook [2], p. 223.

By employing these techniques, the population is likely to improve in fitness. By furthermore allowing only a fitter generation than the last to proceed in the progress, we can improve the chances of finding the optimal solution. The genetic algorithm is hence an advanced stochastic search algorithm based on biological evolution.

As the initial population is generated randomly, the speed (number of generations) of finding a solution to a problem depends very much on the fitness of the initial population. As there are some probability factors in the algorithm, we also have to accept the fact that the search for a optimal solution may be protracted.

Figure 1 shows the path through a basic generic algorithm. The program in this assignment is structured almost completely like shown in this diagram. One of the only differences is that I do not have to control whether the population size is equal to `N` or not, because there will always be generated the same number of offspring than the previous generation has parents.

2 The n Queens Problem

This problem is a well known mathematical problem in the computer world. The example is often used to demonstrate how to use recursive backtracking. However, in this case, a genetic algorithm is programmed to show nature's way of handling these kind of problems.

By origin, this problem is known as 'The Eight Queens Problem'. The object of this problem is to place eight queens on a chessboard in such a way that no queen threatens any other queen. A chessboard consists of eight columns with each eight rows - and as it turns out, it is always only possible to place `n` queens on an `n x n` board. Therefore, it is interesting to look at this problem more generally - hence the '`n` Queens Problem'.

In a recursive algorithm one would start in either of the corners of the `n x n` chessboard. It would here probably make sense to place one queen after another in a legal position (Queens can move any number of squares in a single turn both vertically, horizontally and diagonally). Because queens can move any number of squares vertically in a single turn, such placement is not cost efficient. Rather than viewing the board as consisting of `n x n` squares, it can be seen as comprised of `n` columns, each with `n` rows. Because we are placing `n` queens in `n` columns, we can determine that there has to be one and only one queen in each column. Applying this knowledge to the recursive algorithm makes placing the queens more efficient. This is also true for the horizontal direction, as we can rotate the board. As for the last, the diagonal direction must be accounted for as well.

These reflections will be used in the implementation of the genetic algorithm explained in the following chapter.

3 The program

In this chapter, I will elaborate on the program and explain the methods. Where appropriate, I will emphasize on and discuss aspects of the code. The program is written in `PHP5`, which seems similar to `JAVA` or `C++`. The main reason for choosing this language, apart from personal interest, is that `PHP5` has a completely new object oriented part, which I wanted to test.

Because PHP earlier was not object oriented, a method is declared with the reserved word `function`. Even though it is named `function`, its usage is identical to a method in other object oriented languages - therefore, in the following, I call it a method and not a function.

Normally, in genetic algorithms a fitness function is applied in order to see if one chromosome (in this case the chessboard) is closer to the optimal solution than another. I have turned this around, so that I keep track of how many queens **are** threatened, instead of how many of them **are not** threatened. Thereby, this value can also be used to record to which degree these queens are threatened. This value is a kind of inverted fitness or cost value.

3.1 const

At the top of the class, the sevens constants `maxGenerations`, `boardSize`, `populationSize`, `crossoverPos`, `mutationPos`, `parentsChoose` & `debug` are declared.

`maxGenerations` is the maximum number of generations the application will try to run before it gives up. Without this constant, an infinite loop could occur. `boardSize` defines the height and width of the chessboard and the number of queens the application will try to place. `populationSize` represents the number of chessboards that will be used to make the next generation. `crossoverPos` is the probability between 0 & 100% that crossover will occur when two parents mate. `mutationPos` is the probability between 0 & 100% that mutation occure in a generation. `parentsChoose` and `debug` are Boolean constants for test purposes.

3.2 __construct

The `__construct` method is called every time a new object is created from the queen class. Here, the initial generation is created where the queens are randomly placed on the boards. Furthermore the four methods `getCostArr`, `getTotalCost`, `chooseParents` & `run` are called to continue the search for an optimal solution. These four methods will be explained later.

3.3 prittyPrint

This method is used for readable display of arrays.

3.4 invertCostArr

The `invertedCostArr` method has to be used from the `chooseParents` method because the `costArr` contains numbers which express a count on how many queens are threatened and to what degree they are threatened. The minimum count is 0 when no queen is threatened. The maximum count is calculated by

$$cost = (n + (n-1) + ... + (n-n)) * 2$$

where n = `boardSize`

The first part of the equation is $(n + (n-1) + ... + (n-n))$ because in the `getCost` method, we run through the hole board for each queen and compare them with all others. The other part $*2$ is added, because we run through the board vertically and diagonally for each row.

3.5 chooseParents

The `chooseParents` method is used to order the generation array according to the cost value. The goal is to let parents mate with a possibility that reflects their inverted cost. This means, the bigger the inverted cost or fitness is the more we want this board to become a parent to the next generation. Multiple selection of a parent is not a problem.

3.6 getCostArr

`getCostArr` calls the `getCost` method for every board in a generation and returns the `costArr` of a generation.

3.7 getCost

The `getCost` method calculates the cost of a board. This cost expresses how many queens are threatened on this board and to which degree.

3.8 getTotalCost

This method sums up all the values in an array. It is used to calculate the total cost of a generation.

3.9 crossOver

The `crossOver` method splits two arrays at a randomly selected column and switches the first array's right side with the second array's right side.

3.10 mutate

The `mutate` method takes a randomly selected column of the entire generation of boards and moves the queen randomly one row up or down. If the top or bottom of the board is reached, the queen is set to 0 or `boardSize`-1.

3.11 mating

The `mating` method mates all the boards in a generation with each other. The `crossOver` & `mutate` methods are applied with probability of `crossoverPos` & `mutationPos`, respectively. If crossover is not applied the boards are cloned to the next generation.

3.12 run

The `run` method chains all the other methods together and continues mating until `maxGenerations` is reached or an optimal solution is found. The `run` method only lets a generation continue to mate if the total cost is lower than the previous one. Otherwise, the generation is mated anew. An optimal solution is found when the `costArr` array contains a 0. The place of the 0 in the `costArr` array indicates which place the board with the optimal solution has in the `gen` array.

4 Running the program

The program is run by executing the queen.php file in a command line or browser. The server or local machine has to support `PHP5`.

If an optimal solution is found within the `maxGenerations`, the following message is printed:

```
Solution Found!! Time 15:54:55
Number of generations: 133
Array
(
    [0] => 0
    [1] => 3
    [2] => 1
    [3] => 4
    [4] => 2
)
```

`Number of generations` represents the number of generations used by the application to find an optimal solution.

If no such solution is found within the `maxGenerations`, the following message is printed:

```
genCount = 500
totalCost = 114

genCount = 1
totalCost = 182
```

`genCount` is the number of generations the program has run. `totalCost` is the cost of the generation. The first two lines show the last run and the last two lines show the first run or the initial values.

5 Conclusion

The program was tested by running it again and again with different values for `maxGenerations`, `boardSize`, `populationSize`, `crossoverPos`, and `mutationPos`. As one would expect, the most evident result of the test is that the bigger the boards the more generations are needed to find an optimal solution.

By trying to switch off the favouring of the fittest parents in the mating process, I tried to see if the number of generations would decrease. Unfortunately, no unequivocal conclusion could be reached in this matter.

The matter of cost vs. fitness mentioned in chapter three would be an interesting thing to analyse. It would be interesting to see whether this approach actually makes any difference. By using a normal fitness function (which counts the non threatened queens), one would, on an 8 x 8 board, have a fitness value between 8 and 0 for the best and the worst case, respectively. By using the cost value instead, I have a value between 72 and 0. This should make it easier to improve the total cost of a generation, and thereby accept this generation to continue in the mating process.

As way of testing the efficiency of the algorithm, a test program providing statistical data would be very useful.

6 The Queen Class

```
   <?php
set_time_limit(3000);
class queen{

   const maxGenerations   = 2000;
     // maximal number of genarations
   const boardSize       = 8;
     // height og width of every
   const populationSize    = 20;
     // size of the population
   const crossoverPos       = 70;
     // Crossover possibility (0-100%)
   const mutationPos       = 30;
     // Mutation possibility (0-100%)
       const parentsChoose     = false;
     // Debugging constant {true,false}
   const debug          = false;
     // Debugging constant {true,false}

   /**************************************************
   Constructor
   **************************************************/
   function __construct()
   {
     /* The first generation is initialized randomly */
      for($i = 0; $i <= self::populationSize-1; $i++)
      {
        for($j = 0; $j <= self::boardSize-1; $j++)
       {
        $gen_1[$i][$j] = rand(0,self::boardSize-1);
       }
      }
      // An array with the cost of every
            // boards is created
      $costArr    = $this->getCostArr($gen_1);
      // The total cost is calculated
      $totalCost = $this->getTotalCost($costArr);
      // The generation is sorted
            // according to cost for mating
           if(self::parentsChoose){
             $gen_1    = $this->chooseParents($gen_1,$costArr);
             }
            // The process is run until a solution is found
            // or maxGenerations is reached
            $this->run($gen_1,$totalCost,0);

       /* DEBUG */
      if(self::debug==true){$this->prittyPrint($costArr);}
      if(self::debug==true){$this->prittyPrint($gen_1);}
        /* DEBUG */
   }

   /*************************************************
   Function to run all the generations
   *************************************************/
   function run($gen,$totalCost,$genCount)
   {
         if($genCount <= self::maxGenerations)
     {
     $nextGen       = $this->mating($gen);
     $nextCostArr    = $this->getCostArr($nextGen);
       // Check if the goal is reached
       if(in_array(0,$nextCostArr)){
       print("Solution Found!! ");
       echo "Time ".date("H:i:s")."\n";
             print("Number of generations: $genCount\n");
       $key = array_search(0,$nextCostArr);
       print_r($nextGen[$key]);
       exit();
       }
     $nextTotalCost = $this->getTotalCost($nextCostArr);

       if($nextTotalCost<$totalCost){
       $genCount++;
             if(self::parentsChoose){
              $nextGen = $this->chooseParents($nextGen,$nextCostArr);
                 }
             $this->run($nextGen,$nextTotalCost,$genCount);
       }else{
       $genCount++;
       $this->run($gen,$totalCost,$genCount);
       }
       if($genCount==1 ||
                    $genCount==self::maxGenerations){
           echo "genCount = $genCount\n";
           echo "totalCost = $totalCost\n\n";
       }
     }

   }

   /**************************************************
   The hole generation is mated
   **************************************************/
   function mating($parrents){
     // Mating is totaly random -
         // the parents array is shuffled
     // shuffle($parrents);
     for($i = 0; $i <= self::populationSize-1; $i++){
       // crossover - parents are splittet
             // and reassembled
       if(self::crossoverPos<=rand(0,100)){
       $children =
       $this->crossover
             ($parrents[$i],$parrents[$i+1],$children,$i);
       }else{
       // cloning - parents go to the next generation
       $children[$i]=$parrents[$i];
       $children[$i+1]=$parrents[$i+1];
       }
       $i++;
     }
     // mutation
     if(self::mutationPos<=rand(0,100)){
       $children = $this->mutate($children);
     }
     return $children;
   }

   /**************************************************
   One row is altered in this function
```

```
**************************************************/
function mutate($children)
{
  $randBoard   = rand(0,self::populationSize-1);
  $randRow     = rand(0,self::boardSize-1);
  $randUpDown = rand(0,1);

  if($randUpDown==0){
  $children[$randBoard][$randRow]=
                $children[$randBoard][$randRow]+1;
    if($children[$randBoard][$randRow]==
             self::boardSize){
      $children[$randBoard][$randRow]=0;
    }
  }else{
  $children[$randBoard][$randRow]=
                $children[$randBoard][$randRow]-1;
    if($children[$randBoard][$randRow]==-1){
      $children[$randBoard][$randRow]=
                self::boardSize-1;
    }
  }
  return $children;
}

/**************************************************
Crossover is calculated
**************************************************/
function crossOver($parrent1,$parrent2,$children,$i)
{
  $crossoverCut = rand(0,self::boardSize-1);
  for($c = 0; $c <= $crossoverCut; $c++){
    $child1[$c]=$parrent1[$c];
    $child2[$c]=$parrent2[$c];
  }
  for($c = $crossoverCut;$c <= self::boardSize-1; $c++){
    $child1[$c]=$parrent2[$c];
    $child2[$c]=$parrent1[$c];
  }
  $children[$i]=$child1;
  $children[$i+1]=$child2;
  return $children;
}

/*************************************************
Generation af array with cost values
*************************************************/
function getCostArr($arr)
{
  foreach($arr as $key => $value){
  $costArr[] = $this->getCost($value);
  }
  return $costArr;
}

/*************************************************
Calculates the total cost af th costArr
*************************************************/
function getTotalCost($costArr)
{
  return array_sum($costArr);
}

/*************************************************
Cost is calculated as number of threats for a board
Everey threat is counted...
*************************************************/
function getCost($board)
{
  $costValue = 0;
  for($k = 0; $k <= self::boardSize-1; $k++){
    $r = $board[$k];
    for($j = 0; $j <= self::boardSize-1; $j++){
      if($j<>$k && $board[$j]==$r){
        $costValue++;
        }
    }
    for($j = 0; $j <= self::boardSize-1; $j++){
      if($j<>$k &&
              (abs($board[$k]-$board[$j])==abs($k-$j))){
        $costValue++;
        }
    }

  }
  return $costValue;
}

/**************************************************
The parrrenst for the next generation are determined
**************************************************/
function chooseParents($gen,$costArr)
{
    // invertering af cossArr, s de
    // bedste forldre favoriseres
    $costArr = $this->invertCostArr($costArr);
    // totalCost of the inverted costArr
    $totalCost = $this->getTotalCost($costArr);
    foreach($costArr as $key => $cost)
    {
      $tmpCost = 0;
      $i = 0;
      $costRand = rand(0,$totalCost);
      while($tmpCost<=$costRand)
      {
        $tmpCost = $costArr[$i] + $tmpCost;
        $i++;
      }
      $newGen[$key]=$gen[$i-1];
    }
  return $newGen;
}

/**************************************************
The costArr is inverted, so that the biggest numbers
represent the best parrents
**************************************************/
  function invertCostArr($costArr)
  {
    for($i = 0; $i <= self::boardSize-1; $i++)
    {
      $worstCost=$worstCost+(self::boardSize-$i);
    }
```

```
        $worstCost = $worstCost*2;
        //Can also be calculated as ()n * (2 + (n-1)))
        foreach($costArr as $key => $cost)
        {
          $costArr[$key]=$worstCost-$cost;
        }
        return $costArr;
      }

    /**************************************************
    Print of arrays for debugging
    **************************************************/
    function prittyPrint($arr)
    {
      echo "<pre>";
      print_r($arr);
      echo "<pre>";
    }

}

$run = new queen;

?>
```

References

[1] H. Hirsh: *Geneticprogramming*, `http://www.cs.nott.ac.uk/~gxk/courses/g5baim/papers/gp-001.pdf`, (2000)

[2] M. Negnevitsky: *Artificial Intelligence - A Guide to Intelligent Systems*, Addison Wesley, Second Edition, (2005)