Introduction to Artificial Intelligence

Final Project

Genetic Algorithm for the Traveling Salesman Problem

Asher Saban
Liat Waltuch

15.4.2012
Table of content

1. Introduction ........................................................................................................... 3
2. The Travelling Salesman Problem ....................................................................... 3
3. Approximations for TSP ....................................................................................... 5
4. Genetic algorithms .................................................................................................. 8
   4.1. Overview ......................................................................................................... 8
   4.2. Pseudo-Code .................................................................................................. 8
   4.3. Main components of GA ............................................................................... 9
5. Implementation Notes ............................................................................................ 16
6. Complexity Analysis .............................................................................................. 17
7. Experimental Tests ............................................................................................... 19
8. Avoiding Local Optima ......................................................................................... 24
9. Final Conclusions ................................................................................................. 28
10. Operating instructions ......................................................................................... 29
11. Bibliography ......................................................................................................... 30
**Introduction**

In this paper we attempt to deeply investigate the use of a genetic algorithm for solving a known computational problem, The Travelling Salesman Problem. We will present some variations of the main components of the genetic algorithm – selection, crossover and mutation. We will discuss the differences between them and conclude the optimal combination that forms a robust genetic algorithm. Finally, we will address the problem of avoiding local optima, which is a known disadvantage of local search algorithms.

Our discussion will be based on experimental tests that will combine all main components of the GA. Results will be compared to approximation algorithms and to best known solutions which we found on the web.

**The Travelling Salesman Problem (TSP)**

The TSP is a well known computational problem in combinatorial optimization, which is known to be NP-Complete. Here is an informal definition of the problem:

- N points (cities), as well as the cost of travelling between every pair of them are given.
- Assume that a salesperson, starting from a given city, has to visit each city exactly once and hence make a round trip. The aim is to find an optimal tour in which the total cost of the round trip is minimized.

More formally, the TSP can be formulated as a problem in graph theory.

- Given a complete graph $G$ on a set of $V$ vertices (cities) and a set of $E$ edges, and the cost for each edge $e \in E$, find a Hamiltonian Cycle of minimum total cost.

To complete the definition we shall define the following terms:

Complete graph - a simple undirected graph in which every pair of distinct vertices is connected by a unique edge.
Hamiltonian Cycle – given a graph $G$ on a set of $V$ vertices, a closed sequence of edges in $G$ (i.e. a cycle) which passes through each vertex of $G$ exactly once is called a Hamiltonian Cycle.

In our TSP instances we will use the Manhattan Distance between two vertices as the cost of their edge. That is the cost function is the distance between two nodes and is given by:

$$c(x, y) = \text{dist}((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

From the computational point of view, any solution for TSP would have to determine a non-repeating sequence of 1,2,...,$V$ where the cities are numbered consecutively from 1 to $V$ and the weight sum of the permutation represents the visiting order is minimized.

The search space contains the set of all possible permutations for the sequence 1,2,...,$V$. In fact, there are $\frac{(n-1)!}{2}$ possible solutions. Since TSP is an NP-Complete problem, the best known algorithms that outputs optimal solution have exponential (deterministic) run time complexity.

To following table demonstrates it:

<table>
<thead>
<tr>
<th>No. cities</th>
<th>No. tours</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>12</td>
<td>12 microsecs</td>
</tr>
<tr>
<td>8</td>
<td>2520</td>
<td>2.5 milliseconds</td>
</tr>
<tr>
<td>10</td>
<td>181,440</td>
<td>0.18 seconds</td>
</tr>
<tr>
<td>12</td>
<td>19,958,400</td>
<td>20 seconds</td>
</tr>
<tr>
<td>15</td>
<td>87,178,291,200</td>
<td>12.1 hours</td>
</tr>
<tr>
<td>18</td>
<td>177,843,714,048,000</td>
<td>5.64 years</td>
</tr>
<tr>
<td>20</td>
<td>60,822,550,204,416,000</td>
<td>1927 years</td>
</tr>
</tbody>
</table>

The proof for the NP completeness of TSP is discarded in this paper and can be found in:

Approximations for TSP

Various heuristics and approximation algorithms, which quickly yield good solutions have been devised. Among them one can find Nearest Neighbor Algorithm, Christofides Algorithm (1.5 approximation) and the Minimum Spanning Tree (MST) heuristic (2-approximation). For evaluating the results of our GA we have implemented the MST 2-approximation.

Following is a short overview and example of the algorithm:

2-Approximation for TSP with minimum spanning tree heuristic

First we will state that, if $P \neq NP$ (which haven’t been proven yet), then there is no such constant $c$ such that the TSP has a $c$-approximation that can be found in polynomial time (proof is beyond paper’s scoop). So we will limit our discussion to the case which the TSP complies with the triangle inequality:

$$\text{for each } x, y, z \in V: \quad c(x, y) + c(y, z) \geq c(x, z)$$

Note that the cost function we will use for the TSP instances (Manhattan Distance) compiles with the triangle inequality.

Pseudo-code APPROX-MST-TSP

Input: a complete undirected graph $G = (V, E)$ and the costs of each edge $e \in E$
Output: a non-repeating sequence of vertices, which represents a solution for the TSP.

1. Choose an arbitrary root in G. Let $\alpha$ be a root in the following figure.

![Diagram of a graph with vertices labeled a, b, c, d, e, f, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, and edges connecting them.](attachment:graph.png)
2. Construct MST from the root using Kruskal’s algorithm.

3. Traverse the graph with Depth-First Search and list the visited vertices.

   \[ L = \{a, b, c, h, d, e, f, g\} \]

4. Return the Hamiltonian Cycle.

We will prove that the solution length is guaranteed to be less than twice the length of the optimal solution, hence 2-approximation for TSP.

**Theorem:** APPROX-MST-TSP is a 2-approximation algorithm for TSP with triangle inequality

**Proof:**

1. Let \(H^*\) denote the optimal tour. Observe that a TSP solution with one edge removed is a spanning tree (not necessarily MST). It implies that the weight of the MST T is a lower bound on the cost of an optimal tour:
2. A full walk, \( W \), traverse every edge of MST, \( T \), exactly twice. That is,

\[
c(W) = 2c(T) \Rightarrow c(W) \leq 2c(H^*)
\]

3. For each vertex we already visited we make a “shortcut” to the next vertex we haven’t visited. The triangle inequality guarantees that the shortcut we take does not increase the cost of the tour.

4. Hence, \( c(W) \leq 2c(H^*) \)
Genetic algorithms

Overview

The GA is an optimization technique based on natural evolution that is the change over a long period of time. In nature, the fittest individuals are most likely to survive and mate, therefore the next generation’s individuals should be fitter and healthier because they were bred from healthy parents. The same idea is applied to a problem by first ‘guessing’ solutions and then combining the fittest solutions to create a new generation of solutions which should be better than the previous generation. That is, one of the main concepts of a GA is improving existing solutions.

The algorithm starts by randomly generating a set of solutions (a population). Then two or more parents are selected to form new solutions (offspring). Selection is done according to the solution’s fitness - the more suitable it is the more chances it has to reproduce. Next, new solutions are generated from the selected parents by a crossover operator. Finally, the new generated solutions undergo a mutation with a certain probability. This process is repeated until some condition (for example number of generations that passed or improvement of the best solution) is satisfied.

Pseudo-code for GA

1. [Start] Generate random population of n individuals (suitable solutions for the problem)
2. [Fitness] Evaluate the fitness f(x) of each individuals x in the population
3. [New population] Create a new population by repeating following steps until the new population is complete.
   3.1. [Selection] Select two parent individuals from the population.
   3.2. [Crossover] With a crossover operator cross over the parents to form a new offspring (children).
3.3. [Mutation] With a mutation probability, mutate the new offspring.

3.4. [Accepting] Place new offspring in a new population

4. [Replace] Use new generated population for a further run of algorithm

5. [Test] If the end condition is satisfied, stop, and return the best solution in current population

6. [Loop] Go to step 2

**Components of Genetic algorithm**

In this section, we break down the main components of the GA and discuss them more thoroughly.

1. **Encoding**
   Before a genetic algorithm can be put to work on any problem, it is needed to encode potential solutions to that problem in a form in which a computer can process.
   The encoding we used for a TSP solution:
   Given $N$ cities, each city was given a unique integer in the range of 0,1, ..., $N - 1$. Thus a possible solution is a permutation of the set {0,1, ..., $N - 1$}, where the order of the cities is specified from left to right.
   For example: 5 cities {0,1,2,3,4}. The encoding 31240 encodes the following tour:
   
   $city_3 \rightarrow city_1 \rightarrow city_2 \rightarrow city_4 \rightarrow city_0 \rightarrow city_3$

2. **Fitness function**
   The goal of a fitness function is to provide a meaningful, measurable, and comparable value given a set of individuals (i.e. set of solutions). The fitness functions allows us to determine which individuals will be selected for recombination and mutation at each phase of the GA, where the basic rule is – higher fitness value denotes a better individual. The fitness function used in our case:
   Given a tour $c_1, ..., c_N$, the length of a tour is defined to be:
\[ l = \text{dist}(c_N, c_1) + \sum_{i=1}^{N-1} \text{dist}(c_i, c_{i+1}) \]

Where \(\text{dist}(c_i, c_{i+1})\) is the Manhattan distance between city \(i\) and city \(i + 1\).

In order to describe shorter tours with higher fitness we chose the fitness function:

\[ \text{fitness} = \frac{1}{l} \]

3. **Initial population**

Once the genetic representation and the fitness function are defined, a GA proceeds to initialize a population of solutions and then to improve it through repetitive application of selection and recombination operators.

Initially many individual solutions are randomly generated to form an initial population. The population size depends on the nature of the problem, but typically contains several hundreds or thousands of possible solutions. Traditionally, the population is generated randomly, allowing the entire range of possible solutions (the search space). In our case we generated initial population in the size of 100-1000 random individuals.

4. **Selection**

During each successive generation, a proportion of the existing population is selected to breed a new generation. Individual solutions are selected through a stochastic fitness-based process, where the requirement is that fitter solutions (as measured by a fitness function) are typically more likely to be selected.

We have implemented two Selection operators:

**A. Roulette Wheel Selection**

Parents are selected according to their fitness. The better the individuals are, the more chances to be selected they have. Imagine a roulette wheel in which all individuals in the population are placed in it, each one has its place big enough
according to its fitness function. Then a marble is thrown there and selects the Individual. The chance to be selected increases as the share of the individual on the wheel, which is proportional to its fitness increases.

\[
P(X = i) = \frac{\text{fitness}(i)}{\sum_{j=0}^{N} \text{fitness}(j)}
\]

Pseudo-code: ROULETTE-WHEEL-SELECTION

1. Repeat until num of desired individuals are selected:
   1.1. \( r \leftarrow \text{random number in the range } (0,1) \)
   1.2. \( \text{sumOfProb} \leftarrow 0 \)
   1.3. Iterate over all individuals \( i \) in the population:
      1.3.1. \( \text{if } r \text{ is in the range of } (\text{sumOfProb, sumOfProb + P(X = i)}) \) then individual \( i \) has been selected.
         1.3.1.1. Go back to (2).
      1.3.2. \( \text{sumOfProb} \leftarrow \text{sumOfProb + P(X = i)} \)
B. **Tournament selection**

Tournament selection involves running several "tournaments" among a few individuals chosen at random from the population. The winner of each tournament (the one with the best fitness) is selected for crossover. *Tournament* pressure is defined to be the tournament size. As it grows larger, weak individuals have a smaller chance to be selected.

**Pseudo-code: TOURNAMENT-SELECTION**

2. Repeat until num of desired individuals are selected:
   2.1. Randomly choose tournamentSize individuals from the population
   2.2. Select the best individual among them.

In our implementation we used a tournament pressure of 5%, i.e. tournament size equals to 5% of the population.

5. **Crossover**

Cross over is a process of taking more than one parent solutions and producing a child solution from them. Many common crossover techniques are based on exchanging portions of encoding between two individuals. In the TSP it cannot be used since it would in some cases generate an invalid offspring. For example:

Parent 1:

\[
\begin{array}{c}
0 1 2 3 4 5 6 7 \\
\end{array}
\]

Parent 2:

\[
\begin{array}{c}
7 1 3 0 4 2 6 5 \\
\end{array}
\]

Exchanging the highlighted portions of encoding would result in an invalid offspring since 0 or 2 would appear or not appear at all. This off course is an invalid solution for the TSP.
So there is a need for a more sophisticated crossover type. We have implemented 3 different types of crossover:

A. **Cyclic Crossover (CX)**

If our parents are:

Parent1 = 12345678  
Parent2 = 85213647

We pick the first gene from parent 1, i.e. 1 and place it in the offspring:

Offspring = 1******

We must pick every element from one of the parents and place it in the position it was previously in. Since the first position is occupied by 1, the number 8 from parent2 cannot go there. So we must now pick the 8 from parent1:

Offspring = 1*****8

This forces us to put the 7 in position 7 and 4 in position 4, as in parent1.

Offspring = 1**4**78

Since the same set of position is occupied by 1,4,7,8 in parent1 and parent2, we finish by filling in the blank positions with the elements of those positions in parent2. Thus

Offspring = 15243678

Note that in CX it is possible to end up with the offspring being the same as one of the parents. For instance if parent2 would also start with 1 the cycle would end at the first iteration and we would end up with parent2 as an offspring. We will elaborate on this observation in our discussion on avoiding local optima.

B. **Partially Matched Crossover (PMX)**

Given two parents \( s \) and \( t \), PMX randomly picks two crossover points. The child is then constructed in the following way: Starting with a copy of \( s \), the positions between the crossover points are, one by one, set to the values of \( t \) in these positions. To keep the string a valid chromosome the cities in these positions are
not just overwritten. To set position p to city c, the city in position p and city c swap positions. Below is an example, where the crossover points that were selected are 0 and 2, and the parents are 5713642 and 4627315:

First offspring: 4623751

Second offspring: 5716325

C. Order Crossover (OX)

To apply OX, two random cross points are selected. Genotype from parent1 that fall between the two cross points are copied into the same positions of the offspring. The remaining genotype order is determined by parent2. Nonduplicative genotype is copied from parent2 to the offspring, beginning at the position following the second cross point. Each genotype in parent2 that doesn’t appear already in the offspring is copied to it in the same order it appears in parent2. An Example of OX is given below with two random cross points; 3 and 6, the genotype between the crossing positions from parent1 (GHB) is copied into the same positions of the offspring. The genotype after second cross point in parent2 (BA), B is skipped since it already exists in child; therefore only A is copied to child at position 7. Traversing parent2 circularly, the genotype (HDE), skip H, D which is
copied to child at position 8. EF are copied to child at positions 1 and 2. Finally, skip G to C and copy it to child at position 3.

\[
\begin{align*}
\text{Parent}_1 &= ABC | GHB | DE \\
\text{Parent}_2 &= HDE | FGC | BA \\
\text{Child} &= EFC | GHB | AD
\end{align*}
\]

6. **Mutation**

Mutation is an operator used to maintain genetic diversity from one generation of a population to the next. Mutation alters one or more gene values in an individual from its initial state. In mutation, the solution may change entirely from the previous solution. Hence GA can come to better solution by using mutation. Mutation occurs in a user-definable probability (in our case 3%). This probability should be set low. If it is set to high, the search will turn into a primitive random search.

We have implemented three types of mutations:

A. **Random swap.** Choose 2 random points in the encoding and swap between them.

B. **Adjacent swap.** Choose 1 random point in the encoding and swap it with the gene to its right.

C. **Inverted exchange.** Select 2 random points in the encoding and invert the order of the genes. For example:

Parent = 12345678.

If sub tour 345 is selected at random using 2 points then the result would be:

Parent = 12|345|678 → 12|543|678
Implementation Notes

In this section we will give a short description about the implementation that is included in this paper. According to main classes:

1. Genetic_algorithm.GeneticProblem<E>
   Abstract class. A framework for a problem to be solved by a genetic algorithm. E stands for “Encoding”. In our implementation the encoding is integer based.
2. Genetic_algorithm.GeneticAlgorithm<E>
   Receives a GeneticProblem and solves it with genetic algorithm.
3. Genetic_algorithm.Crossover
   Genetic_algorithm.Selection
   Genetic_algorithm.Mutation
   Genetic_algorithm.FitnessFunction
   Genetic_algorithm.Gene
   Genetic_algorithm.Individual
   Interfaces which defines each component.
4. Tsp.algorithms.approximations.SpanningTreeApprox
   MST 2-approximation algorithm
5. Tsp.algorithms.genetic.TspProblem
   A TSP representation. Extends the abstract class GeneticProblem.
6. Tsp.algorithms.genetic.crossover.OXCrossover
   Tsp.algorithms.genetic.crossover.PMXCrossover
   Tsp.algorithms.genetic.crossover.CyclicCrossover
   Tsp.algorithms.genetic.crossover.RandomOffspringGeneration
   Implementation for the different crossover types.
7. Tsp.algorithms.genetic.selection.TournamentSelection
   Tsp.algorithms.genetic.selection.RouletteWheelSelection
   Implementation for the different selection types
8. Tsp.algorithms.genetic.mutation.RandomSwap
Tsp.algorithms.genetic.mutation.AdjacentSwap
Tsp.algorithms.genetic.mutation.InvertedExchange

Implementation for the different mutation types.

9. Tsp.gui.GuiDriver

Contains implementation for the UI.

10. Tsp.tests – contains most of the test plans which will be introduced in the next sections.

11. Tsp.templates – A package that contains most of the TSP instances which we used in our experimental tests.

Specific implementation notes can be found inside comments in the code itself.

**Complexity Analysis**

Since the algorithm is probabilistic and is based on many random factors we will give a rough complexity analysis.

1. Given encoding in size $n$
2. The algorithm runs for $\text{maxGeneration}$ iterations – $m$
3. At each iteration, evaluation, selection, crossover and mutation are executed.
4. Evaluation $O(n)$
5. Each selections will take up to $O(n)$
6. Crossover:
   a. Cyclic crossover – $O(n)$
   b. PMX/OX – $O(n^2)$. However, we have made optimizations that consume more space in order to achieve an upper bound of $O(n)$
7. Mutation:
   a. Random swap / adjacent swap – $O(1)$
   b. Inverted exchange – $O(n)$
Thus, the total complexity of the algorithm is $O(mn)$

This analysis does not include analysis of the diversity of the population which will be discussed later on.
Experimental Tests

In this section we will introduce the experimental tests we conducted in order to test the differences between different types of selection, crossover and mutation.

In order to indicate how good our solutions are we applied the tests on four instances of TSP taken from TSPLIB95 (see references). Each instance contains a list of 51-152 cities. TSPLIB95 states the best known solution so far for each instance (we do not know how it was achieved or if it’s 100% reliable). The problems are listed in the following table, as well as the values of some of the relevant parameters for the GA.

Fixed parameters for all tests:

A. Four TSP instances:

<table>
<thead>
<tr>
<th>Problem</th>
<th>No. cities</th>
<th>Optimum (TSPLIB95)</th>
<th>MST 2-Aproximation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eil51</td>
<td>51</td>
<td>432</td>
<td>601.2</td>
</tr>
<tr>
<td>Eil76</td>
<td>76</td>
<td>545.4</td>
<td>743</td>
</tr>
<tr>
<td>Eil101</td>
<td>101</td>
<td>642.3</td>
<td>866.8</td>
</tr>
<tr>
<td>Pr152</td>
<td>152</td>
<td>73682</td>
<td>90557.7</td>
</tr>
</tbody>
</table>

B. Each test is run 20 times independently on each instance. Best and average results are taken.
C. Population size = 150
D. Max generation = 1000
E. Mutation rate = 3%

Test agenda:

<table>
<thead>
<tr>
<th>No. Test</th>
<th>Description</th>
<th>Selection type</th>
<th>Crossover type</th>
<th>Mutation type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Test different selection types</td>
<td>Roulette Wheel</td>
<td>PMX</td>
<td>Inverted Exchange</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tournament</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Test different crossover types</td>
<td>Tournament</td>
<td>PMX, OX, CX</td>
<td>Inverted Exchange</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Test different mutation types</td>
<td>Tournament</td>
<td>OX</td>
<td>Random swap, Adjacent swap, Inverted Exchange</td>
</tr>
</tbody>
</table>
### Experimental results

#### TEST 1:

<table>
<thead>
<tr>
<th>problem / selection type</th>
<th>Tournament</th>
<th>Roulette Wheel</th>
</tr>
</thead>
<tbody>
<tr>
<td>eil51.tsp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>best</td>
<td>438.052</td>
<td>450.1662</td>
</tr>
<tr>
<td>average</td>
<td><strong>448.3682</strong></td>
<td><strong>457.8112</strong></td>
</tr>
<tr>
<td>eil76.tsp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>best</td>
<td>597.305</td>
<td>622.7889</td>
</tr>
<tr>
<td>average</td>
<td>615.9328</td>
<td>634.8917</td>
</tr>
<tr>
<td>eil101.tsp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>best</td>
<td>740.0207</td>
<td>824.0275</td>
</tr>
<tr>
<td>average</td>
<td>768.121</td>
<td>1013.766</td>
</tr>
<tr>
<td>pr152.tsp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>best</td>
<td>116716.1</td>
<td>187900.5</td>
</tr>
<tr>
<td>average</td>
<td>127323.4</td>
<td>387294.5</td>
</tr>
</tbody>
</table>

#### Test results for selection types - eil101.tsp

![Graph showing test results for selection types on eil101.tsp](image-url)
**Summary:**

1. Tournament selection receives better results in all tests.
2. Over 51 cities, both types are close to optimum. However, as the number of cities increases the best solution is almost 70% worst than the optimal. In such case the MST approximation outputs better results.
3. Over 152 cities, roulette wheel selection is almost 150% worse than tournament selection.

**Conclusions**

1. As the number of cities grow, the chance to get stuck in local optima increases. This explains why as the number of cities grow, both types output results that are far from optimum.
2. Roulette wheel outputs lower results than tournament because individuals with lower fitness have a better chance to be selected for recombination in comparison to tournament. This is because the wheel is divided into 150 very small slots which can benefit with less fitted individuals. So it might be that recombination is done over less fitted individuals which will output lower results.
Test 2:

<table>
<thead>
<tr>
<th>file name  / crossover type</th>
<th>PMX</th>
<th>OX</th>
<th>Cyclic</th>
</tr>
</thead>
<tbody>
<tr>
<td>eil51.tsp</td>
<td>451.3019</td>
<td>432.123</td>
<td>475.4575</td>
</tr>
<tr>
<td>Best</td>
<td>461.8472</td>
<td><strong>439.1969</strong></td>
<td>504.8792</td>
</tr>
<tr>
<td>Avg</td>
<td>611.3813</td>
<td>563.2915</td>
<td>742.026</td>
</tr>
<tr>
<td>eil76.tsp</td>
<td>641.6387</td>
<td>576.6573</td>
<td>774.6911</td>
</tr>
<tr>
<td>best</td>
<td>797.4176</td>
<td>688.0397</td>
<td>974.1512</td>
</tr>
<tr>
<td>avg</td>
<td>832.9989</td>
<td>704.8496</td>
<td>1089.072</td>
</tr>
<tr>
<td>eil101.tsp</td>
<td>145672.3</td>
<td>86558.74</td>
<td>194136.2</td>
</tr>
<tr>
<td>best</td>
<td>169728.8</td>
<td>92844.86</td>
<td>209778.7</td>
</tr>
<tr>
<td>pr152.tsp</td>
<td>169728.8</td>
<td>92844.86</td>
<td>209778.7</td>
</tr>
</tbody>
</table>

Summary:

1. OX outputs better results in all tests. Over 51 cities optimum is found.
2. At 101 cities OX is only a 15% from best known optimum as opposed to PMX and CX.
3. CX output the worst results among the three types.

Conclusions

1. OX outputs better results because when the offspring receives a portion of genotype from one parent it then receives the portion of genotype from the second parent in a way that maintains the order of that portion. As opposed to PMX where the portion of genotype from the other parent is likely to be more scrambled, and thus the idea of creating an offspring by the combination of two parents has less effect.
2. CX can be ineffective in many cases, or can generate only a minor change in the encoding string and thus as the number of cities grow (and so is the encoding string) it outputs lower results.
**Test 3:**

<table>
<thead>
<tr>
<th>file name / mutation type</th>
<th>Inverted Exchange</th>
<th>Random Swap</th>
<th>Adjacent Swap</th>
</tr>
</thead>
<tbody>
<tr>
<td>eil51.tsp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>best</td>
<td>432.1621</td>
<td>446.2509</td>
<td>434.6247</td>
</tr>
<tr>
<td>avg</td>
<td>442.212</td>
<td>456.5343</td>
<td>456.6379</td>
</tr>
<tr>
<td>eil76.tsp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>best</td>
<td>568.5908</td>
<td>566.1373</td>
<td>581.3615</td>
</tr>
<tr>
<td>avg</td>
<td>581.2067</td>
<td>600.5834</td>
<td>601.7279</td>
</tr>
<tr>
<td>eil101.tsp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>best</td>
<td>686.1816</td>
<td>686.7593</td>
<td>715.2853</td>
</tr>
<tr>
<td>avg</td>
<td>709.965</td>
<td>727.3411</td>
<td>743.3891</td>
</tr>
<tr>
<td>pr152.tsp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>best</td>
<td>88377.2372</td>
<td>88132.192</td>
<td>91248.0135</td>
</tr>
<tr>
<td>avg</td>
<td>94298.5256</td>
<td>102716.38</td>
<td>101590.975</td>
</tr>
</tbody>
</table>

**Summary:**

1. In all tests inverted exchange reaches better average results than the two other mutation types.
2. Over 51 cities, optimum is found.
3. Over 152 cities, best result is better than the spanning tree approximation, however, average results are still lower.

**Conclusion**

1. Inverted exchange outputs better results as the number of cities is higher since the change it makes on the encoded string is more effective. Random swap and adjacent swap only swaps one city with another one. On a 101-152 cities string it will not cause a major change and even might create an individual with a lower fitness which will not survive in the next generations.

**Avoiding local optima**

In the previous section we noted that as the number of cities grows a local optimum is found. And indeed, one of the main problems in local search algorithms such as genetic search is the how to avoid from local optima. The cause for that is a premature convergence of the population which losses its diversity. That is, as the generations progress during the run of the algorithm, a large number of individuals share the same genetic material (those with higher fitness). When situation occurs, the selection operator selects with high probability two identical individuals. Then the crossover operation is ineffective since of the offspring is a clone of its selected parents. It is worth noting here that one of the purposes of mutation is to maintain a certain diversity level of the population, however, with a fixed mutation rate (usually low) it doesn’t prevent the premature convergence.

The following figure depicts the convergence rate of GA using Tournament Selection and Roulette wheel Selection, with population size 100 and over 1000 generations.
As expected, tournament selection converges much faster than roulette wheel selection, since individuals with higher fitness has a greater change to be selected in tournament an opposed to roulette wheel.

In this section we will introduce two techniques we have implemented in order to avoid the premature convergence to local optima.

1. **Social disaster operator**

   The general idea is to diagnose the situations of loss of genetic diversity of the population, and in such a case to apply a catastrophic operator to it. The purpose of this operator is to return the population to an acceptable degree of genetic diversity, by replacing a number of selected individuals, by others, generated at random.

   We have implemented the following SD operator:

   Of all the individuals having the same fitness value, only one remains unchanged. The rest of the population is generated at random.

   Pseudo-code: SOCIAL-DISASTER:

   1. Every 100 generations do:
      1.1. Analyze the diversity of the population.
      1.2. If one individual composes 85% of the population, apply SD:
1.2.1. Add each distinct individual from old population to the new population.
1.2.2. Complete the new population with randomly generated individuals.

The procedure of analyzing the diversity of the population can be found at:
TspProblem.java, analyzeDiversity

2. Random offspring generation
The idea behind the Random Offspring Generation (ROG) is to test the individual's genetic material, before the crossover operation, and if the two selected individuals are the same then a random offspring is generated and returned.

Implementation can be found at:
RandomOffspringGeneration.java

Experimental tests

Test agenda:

<table>
<thead>
<tr>
<th>No. Test</th>
<th>Avoid local optima</th>
<th>Selection type</th>
<th>Crossover type</th>
<th>Mutation type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ROG</td>
<td>Roulette Wheel</td>
<td>PMX</td>
<td>Inverted Exchange</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tournament</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Social disaster</td>
<td>Roulette Wheel</td>
<td>PMX</td>
<td>Inverted Exchange</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tournament</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Experimental results:

<table>
<thead>
<tr>
<th>file name / selection type</th>
<th>Tournament</th>
<th>Roulette Wheel</th>
<th>Tournament</th>
<th>Roulette Wheel</th>
</tr>
</thead>
<tbody>
<tr>
<td>eil51.tsp</td>
<td>ROG</td>
<td>Social Disaster</td>
<td></td>
<td></td>
</tr>
<tr>
<td>best</td>
<td>433.889</td>
<td>442.224</td>
<td>443.7282036</td>
<td>447.9162096</td>
</tr>
<tr>
<td>avg</td>
<td>448.857</td>
<td>456.679</td>
<td>455.0798635</td>
<td>453.0360183</td>
</tr>
<tr>
<td>eil76.tsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>best</td>
<td>596.753</td>
<td>607.948</td>
<td>587.6545156</td>
<td>600.6088069</td>
</tr>
<tr>
<td>avg</td>
<td>619.665</td>
<td>633.131</td>
<td>607.9333161</td>
<td>618.6403493</td>
</tr>
<tr>
<td>eil101.tsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>best</td>
<td>746.846</td>
<td>802.423</td>
<td>744.8920468</td>
<td>796.4839414</td>
</tr>
<tr>
<td>avg</td>
<td>760.119</td>
<td>866.690</td>
<td>767.0474856</td>
<td>1014.732967</td>
</tr>
<tr>
<td>pr152.tsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>best</td>
<td>115709.180</td>
<td>259658.18</td>
<td>119857.7454</td>
<td>270788.6617</td>
</tr>
<tr>
<td>avg</td>
<td>126166.681</td>
<td>359309.787</td>
<td>134200.1262</td>
<td>341042.2451</td>
</tr>
</tbody>
</table>

Conclusions

1. As for roulette wheel selection, ROG doesn’t output results that are far from the previous results without the use of ROG. In addition, over 152 cities, results are still very far from optimum. This actually corresponds to the previous diagram that shows that the convergence rate of roulette wheel is reasonable as the number of cities grow.

2. ROG does not deteriorate the performance of tournament selection, however it doesn’t enhance it either.

3. The same conclusions are correct also for the social disaster.

4. As for the differences between ROG and social disaster, we note that ROG outputs better results. We believe it's because social disaster maintains the population too diverse, which is not necessarily a positive thing.
Final Conclusions

1. The GA which we implemented is a very good alternative for solving hard problem in a relatively low time complexity. The GA outputted optimal results for up to 51 cities within a few seconds. All results were better than the MST 2-approximation.

2. As for problems with a large amount of cities, the GA outputs result that are pretty far from known optimum, however, very close and even better than the 2-approximation algorithm. Here we must note that finding a solution for such problems takes more time and consumes more space than with the approximation algorithm.

3. According to our experimental tests, best results are achieved by the following combination:
   a. Tournament selection. We are indifferent for the use of ROG.
   b. OX crossover
   c. Inverted Exchange mutation.

4. Avoiding local optima. Our attempts to avoid from local optima did not prove themselves. Further exploration is needed in this section. It seems that our implementation spun between a well diverse population and a population that lacked diversity. Other methods are needed in order to maintain a healthier balance of diversity. Some ideas that were brought up during this research and were not implemented:
   a. Adaptive mutation rate – as the population losses its diversity, higher mutation rate is needed in order to maintain diversity. This attitude seems a natural solution that conforms to the way the algorithm works.
   b. Seed “good” individuals when generating the initial population. By “good” we mean solutions of approximations for instance. In this attitude, instead of coming up with a solution at random, we can improve and existing solution that we know is close in some factor to the optimum.
Operating instructions

The file ai_project_tsp.ga.jar contains our eclipse project and an executable jar file. After extracting the jar, there are two ways to run the project:

1. Run the executable jar TSP_GA/GuiDriver.jar through a UNIX shell with the command ./GuiDriver.jar
   This option will work only on UNIX.

2. Import the project to eclipse and run the file:
   Src/tsp/gui/GuiDriver.java
   This option will work on every OS.

Instructions how to use the UI:

1. You can select a TSP instance from a given files list.
2. You must choose algorithm type.
3. You can generate a random instance of TSP by choosing amount of cities (will work only if a file is not selected).
4. Choosing a seed will generate the corresponding TSP instance.
5. Clicking on Solve will solve the problem with our own parameters.
6. Otherwise you can choose your own parameters and then click on the lower button “Start”. If you choose this option you must choose all the parameters.
7. Null parameters are not allowed. In such case the UI will crash and output an error message to the console.
8. The graphical solution will be displayed in addition to the output in the console, so be sure to check out the console too.
Bibliography

4. Larry Yaeger, Artificial Life as an approach to Artificial Intelligence, Intro to Genetic Algorithms.
8. TSPLIB95, http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/