Traveling Salesman Problem

Java Genetic Algorithm Solution

author: Dušan Saiko
23.08.2005

Index

Introduction

The task of this application was to explore the possibilities of genetic programming and to test it on a well known Traveling Salesman Problem (TSP), where the salesman should visit given number of cities and then return back home, all in the shortest way possible.

For more TSP info, resources and research activity, see http://www.tsp.gatech.edu/, where they also present the results of several unique computations for thousands of cities. For solving TSP, numeric, heuristics, genetic, hybrid or other algorithms may be used, as this problem is not easy to solve for large number of cities, the computation is often deployed on multi processor or clustered hardware (see the http://www.tsp.gatech.edu/ for solution on 25000 Swedish cities, using cluster of 96 computers with dual processor).

Still the goal of this research was mainly to explore genetic algorithms more deeply and get overview over the existing genetics approaches and engines available in Java.

Genetic algorithms

The overview of genetic algorithm principles could be read at http://en.wikipedia.org/wiki/Genetic_algorithm .

There could be defined several levels of important tasks for successful implementation of genetic algorithm.

Code the problem

The problem has to be coded into data structure, which can be handed like a chromosome.

For TSP, the chromosome is set of ordered indexes of cities, through which the traveler goes. For other problems, it could be integer or real number, for difficult tasks, there is idea of using Neural Networks for coding the problem.

Define fitness function

The fitness function evaluates each chromosome and sets the numeric value to it, which represents the quality of the chromosome – e.g. of the solution, which the chromosome represents. Fitness function is then used for evaluating the population and preferring the higher quality of individuals for mating and creating offspring.

For TSP, the fitness function of chromosome is computed at the total distance of the represented solution. But even here, computing of the cost of solution is not so easy and could be research, as the total distance equation does not have to be the best fitness function. As TSP algorithm tends sometime to create quite long distance connections, root mean square (RMS) value could be used to compute the cost. RMS value is just sum of squre roots of distances between the cities in path which is encoded in chromosome. In this way, we could prefer more expecting solution of (in

distances) 2 3 2 2 (total distance=9, RMS cost 21) against chromosome 1 2 1 5 (total distance 9, but RMS distance 31).

For more difficult applications, the fitness function could be defined in complex abstract and non exact way that only tries to compare the quality of chromosomes against each other, but fitness function by itself does not return any meaningful information.

Select genetic algorithm engine

The genetic algorithm engine cares about the population, its growth, filtering, selecting and sorting individuals and random mutations of chromosomes. It also handles all the computation process and optionally enables multi threading processing of the problem.

Mating algorithm

Mating algorithm is very important way, how to create offspring (child) from the parent chromosomes. The task of the mating is to create new offspring, which has characteristics from both parent and improves the quality returned by fitness function. I would say this task is most difficult and most important as it decides, how well and how fast will the population improve.

Implement random mutations

Random mutations are side by side to mating algorithm important for moving the population from local extrema. It could happen that the computation is in such a state locked in non optimal position and needs external (random) impulse to break the disability apart an start again moving ahead. On the other hand, if there is too much of random process into computation, the result will never be the most optimal.

Think about using heuristics

Using heuristics can speed up the computation very much, but with heuristics, we little bit shift from genetic programming to hybrid combination with analytical programming. As some problems can be only hard to be analytically described, to write properly the heuristic functionality may be not easy task. With using heuristics, we lose some generic problem solving characteristics of the genetic algorithm. Proper and effective heuristic can reduce the need to have big population base and can operate well on small set of chromosomes, as it can be seen in the results of this implementation of TSP.

Apply the right initial parameters

Other sensitive problem is to choose the right initial parameters of the computation, as population size, mutation ratio, population growth. These parameters are relevant to used algorithm, and can influence the computation process a lot. Pure genetic algorithms usually need bigger population base, algorithms with heuristics could be for the same population base much slower, but could be quite fast and efficient even for small population. The same is with mutation ratio and population growth. As I have tried computations with population size up to 500.000, then even the randomization of part of this population can be quite time expensive.

Initialize population

Initialization of population is probably the last of application issues to be solved, thus is not also easy to decide. In TSP, the population is randomly initialized and mixed, but the initialization of population could be more sophisticated task to do and think about. The initial population is base for all further population growth and development, so if the population is initialized incorrectly, e.g. not enough randomized for TSP, then the whole computation is going in wrong way.

Run the computation on proper hardware

Genetic algorithm applications often require a big computation power of computer, multi threading adaption or even multi computer clustering of the task. As mentioned http://www.tsp.gatech.edu/sweden/compute/compute.htm , for computation of TSP (without genetics) of 25000 cities, the cluster of 96 dual processor workstation was used. As we do not have usually access to such a computation power, we are limited by tasks which could be solved by accessible hardware. University support or spreaded Internet application clients (in means of SETI program) give ideal environment for genetic task solving.

Different approaches

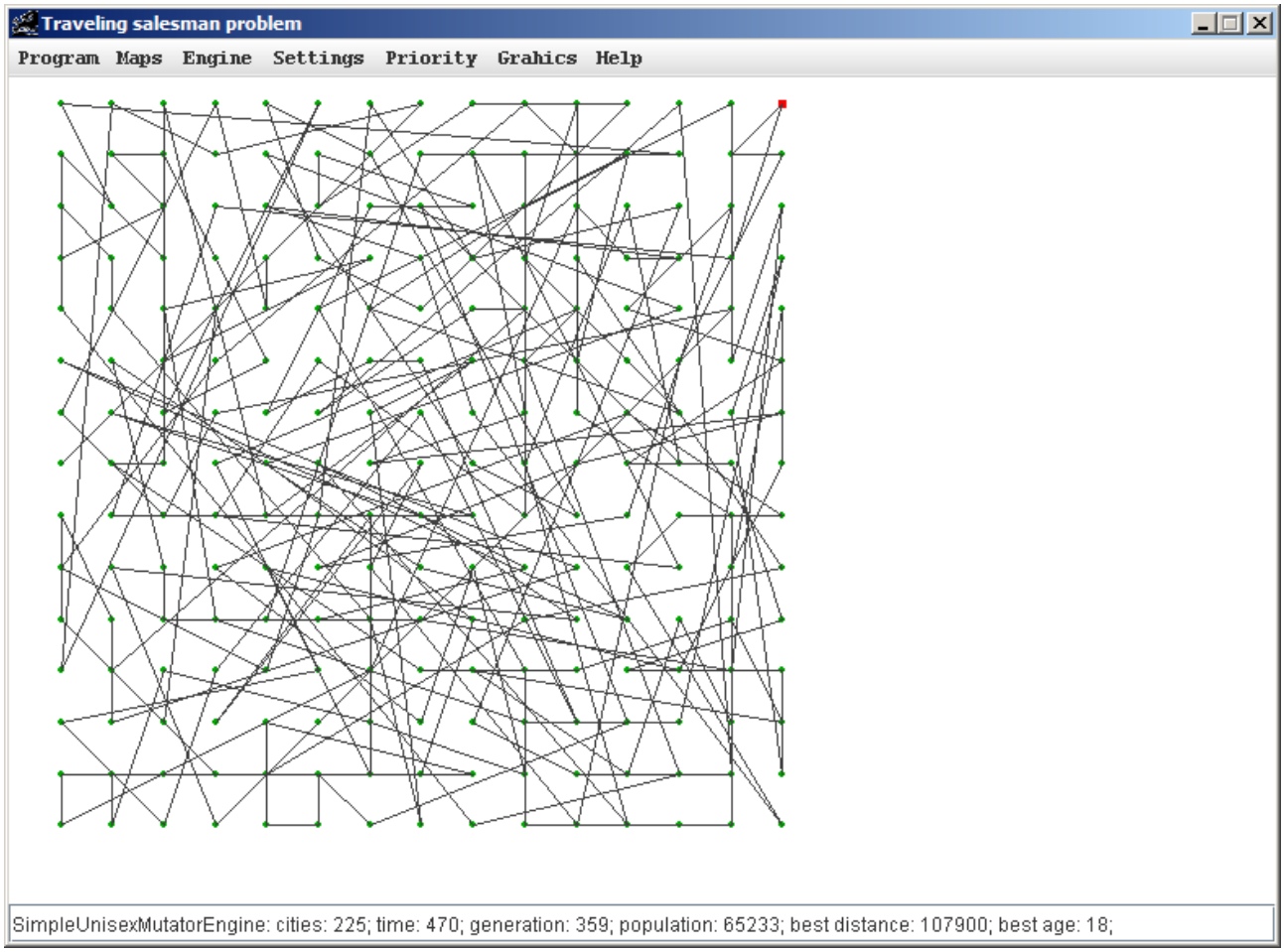
There are the following genetic algorithms used in the application:

SimpleUnisexMutatorEngine

This most simple genetic approach is based on just a random mutations of a single chromosmeme units. The whole population is in this way quite sensitive on the initial conditions and the inheritance of chromosome characteristic is streight lined, there is no cross mutation of chromosomes.

In the implementation of the application, this engine is used as a parent engine for most of others algorithms, as it implements general functionality for randomizing the population and selecting individuals and mating and mutating them, all in the multi threaded environment.

The SimpleUnisexMutatorEngine gives not very good results, but is very simple and in its simplicity it could be still used to optimalize some TSP maps, but with need of large population base.



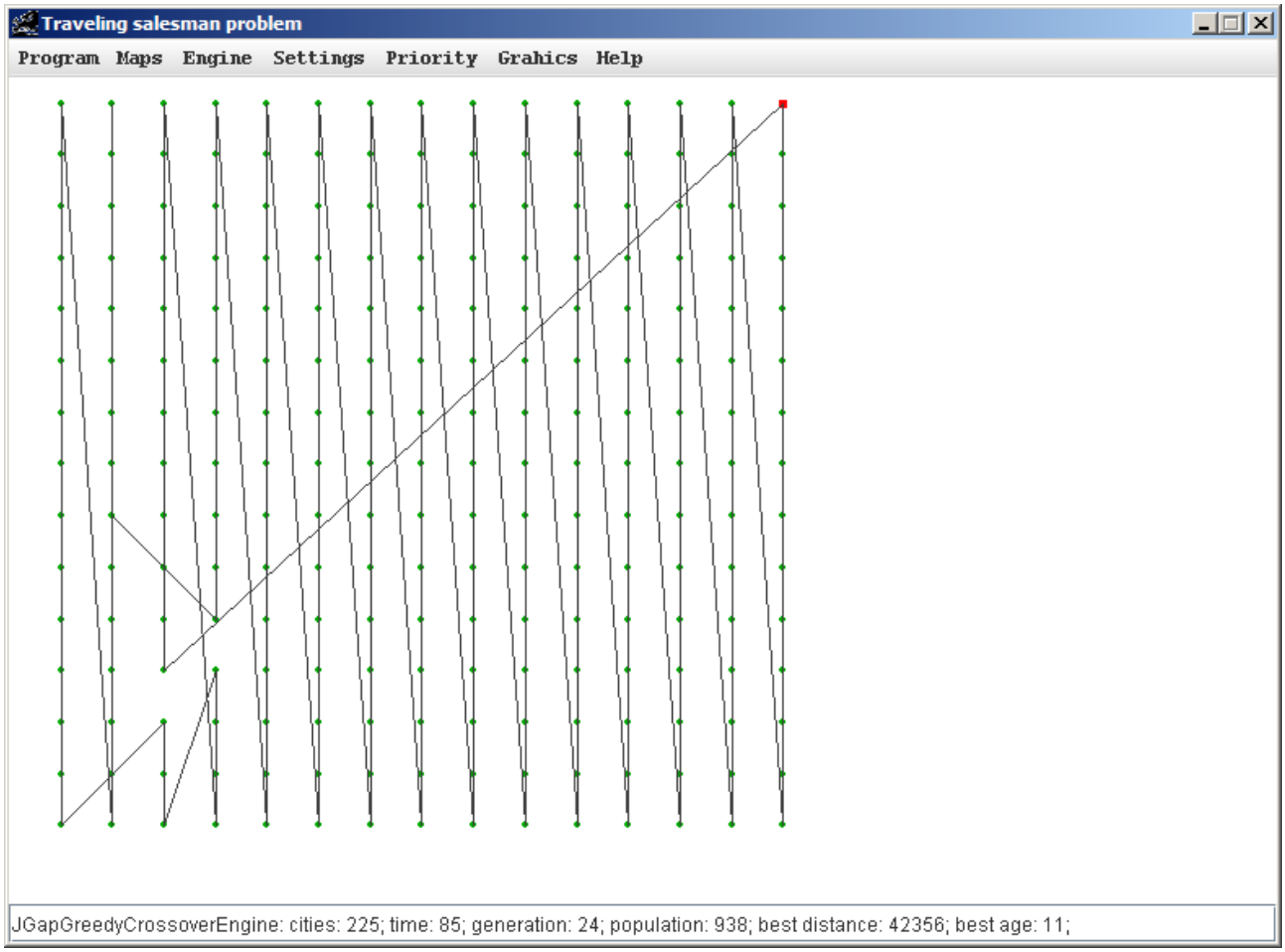
The random mutation only is not enough to solve complicated tasks, but even this simple algorithm could be examined and used for improving the salesman path.

JGapGreedyCrossoverEngine

This engine is based on the JGap (http://jgap.sourceforge.net/) sample for solving of TSP. It uses GreedyCrossover mutation to mate two chromosomes for getting a child. The mating algorithm is for example also described here http://www.gcd.org/sengoku/docs/arob98.pdf

JGap allows in quite nice object way to set lots of genetic properties, use different population transormation etc. But all the complexity of JGap is quite large and generic, in my opinion it was an overkill for the TSP task and the solution is too slow, too complicated and on the other hand too restrictive. JGap also implements in quite complicated way the possibility to use mutlti threading computation. So for me, it was better and more effective to create my own implementation of the greedy crossover mutation. For feature, I would like also to experiment with more complex handling of the population, like aging of chromosomes, migration of chromosomes in the population etc.

The JGap engine tends to provide, in comparison to other examples, some quite different and unique (but not very correct) solution for the square map problem solving.



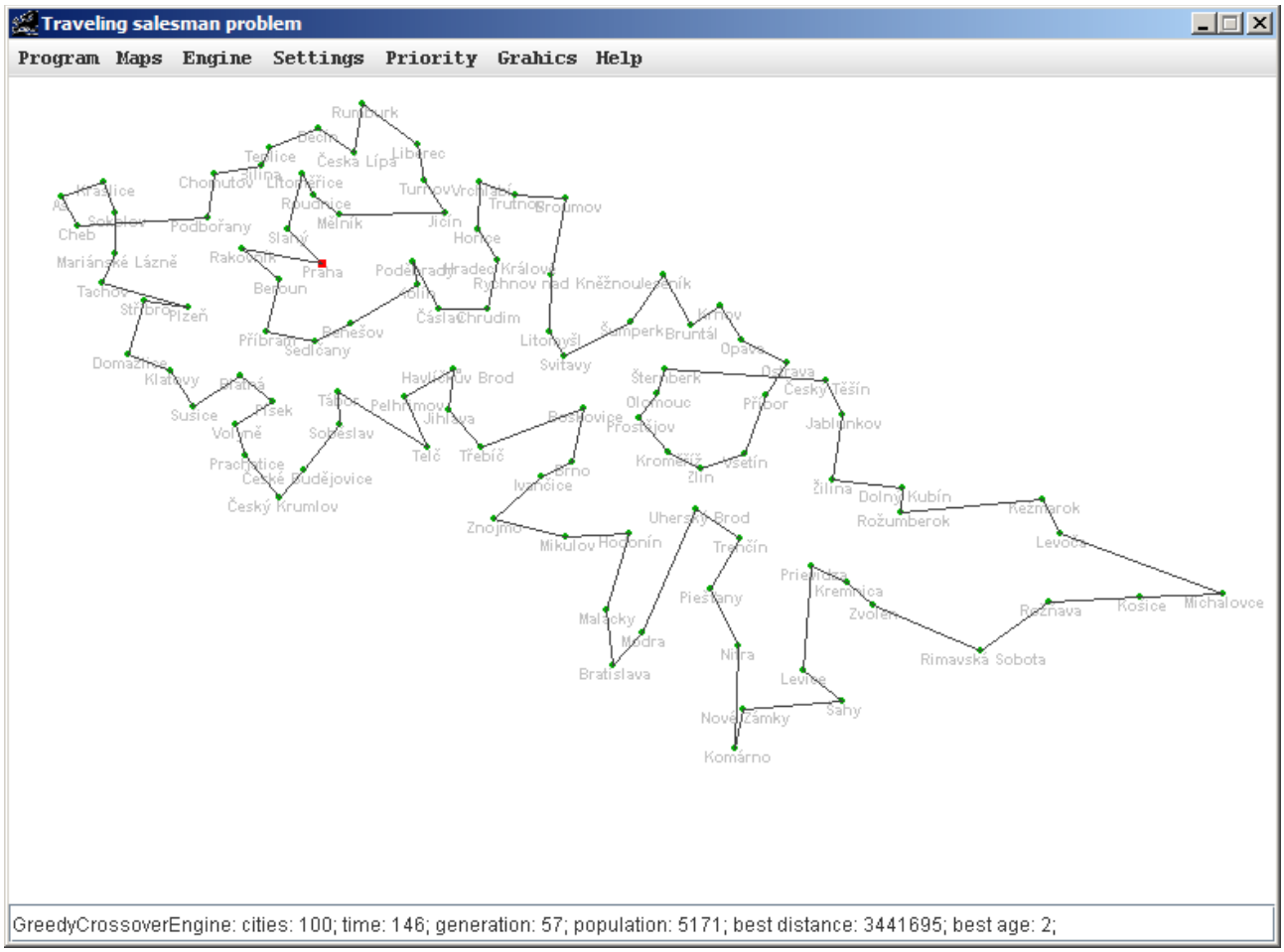
JGap gives unique results for solving square map; but this result is not very good.

GreedyCrossoverEngine

Implementation of greedy crossover algorithm based on sources from JGap, but optimized for simplicity and speed. In implementation, it extends the SimpleUnisexRandomMutator by providing new mating functionality.

This is probably the best achieved pure genetic solution. It is quite generalized without using any analytical functionality, except the mutation algorithm, which is constructed on the knowledge of the chromosome structure and meaning.

The engine gives quite good results for most of the maps, but needs more time and larger population for solving the problem. It goes to some “quite good” solution quite fast, but then it has tendency to solve quite hard some local extrema complications.

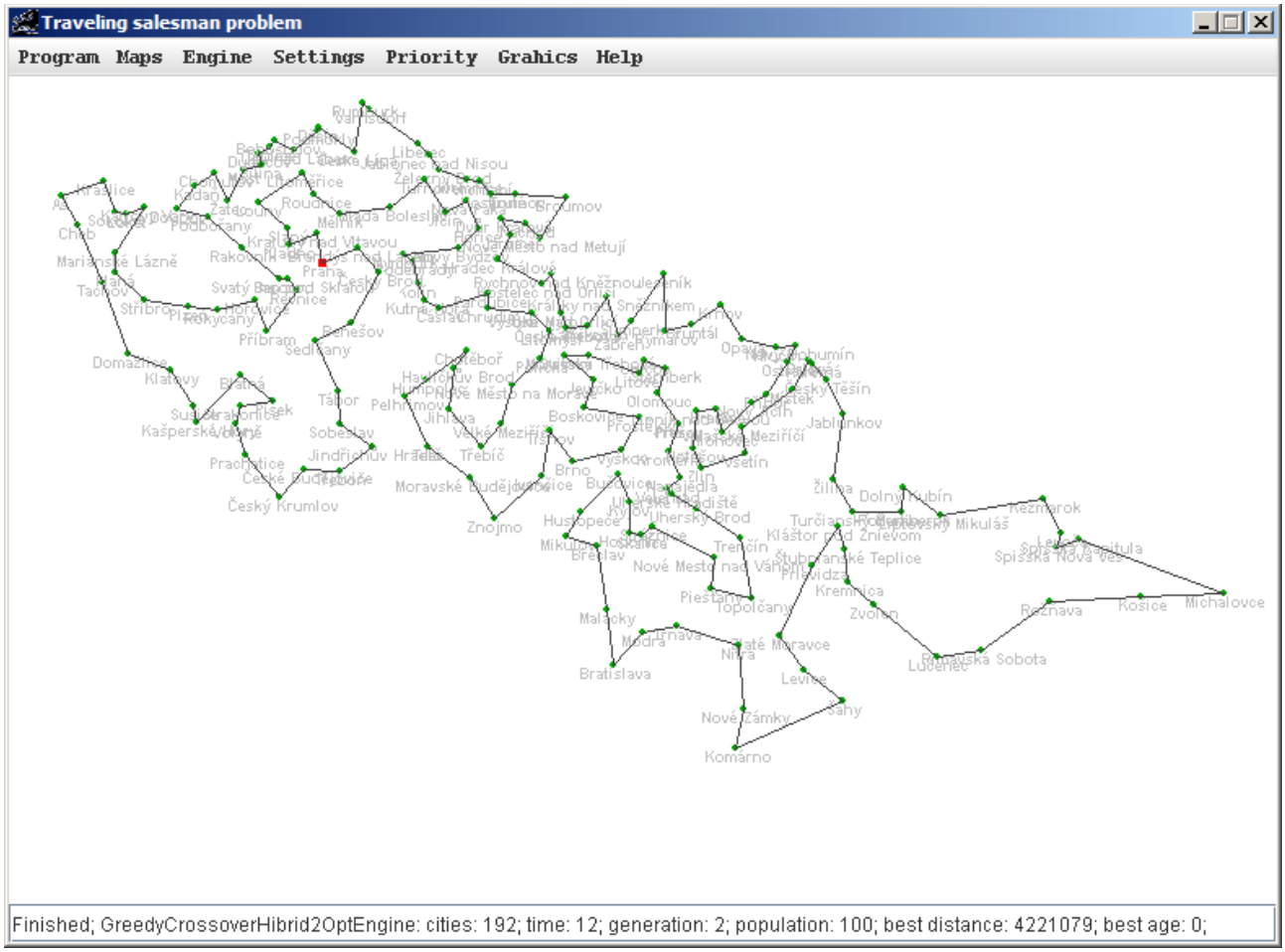


Greedy crossover mutation is quite good, it has very good progress at beginning, but then it tends to solve hard the local complications.

GreedyCrossoverHybrid2OptEngine

This engine extends the GreedyCrossoverEngine and adds the heuristics optimalization of the chromosome by 2opt algorithm (http://www.gcd.org/sengoku/docs/arob98.pdf). This algorithm is quite simple, and provides the way, how to get immediatelly rid of all the path crossings. The heuristics are quite fast, but when applied to all chromosomes in large population, they slow the computation a lot.

By using heuristics, we speed up the path finding very much, but we have to use the analytical algorithm and knowledge for solving the problem – this will not be (easy) possible to use in all more difficult situations. GreedyCrossoverHybrid2OptEngine combines the heuristics optimization and the genetics to get out of the local extremas. For computation, smaller population is sufficient and the "quite good" results are retrieved immediatelly after first optimization.

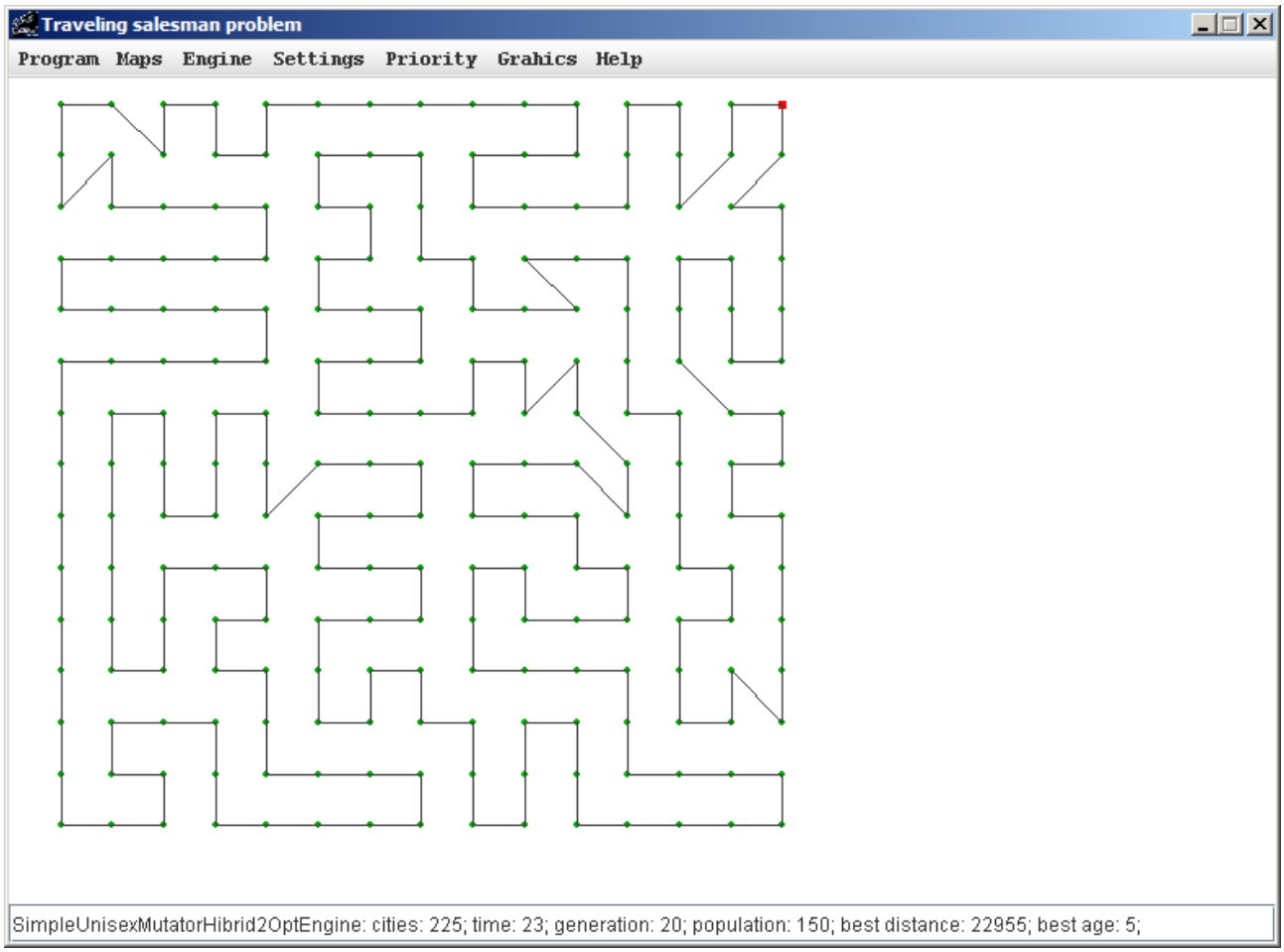


Best solution - combination of greedy crossover mutation together with 2opt heuristics. It computes very good results very fast.

This solution gives the best and fastest results.

SimpleUnisexMutatorHybrid2OptEngine

This engine is similar to the preview one, except the mutation is only based on randomization of single chromosome. This solution is more like using the 2opt only, with some random mutations added. Because of 2opt, it gives very good and very fast results, but this approach is little bit more far from the ideal pure genetic algorithm solution.



2opt optimization by itself gives better results than JGap for this square map.

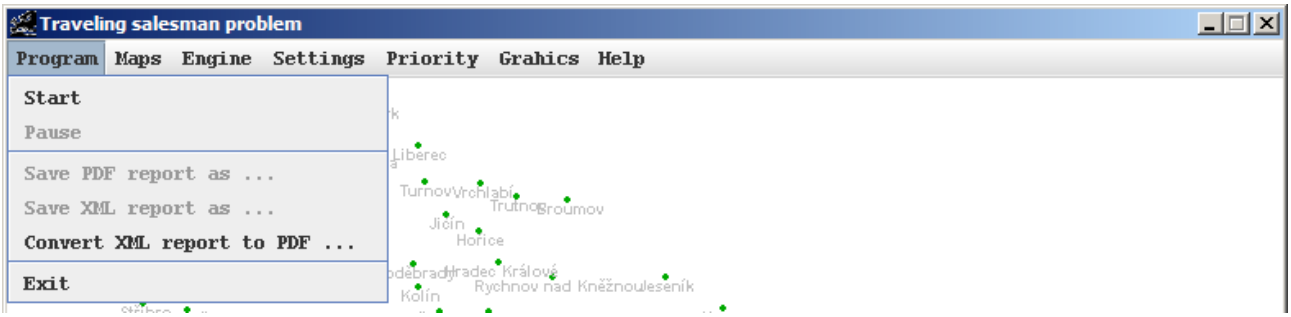
Application description

Application features

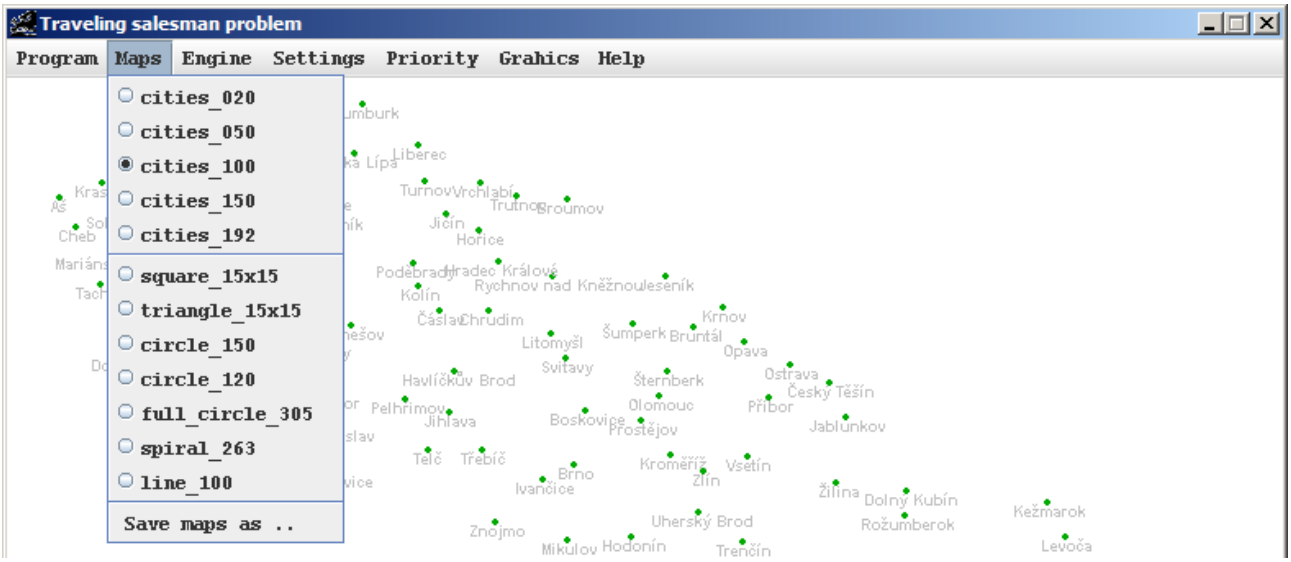
- Implementation and comparison of different genetic algorithms
- Open source multi platform Java application with well commented source code
- Console and GUI application mode
- Graphical result displaying
- Parametrized configuration
- Multi threading computation engine
- Application thread priority settings
- Simple map file formats, exporting existing maps, using external maps
    - map of 192 real cities from Czechoslovakia, coordinates in S-JTSK, distances in meters
    - fractal maps (circle, triangle, square, spiral, ...)
- Descriptive XML and PDF reports, converting XML reports to PDF

GUI

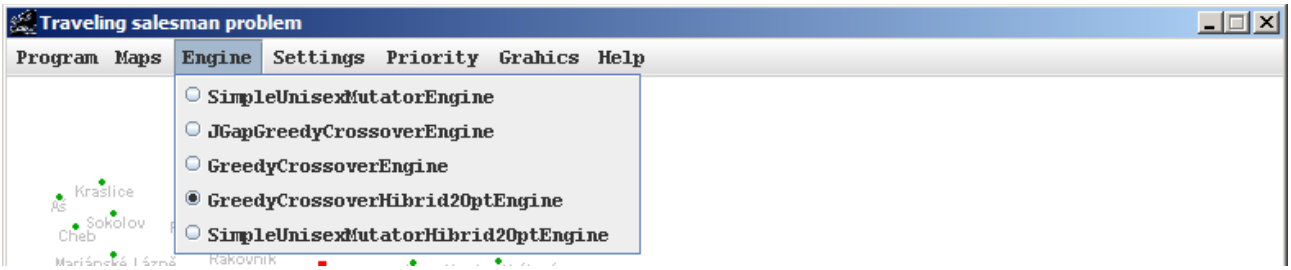
Program menu enables starting, pausing, resuming and stopping the application, create PDF report, XML report, convert XML report to PDF and exit the application. The Creating of report is possible only when computation is finished or stopped.



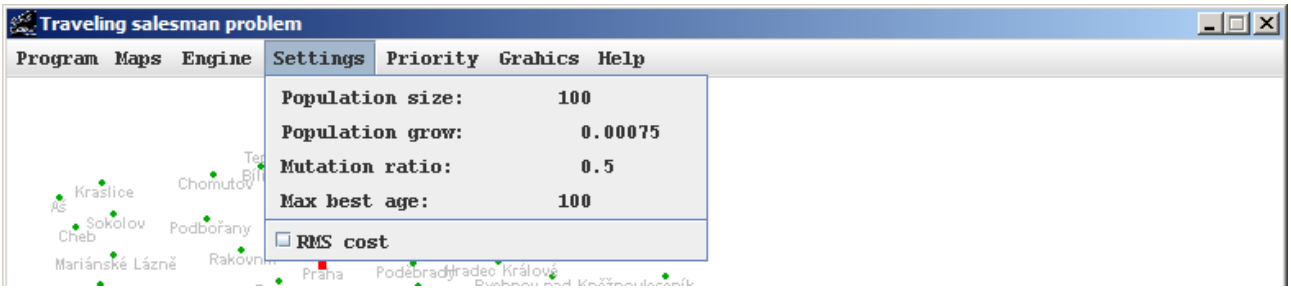
Maps menu allows to select map for TSP solving. It also has the option to export existing maps to single .zip archive with individual maps like .cvs files. Selecting the map is available only when computation is not running.



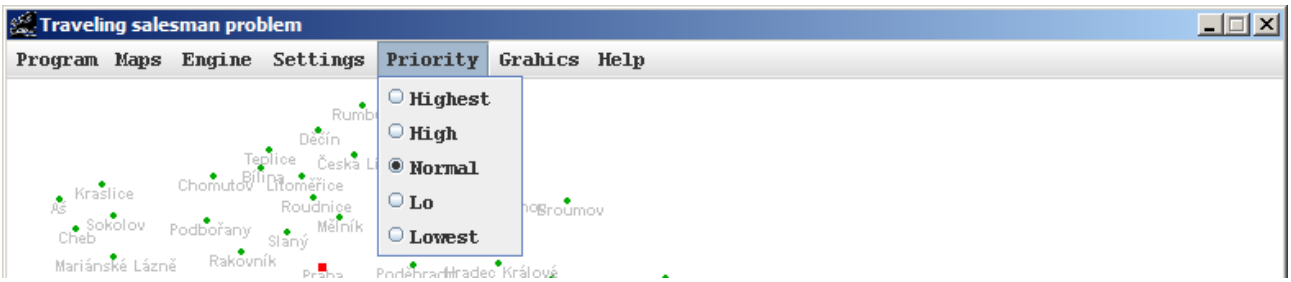
Engine menu enables to select computation algorithm engine. The selection is available only when computation is not running.



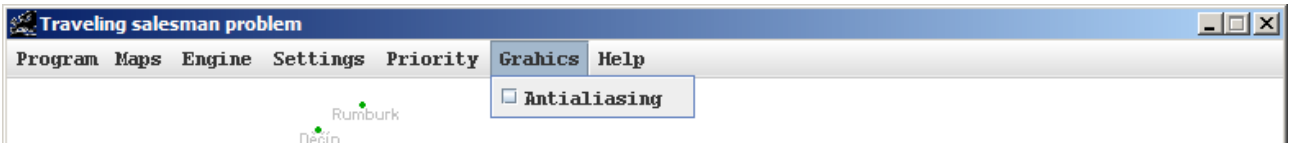
Settings menu allows to set the algorithm parameters before the computation is started.



In **Priority** menu, you can set the thread priority of the application. Lowest=1, Highest=10. This priority could be changed even during the computation.



Graphics menu lets user to turn on antialiasing displaying. This slows refreshing of the results and even antialiasing is not available on all server-side systems wit missing graphical libraries. Eg. when I was running remotely the application on Sun server, the antialiasing was not working (and also the processing of screen image for PDF export was crashing on image transformations).



Help menu with basic version and contact information.

Command line parameters

To be able to run the computation as a background server task, there is the possibility of running the application in a console mode. This mode takes the command line parameters, starts computations and after finishing, writes the report to XML file. This XML report could be than converted to PDF report from GUI instance of the application. This conversion will even create the picture of path and insert it into PDF report.

The disadvantage of console mode is, that the results are not visualized immediately and we do not know, if the live result of computation (which is shown on console) is close to best result, or if it is just still only the representation of mainly random paths.

Possible command line parameters are

- --*console* option to set the output to console, no graphics displayed
- --*map=NAME* where name is name of resource with .csv format, you can use only number part (020, 050...) as map name for built in country maps, or specify full resource path or path external resource (which still has to be in classpath).
- --*priority=N* where N in <1..10>; DEFAULT 5
- --*engine=N* where N is the index of engine to use; DEFAULT engine is GreedyCrossoverHibrid2OptEngine
- --*rms=T* where T in <true,false> - computes RMS cost from distance; DEFAULT false
- --*population=N* where N is the initial population size. DEFAULT 5000
- --*max=N* where N is the max number of the same best result; DEFAULT 100
- --*growth=N* where N is population growth. DEFAULT 0.0075
- --*mutation=N* where N is mutation ratio. DEFAULT 0.5
- --*xml=FILE* where FILE is output name for XML report file, DEFAULT filename is tsp_report_yyyy_MM_dd_HH_mm.xml

Example:

--console --map=192 --priority=1 --engine=3 --rms=false --population=200 --growth=0.01 --max=200 --mutation=0.5

--console

--help

Summary

From the application development and research process, following resume could be done.

Genetic algorithm is great approach for general problem solving tasks, with some limitations and not for 100% predictable and verifiable results. Still it is very interesting subject to explore and try, and I will continue in research over this and similar topics of AI.

There exist lots of examples and literature over the Internet for genetic programming. But, as the principles of genetic programming are quite simple, most sources and examples do not use any common library or framework of genetic functionality, even some of them exists.

The most stable Java genetic programming framework is created by JGap project, but I found it more comfortable and effective to suite the application needs with my own implementation of genetics. For the feature experiments, I would even like to try some more features of handling with population, as aging of chromosomes, advanced migration and mixing of population etc.

On the tests and examinations of the application results, it could be seen the importance of heuristics analysis integrated into genetic algorithm and the rapid increase in performance. With different sets of parameters, the question, if the genetic algorithm is reusing heuristics or if it is in the opposite way, arise.

The best, most precise and most fast solution of solving TSP was found by the GreedyCrossoverHybrid2OptEngine algorithm.

Links

- Application home
  http://www.saiko.cz/ai/tsp/
- Definition and research of Traveling Salesman Problem
  http://www.tsp.gatech.edu/
- Basic description of genetic algorithms
  http://en.wikipedia.org/wiki/Genetic_algorithm
- Suggested and probably best way for solving TSP using genetic algorithm and 2opt heuristic optimization described by Hiroaki Sengoku and Ikuo Yoshihara
  http://www.gcd.org/sengoku/docs/arob98.pdf