Solving TSP with Help of a Genetic Algorithm in Java

This article is about solving the famous **travelling salesman problem** (TSP) with help of a **genetic algorithm** (GA), implemented with Java.

Before digging into the code, let's briefly explain this appalling problem.

The Problem

Imagine you defined some locations on a map. Now, you want to travel through all of them within one journey while trying to keep the overall distance at a minimum. Which order would you choose? This may sound like an easy question at first glance. It actually is when talking about three, five or six cities. But what if we talk about 50, 100 or even 10'000 cities? Then you realize how hard this problem is.

You might think, we have computers, so let's let them do the work. Unfortunately, it's not that easy because the TSP belongs to a special kind of problem having a so-called non-deterministic polynomial-time hardness.

Here is why:

Cities	Possible Solutions
5	5! = 120
8	8! = 40'320
15	15! = 1'307'674'368'000
45	45! = 119'622'220'865'480'194'561'963'161'495'657'715'064'383'733'760'000'000'000
60	60! = 83'209'871'127'413'901'442'763'411'832'233'643'807'541'726'063'612'45'952'449' 277'696'409'600'000'000'000'000

As you can see, the number of possible solutions increases quickly in unimaginable spheres. These are such high numbers that computers would need multiple decades to check all variations just for determining the optimal solution.

In contrast to conventional problems, the best solution is not known in advance. Think of a Sudoku: it may demand a lot of intellectual power and time, but once completed, you are easily able to check if the (only) solution is given. That's not the case in the TSP. You will find a more comprehensive explanation of such NP-problems on Wolfram. Additionally, I recommend this video – it may lower the level of weirdness for you (or increase it…).

A Solution

Solving extraordinary problems demands extraordinary approaches. One is a so-called Genetic Algorithm (GA). Its logic is based on the natural selection seen in evolution. A key aspect is «the survival of the fittest», as Darwin once stated. Before looking at an implementing of a GA, I kindly invite you to read a bit of theory regarding the algorithm's structure and behavior.

Components

Element

An element is an exact definition of how to solve the problem. Referring to the TSP, this would be a specific path or route, going through all the cities. This is a possible solution, but it is not necessarily the best one.

Gene

A gene represents the smallest entity in the whole GA. Knowing that an element defines a concrete solution, a gene is one single component of it. Adapted to the TSP, one city or stop within an element is considered to be a gene.

Population

A set of elements forms a population. The number of elements instantiated defines the population's size. When creating a new population, all elements should be constructed completely at random. Subsequently, the GA tries to evolve its population further in order to develop better elements.

Generation

A current state of a population can be seen as a generation. When evolving the population further, we generate new generations. In other words, a generation is a specific set of elements held in the population.

Fitness

According to Darwin, only the fittest will survive. To evaluate which elements are more likely to survive, we have to assess the quality of each element's solution in relation to all other solutions. In doing this, we are able to indicate which solutions are better and which are worse, and we can award each of them with an appropriate fitness value. In TSP, the paths with the smallest total distance must be rewarded with the highest fitness. In other problems, this fitness function may differ.

Approach

Selection

Since we try to stimulate our population to evolve a more sophisticated and better version of itself, we should wisely select the elements which may survive a generation. Obviously, we want the genes of those elements with a higher fitness to survive while those with a poor fitness to cease.

Crossover

With just determining the fitness and selecting the fitter elements, we did not optimize our solution to the TSP yet. Therefore, we will apply crossover, like its done in nature during the mating season. This enables to bequeath the qualitative good genomes of two pretty fit elements (depends on the selection) to a new one, which eventually will be part of the population's next generation.

As an element refers to a specific path in TSP, we extract a random sequence of a fit element's path (parent I) and put it into a new element (child). Then, we fill the remaining slots with path fragments (genes) of another fit element (parent 2). Doing this, we hopefully increase the possibility of having a child who is fitter than his parents.

Mutation

Like in the real world, the evolution is influenced by some mutated genomes. By pairing parent AAAA with parent BBBB, we will end up with any kind of A-B-combination, but we will never get a C part in it. Hence, it is advisable to enforce some mutation in a GA.

Once we bred our child element in the TSP, we simply switch the order of the vertices (the genes), representing a concrete path, only very little but randomly.

Evolution

The last and most important part of a GA is the act of repetitively creating new generations (evolving the population), as this is the only way to boost evolution in an advantageous direction. In doing this, we hopefully will replace the randomly constructed elements in our initial population with fitter ones over time. However, there is a good chance that our initial population will get stuck in an insufficient level of quality. In such a case, we only can hope for a lucky mutation to receive a significant fitter child element, but then we solely rely on randomness, which is not a good approach in such an enormous problem. To tune our GA, it is good practice to discontinue the evolution process of a population after a certain amount of generated generations, for instance, 100 in number, and start with a completely new population. Since the new population has to initiate its elements, we re-introduce a great amount of randomness, which hopefully yields a better solution when evolving this population again 100 times.

The (more or less) pseudo code below will support your understanding of the overall process.

```
//take a guess
bestPath = random path;

while(!stop) {

   //create a new population
   Population pop = new Population;

   //evolve population a limited number of times
   for (int i=0 ; i<100 ; i++){

      //create the next generation
      pop.evolve;

     //retrieve the generation's best solution
     Path p = pop.getFittestPath;

     //compare it with the overall best solution
     if(p.distance < bestPath.distance){

        //we've found a better solution
        bestPath = p;
        GUI.repaint(bestPath);
      }
   }
   /*despite we may have developed an outstanding population,
    *we stop evolving it here and start from scratch again   */
}
```

Parameters

Since a GA is a heuristic approach to solve NP problems, the solution’s quality, as well as the time frame needed to reach a good solution, strongly depends on the custom configuration of some constants (parameters).

For instance, choosing a high population size increased the possibility of finding a fitter element but it also raised the computational work. In a TSP with 5 vertices, a population size of 200 would be nonsensical as there are only 5! = 120 possible solutions.

Another important constant is the mutation rate. The level of randomness influencing the GA’s effectiveness is directly linked to it. The more mutation we put in, the fewer genomes will survive a generation identically. It is advisable to keep the rate low since crossing over fit parents to birth an even fitter child would become superfluous and the whole process would resemble more of an inefficient brute force attempt.

To avoid the problem of getting stuck with an unfavorable population, the evolving process should be limited to a maximum number of generations. Once reached, the GA simply initializes a complete new population. In doing this, all elements are defined with a big amount of randomness. Therewith, we do not gamble away the chance of discovering entirely different solutions. Based on experience, a value between 50 and 200 is a good limitation for the number generations evolved.

Code Extracts

In this chapter, I would like to explain some significant code extracts of my TSP implementation. In order to be more specific, I name from now on a population's element as a path. Such a path object in my solution only holds integer values in an array, which corresponds to an index in the set of vertices (which is an ArrayList). Additionally, I omitted the requirement of calculating a closed path (Hamiltonian path instead of Hamiltonian circle). Please be aware, there are numerous other ways to implement these extracts in Java, but here comes mine.

Shuffling

When initiating a population, I would like to randomize the content of all paths in my population.

```
Path p = new Path(number of vertices to travel)
p.shuffle(x)
```

When constructing a new path, the order of how to travel the vertices equals the order by which the vertices were put into the vertex set by default. To rearrange this order, I am obliged to shuffle it. The *x* implies the intensity of the shuffling process, concretely the number of how many swaps being executed. When initiating the population, I demand the paths be mingled intensively to ensure a great amount of randomness. In doing so, this method will perform 10 swaps on a path for 10 vertices.

```
private void shuffle(int nIntensity) {

   int nMax = this.getLenght();
   int nIndexA = -1; //arbitrary
   int nIndexB = -1; //arbitrary

   boolean lExit = false;

   if(nMax == 1) {
      //no need to swap
      lExit = true;
   }

   if(nMax == 2) {
      //swap once
      this.swap(0, 1);
      lExit = true;
   }

   if (!lExit) {
      //swap two elements randomly a certain number of times
      for ( int i = 0 ; i <= nIntensity ; i++ ) {
         //set indices equal
         nIndexA = nIndexB;

         //find two different indices
         while(nIndexA == nIndexB) {
            //choose two indices randomly
            nIndexA = (int)(Math.floor(nMax * Math.random()));
            nIndexB = (int)(Math.floor(nMax * Math.random()));
         }

      //swap those two
      this.swap(nIndexA, nIndexB);
      }
   }
}
```

Ultimately, making two vertices changing its place can simply be achieved by swapping them.

```
private void swap(int a, int b) {
   int temp = this.Order[a];
   this.Order[a] = this.Order[b];
   this.Order[b] = temp;
}
```

I also make use of the shuffling function when it comes to mutation. In this case, the intensity of mingling is linked to the mutation rate which is a constant provided by the user.

```
public void mutate(double nMutationRate) {

   int nIntensity = 0;

   /*when mutation rate is 0.4, only 40% of the path
    *should be changed   */
   int len = this.getLength();
   double a = Math.floor(len * Math.abs(nMutationRate));
   nIntensity = (int)(a);

   /*since the mutation will be simply execute by swapping
    *the elements we have to divide the intensity by
    *two as well (one swap ~~> two changes)   */
    nIntensity = nIntensity / 2;
    this.shuffle(nIntensity);
}
```

Picking

Probably the most important part in the GA is the selection procedure. In order to make the next generation of a population more sophisticated, we have to pick paths wisely before crossing them. One implementation to do the selection could be the following.

```java
//the aim is, to pick a path with a better fitness
//more likely than a path with a smaller fitness
private static Path pickOne(ArrayList<Path> list) {

   Path aReturn =null;
   int index = 0; //assumption
   double r = Math.random();

   /**
    * imagine we would have two elements in the list.
    * The first element has a fitness of 0.9, the
    * second one of 0.1. Consider the fact, that the
    * fitness indicates (rather obvious) how well an
    * element fits. Hence we misuse this value as the
    * probability of being picked. The random method
    * returns a value between 0.0 and 1.0, which gets
    * stored in the variable 'r'. Now, the probability
    * that we will oversee the first well fitting
    * element (e.g.: fitness of 0.9) resides by 10%,
    * since in this case 'r' must be between 0.91 and 1.0.
    */
   while (r > 0) {
      r = r - list.get(index).getFitness();
      index++;
   }

   //take previous one (the one which caused the loop exit)
   index--;

   //get the path
   aReturn = list.get(index);
   return aReturn;
}
```

When dealing with a large population size, I advise against using this method. Since the sum of all fitness values in the population equals 1.0 (proportional allotment), every single value would be extremely small. Consequently, the break-out of the while-loop will rather be based on arbitrariness than on a wise selection. Therefore, I implemented another method, which is admittedly less spectacular than the first one, but I experienced it as more expedient.

```java
private static Path pickRank(ArrayList<Path> list, int nIndex) {

   Path aReturn =null;
   ArrayList<Path> clonedList = new ArrayList<Path>(list);

   //sort paths according to their fitness in descending order (using lambda)
   clonedList.sort( (p1, p2) -> Double.compare(p2.getFitness(), p1.getFitness()) );

   //get the required path
   aReturn = clonedList.get(nIndex);

   return aReturn;
}
```

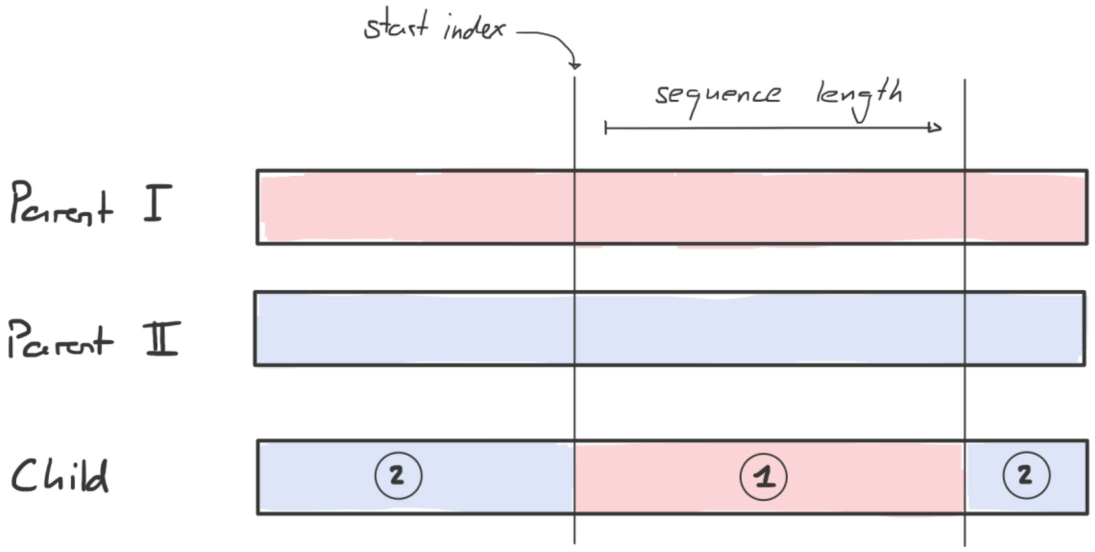
After the provided list of paths was sorted by its fitness in ascending order, this method was able to pick a path by its rank. In first place is the fittest path, in second place the second-fittest path and so on. Therewith, I have the opportunity to pick the two fittest paths in the population and create all the child paths for the next generation based on them.

Crossing

Once two paths have been picked, I cross them over. The number of genes originating from the first path, respectively from the second path, is defined randomly. Alternatively, one could also set the shares to 50%.

```
private static Path cross(Path p1, Path p2) {

   Path aReturn = null;

   int nLength = p1.getLength();
   int nSequenceLenght = (int) (nLength * Math.random());

   //set minimum length
   if(nSequenceLenght == 0) {
      nSequenceLenght = 2;
   }

   //decrease length
   if(nSequenceLenght == nLength) {
      nSequenceLenght -= 2;
   }

   //define start index randomly
   int nStartIndex = (int) ((nLength-nSequenceLenght) * Math.random());

   //initiate a new order (child path)
   int[] order = new int[nLength];

   //put -1 into each slot as a placeholder
   for (int i=0 ; i<nLength ; i++) {
      order[i] = -1;
   }

   //fill in genome of first parent path
   for(int i=nStartIndex ; nSequenceLenght>0 ; i++) {

      order[i] = p1.get(i);
      nSequenceLenght--;
   }

   //fill in genome of second parent path
   int n=0;
   for (int i=0 ; i<nLength ; i++) {
      if (order[i] == -1) {

         //fillable slot found
         boolean lExit=false;
         while(!lExit) {

            //check if vertex-index already included
            if (Path.contains(order, p2.get(n))) {
               n++;
            } else {
               lExit = true;
            }

         }
         order[i] = p2.get(n);

      }
   }

   aReturn = new Path(order);
   return aReturn;
}
```

Since both the start index and the sequence length are arbitrarily defined, my child path could be composited in various ways. The picture below visualizes one of them.



Evolving

After exploring the components of a GA, the cooperation of them, using the PickOne method, would look like this.

```java
public void evolve() {

   ArrayList<Path> ps_new = new ArrayList<Path>();
   this.assessFitness();

   for (int i=0 ; i<this.ps.size() ; i++ ); {

      Path p1 = pickOne(this.ps);
      Path p2 = pickOne(this.ps);
      Path p3 = cross(p1, p2);
      p3.mutate(this.nMutationRate);
      ps_new.add(p3);

   }
   this.ps = ps_new;
}
```

To make usage of the (more target-aimed) method PickRank, I created the following alternative.

```java
public void evolve2() {

   ArrayList<Path> ps_new = new ArrayList<Path>();
   this.assessFitness();

   Path p1 = pickRank(this.ps, 0);
   Path p2 = pickRank(this.ps, 1);

   while(ps_new.size() < this.ps.size()) {

      Path p3 = cross(p1, p2);
      p3.mutate(this.nMutationRate);
      ps_new.add(p3);

   }
   this.ps = ps_new;
}
```

Conclusion

The genetic algorithm approach is just one way to tame the traveling salesman problem. From my point of view, its parallels to the evolutionary process in the real world make it comprehensible and fun to implement. For sure, there are other interesting problem-solving approaches like the so-called simulated annealing or ant colony optimization algorithms.

In case you would like to dig deeper in this fascinating TSP, I recommend exploring this page. It provides historical information, other solving concepts, solution quality assessments and more.