**Java Example Project**

**Handbook v0.02**

This document provides additional information for some of the examples shown in the Java Example Project.

Contents

[Introduction 1](#_Toc495416178)

[Optimization problem 1](#_Toc495416179)

[Traveling Salesman Problem (TSP) 1](#_Toc495416180)

# Introduction

The Java Example Project (JEP) is a simple project to develop an application in Java, which provides a collection of java programming examples. Those examples can be executed and their source code can be viewed via the application. Some examples are simple and self-explanatory – no further information is required to understand them – however some examples are more complex, consist of multiple classes and are not as easy to understand. This handbook provides some additional information, which should help to understand these more complex examples.

# Optimization Problem

An **optimization problem** is a problem, where we search for the best (or a good enough) **solution** where is the **solution space**. To compare two solutions, we need a **fitness-/ rating-function** :

which maps a solution to a scalar fitness value (the value does not necessarily have to be real, it could also be natural etc.). To compare two solutions x and y we compare their fitness values. In most cases we are dealing with **maximization** problems, where a higher fitness is better. In some cases, we deal with **minimization** problems, where lower fitness is better.

Given two solutions :

This relation can be expressed in a **comparison function** (here for maximization; the function for minimization is given analogue):

The solution spaces will likely be different for different problems. The solutions will be represented in different formats. For example, if the problem is “Find the optimal parameters for this function”, the solution can be represented as a vector of the parameters (which are to be optimized). If the problem is the traveling salesman problem (a permutation problem), the solution can be represented as a fixed order of cities.

It is possible to turn any minimization problem into a maximization problem and vice versa simply by multiplying the fitness by -1. Let’s say our problem is to find the shortest route from a city A to a city B of two possible routes R1 and R2 with and . The problem is an optimization problem and R1 is the better solution. Instead of using as comparison (which would be true) we could use – the comparison for maximization problems with the fitness values multiplied by -1 – (which is also true).

This means we can only look at one of the two problem types – maximization or minimization – and the user would simply have to define the fitness function of the optimization problems solutions accordingly. In the JEP we only look at maximization problems.

## Traveling Salesman Problem (TSP)

The **general** **traveling salesman problem** (TSP) (as defined in JEP): “A traveling salesman has to visit each city of a fixed set of cities. He must visit each city only once, except the start city, at which his route must end. The set of cities is fully connected, meaning there is a direct connection between each pair of two cities of the set. Each connection has a fixed distance. We search the route, which visits all cities only once and has a minimal total distance (sum of distances of the routes connections).”

Fig. 1: Traveling Salesman Problem - Cities A, B, …, G as circles with connections as lines. possible route marked via orange arrows with A as start and end of the route.

May be the set of cities which are to be visited, where are cities, a route can be given as a fixed order of cities like:

meaning, we visit A then B then D then C then E then G then F then A. The total distance is given as the sum over the distances between each pair of two cities which are connected (including ).

### Specific Traveling Salesman Problem

The **specific traveling salesman problem** implemented in the JEP, uses a set of 13 capitals as city-set. The distance between two capitals is given as the great circle distance (flying distance) in kilometers. The distances are stored in a symmetric matrix (>> Table 1: Specific traveling salesman problem – distance matrix), they were calculated using <https://www.distancecalculator.net/> (10.10.2017).

# Corrective Procedures

A **corrective procedure** is a procedure which tries to find the optimal (or a good enough) solution for a given optimization problem. Given a solution space and a fitness function the best solution is a solution where (if the problem is a maximization problem):

A good enough solution might be defined as a solution where – given a threshold (if the problem is a maximization problem):

One of the simplest corrective procedures is the **hill climbing algorithm**, its general idea is to search through the solutions, only accepting solutions whose fitness is better than the current solutions fitness.

Listing 1: Hill Climbing Algorithm – as pseudo code (for maximization problems)

solution **:=** pick\_Initial\_Solution()

**WHILE** **(** break\_Condition\_Unfulfilled() **) {**

neighbor **:=** pick\_Neighbor(solution)

**IF** **(** fitness(neighbor) **>** fitness(solution) **) {**

solution **:=** neighbor

**}**

**}**

Looking at “Fig. 2 Graphical representation of the fitness function” we already see the issue of the hill climbing algorithm (>> Listing 1: Hill Climbing Algorithm) – it only allows better solutions, which means, if we reach a local optimum, we won’t get out of it, therefore we will never reach the global optimum.

f

global maximum

local maximum

Fig. 2 Graphical representation of a fitness function

Table 1: Specific traveling salesman problem – distance matrix

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| in km | BEIJING | NEW\_DEHLI | TOKYO | MOSCOW | LONDON | BERLIN | MADRID | ROME | PARIS | DUBLIN | OTTAWA | WASHINGTON\_DC | CANBERRA |
| BEIJING | 0 | 3784 | 2095 | 5800 | 8150 | 7366 | 9233 | 8134 | 8226 | 8292 | 10463 | 11158 | 9018 |
| NEW\_DEHLI | 3784 | 0 | 5844 | 4347 | 6719 | 5787 | 7282 | 5923 | 6594 | 7086 | 11351 | 12059 | 10365 |
| TOKYO | 2095 | 5844 | 0 | 7487 | 9569 | 8926 | 10774 | 9864 | 9723 | 9595 | 10333 | 10916 | 7961 |
| MOSCOW | 5800 | 4347 | 7487 | 0 | 2503 | 1611 | 3444 | 2378 | 2489 | 2798 | 7166 | 7830 | 14498 |
| LONDON | 8150 | 6719 | 9569 | 2503 | 0 | 933 | 1265 | 1435 | 344 | 464 | 5367 | 5904 | 17001 |
| BERLIN | 7366 | 5787 | 8926 | 1611 | 933 | 0 | 1871 | 1184 | 878 | 1318 | 6135 | 6718 | 16084 |
| MADRID | 9233 | 7282 | 10774 | 3444 | 1265 | 1871 | 0 | 1366 | 1054 | 1453 | 5696 | 6095 | 17593 |
| ROME | 8134 | 5923 | 9864 | 2378 | 1435 | 1184 | 1366 | 0 | 1107 | 1888 | 6737 | 7225 | 16235 |
| PARIS | 8226 | 6594 | 9723 | 2489 | 344 | 878 | 1054 | 1107 | 0 | 782 | 5655 | 6172 | 16939 |
| DUBLIN | 8292 | 7086 | 9595 | 2798 | 464 | 1318 | 1453 | 1888 | 782 | 0 | 4905 | 5448 | 17256 |
| OTTAWA | 10463 | 11351 | 10333 | 7166 | 5367 | 6135 | 5696 | 6737 | 5655 | 4905 | 0 | 734 | 16126 |
| WASHINGTON\_DC | 11158 | 12059 | 10916 | 7830 | 5904 | 6718 | 6095 | 7225 | 6172 | 5448 | 734 | 0 | 15962 |
| CANBERRA | 9018 | 10365 | 7961 | 14498 | 17001 | 16084 | 17593 | 16235 | 16939 | 17256 | 16126 | 15962 | 0 |

However, the algorithm shows the necessary base components of a corrective procedure – being the *initial solution*, the *neighborhood-function*, the *acceptance method* and the *break condition*.

* *Initial solution*: An initial solution can be given as a manual input, via a heuristic or simply as a randomly generated solution.
* *Neighborhood-function*: This function constructs a new solution – the neighbor – using the current solution as input. The function should be simple – Ex. if the problem is finding the optimal parameter, the neighborhood-function could create a new solution by changing one parameter of the current solution at random, if the problem is a permutation problem, the function could swap two elements.
* *Acceptance Method*: This method defines which neighbors are accepted as new solution. For the hill climbing algorithm, only neighbors which are better as the current solution are accepted. Other algorithms for example also allow the acceptance of worse solutions.
* *Break Condition*: The corrective procedure must end at some point. This condition could be general, like end after a set time or a set number of iterations. It also could be algorithm specific or it could be a combination of multiple general or algorithm specific conditions.

The first specific corrective procedure implemented in the JEP is the **threshold accepting procedure**. The general idea of this algorithm is to define an initial threshold wish is slowly decreased during the algorithms runtime – for examples at certain times where no better solutions were found: with . The algorithm allows the acceptance of neighbors whose fitness is within threshold range of the fitness of the current solution, meaning the algorithm allows to accept solutions which are worse than the current solution within range of the threshold.

Listing 2: General corrective procedure pseudo code

solution **:=** pick\_Initial\_Solution()

**WHILE** **(** break\_Condition\_Unfulfilled() **) {**

neighbor **:=** pick\_Neighbor(solution)

**IF** **(** accepted\_Neighbor(neighbor, solution) **) {**

solution **:=** neighbor

**}**

**}**

The parameters is problem dependent and must be picked accordingly. The parameter defines how quick/slow is decreased. As smaller is picked as faster is decreased – which likely is not what we want.

The algorithm specific break condition can be defined as a condition with . Since is a function which converges to 0 but never reaches 0, the algorithm would run on (until a general break condition is met) even if such small values of might make no sense for the given problem – which would result in the algorithm calculating a series of neighbors, none of them being accepted.

Listing 3: Threshold Accepting Procedure

solution **:=** pick\_Initial\_Solution()

T **:=** pick\_Initial\_Threshold() **//Ex. via manually set parameter with T > 0**

**WHILE** **(** break\_Condition\_Unfulfilled() **) {**

neighbor **:=** pick\_Neighbor(solution)

**IF** **(** fitness(neighbor) **>=** fitness(solution) **-** T **) { //acceptance**

solution **:=** neighbor

**} ELSE IF (** no\_Improvement\_For\_Some\_Time() **) {**

T **:=** decrease\_Threshold(T)

**}**

**}**

To check if no better neighbors were found in some time, we can simply define a counter variable we increment each time an unaccepted neighbor was picked and reset if a better neighbor was found or if we decreased the threshold.

The second specific corrective procedure implemented in the JEP is the **simulated annealing procedure.**

**TODO**

# Genetic Algorithm

Genetic algorithms (GA) are algorithms of a specific structure inspired by evolution which can be used as optimization procedures. The corrective procedures explained above could also be expressed as genetic algorithms.

Genetic algorithms are inspired by nature, by evolution. Certain methods of nature come into play for the development of a fit species:

* Crossover: Most animals reproduce via sex. Two individuals – the parents – generate a new individual – the child – by recombining their DNA.
* Mutation: Crossover can only find solutions (animal generations) in a limited space about the initial population. Mutation introduces a random factor which allows new DNA sequences to exist whose would otherwise be unreachable.
* Selection: Survival of the fittest – or rather of the fit enough – solutions which are not “fit enough” die, leaving only the fitter solutions behind which than can reproduce and generate even better solutions.

Any corrective procedure could be viewed as a genetic algorithm with its neighborhood function being the mutation function and its acceptance function being the selection function. Since we select via fitness a fitness function is required as well as a break condition which defines at which point we stop the algorithm. Other than for corrective procedure we don’t look at single solutions but rather at populations – a set of individuals where an individual is a single solution.

Listing 4: Genetic algorithm pseudo code

population **:=** select\_Initial\_Population()

**WHILE (** break\_Condition\_Unfulfilled() **) {**

child\_Population **:=** crossover(population)

child \_Population **:=** mutate(population)

population **:=** select(population, child \_Population)

**}**

Some implementations of genetic algorithms use elite selection. This form of selection is “unnatural” an allows that not only individuals of the child population but also the best individuals of the parent population can be selected for the new generation. Our implementation does support this, but it should be noted that elite selection is to be used with care. For example if the elite are individuals of a local optimum than the continues reselection of those individuals might slow down the progress towards the global optimum.