Luke Scott

Intelligent Systems: Problem Solving
MiCS Program
University of Luxembourg

Genetic Algorithms Lab
Friday 23rd December, 2011

Genetic Algorithm for the Traveling Salesman Problem

1 Introduction

This paper describes a Genetic Algorithm solution for the Traveling Salesman Problem (TSP). In this algorithm, a Chromosome consists of a list of unique integer alleles; these integers in turn represent cities, whose coordinates are read in from a text file during the initialization of the algorithm. The task of the algorithm is to successively refine these chromosomes by re-ordering the allele integers (making sure they remain unique) and scoring each new combination based on the total weight (i.e. total distance). A high score corresponds to a low weight; that is, this algorithm seeks a cost-minimizing solution.

2 Refactoring

For this assignment, we were given the source code to "ssGA: Steady State GA" [1] as a starting point. This code is intended for two use cases: OneMAX (maximizing the number of 1's in a binary string) and P-Peaks (a hamming-distance related scoring mechanism).

Finding the code lacking in readability and structure, I spent some time refactoring before adding TSP logic. Among other things, I restructured the class hierarchy to provide better structural clarity and more manageable seams between functional units; that is, instead of rewriting code such that the original use cases no longer worked, I preserved and improved existing functionality while allowing extension into the TSP domain.

The primary challenge while restructuring the classes was to present a single Chromosome API as a proxy to different underlying data structures, with the original chromosome storing an array of `byte` values and a new chromosome storing an array of `int` values representing the city IDs. A conflict exists between the method signatures that appear in the two chromosome types, namely for getters and setters. Because `int` and `byte` values are primitives, there is no way to accommodate them both without contaminating the inheritance hierarchy with leaky abstractions, un-implemented methods, odd method signatures, etc.[1] Instead, I chose to implement getters and setters in the subclasses and add cast commands to the objects from which I call them.

3 Implementation

After refactoring, I implemented six new classes:

City: A simple helper class to store coordinates and compute distances between cities.

IntChromosome: A new chromosome type whose alleles are `int` values. `IntChromosome` does not require its alleles to be unique; `Chromosome` classes are data structures with virtually no logic.

IntegerChecklist: This class represents a list of integers which can be checked off one by one. The list is initialized as a range from 0 to n, and at any time the methods `getValuesNotYetCheckedOff()` or `isCheckedOff()` can be called to query the state of the checklist. It is used during the crossover step to make sure that duplicate integers are not introduced into the chromosome being created.

ProblemTSP: This class defines min-value route weight scoring via `resultHasBeenFound()`, `isScore1BetterThanScore2()` and `evaluate()`, chromosome initialization for the TSP algorithm via `getTemplateChromosome()`, and input file reading via `readCitiesFromFile()`.

[1] If the allele values were objects instead of primitives, this could be solved using Java's Generics mechanism – the parent class would be identified as `Chromosome<? extends Allele>` with subclasses `IntChromosome<IntAllele>` and `ByteChromosome<ByteAllele>`. However, this approach is a bit heavyweight. It would incur the cost of class instantiation and access, in my estimation, on the order of hundreds of times more often than when using primitives.

RandomIntSequence: This class simply generates a random-order permutation of a set of integers. It is used to initialize the alleles of new chromosomes, and also during the last part of the crossover step to insert city IDs at the end of the child chromosome in random order.

UniqueIntegersChromosomeStrategy: This class uses the Strategy design pattern to decouple a chromosome from how it is used. It defines how crossover and mutation processes are executed, and also how to create new chromosomes compatible with this strategy through the `getTemplateChromosome()` factory method.

3.1 Chromosome Templates

Each problem depends on a specific type of chromosome (`int`, `byte`, etc.), and a given problem might place further restrictions on how chromosomes are initialized in some way. For example, `ProblemTSP` uses chromosome templates to implement a technique I call the **Zero-City Anchor**. Here is a brief explanation:

There are potentially many different TSP solutions which are isomorphic to each other – in other words, routes that follow the same path but start at different nodes along the path. If two chromosomes representing isomorphic paths were chosen as parents and crossed with each other, their offspring would have a higher weight than either parent, despite the parents being similar. This is because the crossover point would occur at very different geographic locations for the respective parents. Crossing over at this point means introducing a new edge of higher weight than the one that was removed.

I addressed this by introducing a constraint in my algorithm that all chromosomes at all times must have city 0 at position 0. This is the Zero-City Anchor. It serves the purpose of representing all members of an equivalence class (i.e. isomorphic solutions) by a single integer array[2], and to some extent it will also align semi-isomorphic solutions.

By having the `Population` class call `Problem.getTemplateChromosome()` to generate new `Chromosome` objects through the Chromosome Template mechanism, I allow for chromosome creation to be customized for the problem in which it is being used.[3] Furthermore, various problems may have some features in common; there may be other groups of graph traversal problems besides the TSP that require unique-integer chromosomes to be used in different ways. This use case is accommodated by splitting off the `ChromosomeStrategy` class, which also has its own `getTemplateChromosome()` factory method.[4]

This example illustrates that, in the context of chromosome initialization, a `ChromosomeStrategy` is a special case of the default random-value initialization of the `Chromosome` classes, and a `Problem` is a special case of a `ChromosomeStrategy`. This is why `getTemplateChromosome()` shows up in two places in addition to the `Chromosome` constructors.

3.2 Crossover Logic

Crossing over chromosomes whose alleles must remain unique also poses the challenge of handling duplicate alleles between the inherited portions of the first and second parent. Given two parent chromosomes, $C1 = a|b$ and $C2 = c|d$, where $|$ represents the randomly chosen crossover point, the offspring $a|d$ would contain duplicates whenever c is not a permutation of a (and therefore d is not a permutation of b). This will be true in most cases.

So an alternative d' must be created, consisting of the elements $x \in b$, in some order. How these elements are ordered will have a significant impact on the behavior of the algorithm. A few techniques are possible:

Maximize Original Order: In this technique, d' is ordered the same as $C2 = c|d$, with duplicates removed. This is a very simple and efficient solution, and is likely to reflect some characteristic of $C2$ in the child, because as much of its ordering is preserved as possible. The downside to this tech-

[2] Actually, two integer arrays are needed to represent a given isomorphism. The TSP problem, as we have defined it, is non-directed, and therefore any path is equivalent to the same path in reverse. However, using the Zero-City Anchor and array representation, these paths would be considered unique solutions.

[3] By default, this operation defers to the `Chromosome` constructor, or to a `ChromosomeStrategy.getTemplateChromosome()` factory method, but `Problem.getTemplateChromosome()` always retains the ability to do the final step of chromosome creation.

[4] If you are confused at this point, here are the steps of chromosome initialization: the `Population` constructor calls `Problem.getTemplateChromosome()` while creating its initial population of chromosomes, which calls `ChromosomeStrategy.getTemplateChromosome()`, which calls the `Chromosome` constructor.

nique is that, in the Zero-City Anchor approach I am using, it is likely to introduce long edges at the crossover points.

Single-Origin Order: Here, d' will be composed of the elements of the set $x \in d, x \notin a$ in the order they occurred in d, followed by a randomized selection of the remaining elements $x \in b, x \notin d$. The advantage here is that, because all routes start at the "Zero City", all chromosomes with good scores could potentially have similar overall paths. This feature, in my estimation, will give a high probability of creating shorter edges at the crossover point between a and d. This is the strategy I used for this assignment.

Dual-Origin Order: Here, d' will be composed as in the previous technique, but with the exception that, instead of randomizing the remainder elements, they will instead be ordered as they had occurred in b.

Dual Crossover Points: In this solution, two crossover points are selected instead of one. (For this example I will describe two parents $C1 = a|b|c$ and $C2 = d|e|f$.) The offspring is composed of $a|e'|c'$, where e' and c' are defined similarly as above. The effect of this solution is to select **statistically equivalent** numbers of ordered elements from $C1$ and $C2$.

Continuous Merge: This is essentially the Dual Crossover technique taken to its logical limit. Here, there would be no real concept of a crossover point, but instead the chromosomes would be merged together from start to finish, choosing each next allele from one parent or the other and adding it to the offspring. However, I am afraid that it might have a tendency to overwrite or negate the qualities of each parent which had made them strong candidates in the first place, essentially creating random offspring.

Additional strategies orthogonal to those above present themselves: one of these is a "random order reversal": On a randomized basis, the parent alleles could be copied to the offspring from right to left instead of left to right, essentially retaining more of the right side of the chromosome than the left – reversing the problems described above. Instead of retaining the characteristic of parent segment a from parent $C1$, the characteristic of segment d from $C2$ would be retained.

Another strategy would be to implement an adaptive "seeding" mechanism, which would find cities close to each other in the input data and pre-group them somehow, so that during template creation, crossover and mutation, edges within these groups are favored more than edges that span multiple groups. [2]

With the possible exception of Continuous Merge, I believe all these techniques are worth studying further. I have not done research into what kinds of crossover logic other people have tried, so that would be the best starting point.

3.3 Unit Testing

Because of their importance to the overall function of the algorithm, and because they are buried rather deeply in the code and cannot easily report their health, I implemented unit tests for `IntegerChecklistTest`, `RandomIntSequenceTest`, `ProblemTSPTest` and `UniqueIntegersChromosomeStrategy` to make sure they were maintaining their state, creating chromosomes, calculating scores, and doing crossovers and mutations correctly.

4 Performance and Tuning

I did most of my performance testing with ch150.txt, because it is a larger data set. I later verified all results against Berlin52.txt and the results are similar; I have noted wherever the number of cities makes a difference.

4.1 Test Batches

I ran my performance tests in batches so that I could collect results efficiently. Each batch consisted of 100 individual test runs, where a test run is a single execution of the original `for` loop. This loop is what does the work of inheriting and mutating the chromosomes. The original code had this loop running 50.000 times; in my tests I tried ranges from 1.000 up to 500.000.

I ran 18 full batches altogether, plus additional exploratory tests and partial batches. Starting with the fourth batch, I used randomized parameter selection to give evenly distributed inputs without having to guess at which parameter values might be meaningful.

This approach also minimizes outside factors, such as caching or load issues that may come up when running multiple consecutive tests with large populations. For each test batch, I set upper and lower bounds for population, mutation and crossover probability. I changed the code to output results in tabular format, which I copied into a spreadsheet for graphing.

4.2 Parameters

Each batch was run with parameters from the following list:

Probability of Crossover: The probability that a given pair of selected chromosomes will be crossed over. If parents are not crossed over, one parent is returned by the crossover function unchanged.

Mutations Per Chromosome: On average, how many mutations (swaps) will happen per propagation. This is not a deterministic value, but rather gets translated into a per-allele probability by dividing it by the number of alleles per chromosome.

Binary Tournament Rounds: How many levels of recursion are used when selecting a parent chromosome. The original code effectively used a value of zero, meaning no recursion steps were done and a comparison between two randomly-chosen individuals was done. A value of 1 means that two individuals are compared, each of which are chosen by another level of recursion – in effect, a two-round tournament with four contestants. The total number of contestants will always be 2^{n+1}.

Genetic Iterations: How many crossover-and-mutate steps will be performed on the population during a single test run.

Target Score: If this score is reached during a test run, the algorithm will terminate before reaching the number of genetic iterations specified above. I set this to zero for my tests because I wanted to find the best score possible.

Population Size: How many individuals are selected from by the algorithm for each genetic iteration.

4.3 Test Results

Due to limited space, I will only provide commentary on some representative results.

Batches 1-3 used a fixed population size of 512, and gave essentially random results. I tried different population sizes and found a distinct response in performance (especially good below ~20), so I decided to randomize the population size for batches 4-18, generally testing ranges between 2 and 50, plus a few validation tests between 8 and 512. I was initially surprised to find that quality decreases with a larger sample population, but this does make sense: it is undoubtedly because smaller populations increase the chance that a given chromosome will be chosen more than once. That is, during the selection/crossover step, the bigger the population is, the more likely it is that genes will be selected for the first time, and therefore offspring are being created primarily between parents with random chromosomes.

I also found a relationship between population size and number of genetic iterations. When more genetic iterations are used, the population sizes that perform well become larger on average. In batch 9 and 10, I examined this relationship by performing the same tests with 500.000 genetic iterations and 10.000 iterations, respectively (Fig. 1). In addition to favoring larger populations, tests with more iterations found better routes – roughly half as long using 500.000 iterations as with 10.000. Processing time scales linearly with number of genetic iterations, so this equates to a 50-fold increase in processing time while only doubling the quality of the result.

It is true that, depending on context, the value of these two resources may be completely different and this trade-off is acceptable, but I wished to decouple this relationship and achieve the higher quality results, leveraging the resources of a larger population independently of other variables, using fewer genetic iterations and therefore less processing time.

For this strategy to work, I needed a mechanism by which to choose more intelligently which chromosomes will be paired for crossover. I tried using a series of binary tournaments, rather than a single one. Instead of two pairs of random selections being compared against their partner, I ran a four-round tournament among eight pairs, resulting in two champions. Anticipating that this might eventually scale to a 2^n solution, I parameterized the new method I created for

this step using the Binary Tournament Rounds parameter mentioned above. To test this new algorithm, I kept all values the same as before, including the number of iterations at a relatively small 50.000.

Surprisingly, adding recursion to the binary tournament does not improve results, as batches 11-15 show (Fig. 2, 3 and 4), and they do add computational cost (on the order of 200ms for four rounds and 800ms for six rounds). Further work in this area could include designing a more intelligent multi-round selection mechanism, but results were so poor that I consider this a very low priority.

Finally, there is a correlation between the number of cities in the problem and the population size that is effective in solving the problem. Solving a problem with fewer cities causes the algorithm to behave as if it is doing more genetic iterations, in that the range of population sizes that are effective in finding good solutions broadens (Compare fig. 5 #1 with 3 #1). I believe this is because having fewer cities corresponds to having more of a homogeneous solution space (because there are fewer possible solutions); therefore any randomly selected solution will be closer to ideal than it would be if there were more cities. Parent chromosomes chosen from a population with this characteristic will have a higher likelihood of producing successful offspring.

5 Conclusion and Future Work

Certainly, the main variable that determines the quality of results is the number of genetic iterations. Typical route weights for ch150.txt were around 22.000 for 5.000 iterations; 19.000 for 10.000 iterations; 13.500 for 50.000 iterations, 11.000 for 100.000 iterations, and 10.700 for 500.000 iterations. There is an obvious correlation here, with significantly decreasing rates of return.

As I ran the tests, I found a definitive "sweet spot" with these parameter values:

Population: 4-17 for low numbers of genetic iterations or recursion steps, or for large numbers of cities. The upper range for this goes up to around 35-45 for the opposite characteristics.

Mutations per Chromosome: 0.6-1.4, expanding to 0.8-2.2 for high numbers of iterations or recursion steps, or low numbers of cities

Crossover Probability: I found very little statistical significance here. Generally, the range 0.5-1.0 was good, but no trend emerged beyond that. It did not seem to be correlated to any other variables.

In my opinion, future work should be focused on crossover logic and chromosome initialization, as described in the Crossover Logic section. It is also possible that some more intelligent chromosome selection logic could be created, but as I had mentioned, this does not look likely to produce good results.

6 Appendix: Result Graphs

These graphs show performance (size of the bubbles) as a function of population size (X-axis) and both mutation and crossover rates (Y-axis).

References

[1] J. Cabello Galisteo. ssga: Steady state ga. http://neo.lcc.uma.es/software/ssga/, 2008.

[2] Michael LaLena. Traveling salesman problem using genetic algorithms. `http://www.lalena.com/AI/Tsp/`, 1996-2011.

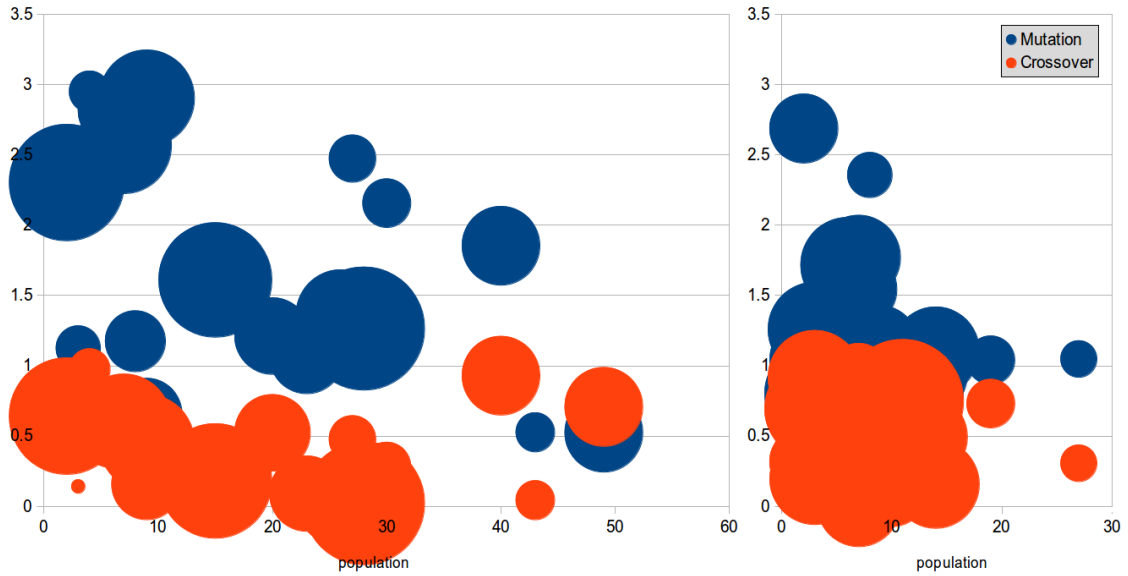

Figure 1: ch150.txt (150 cities), batch 9 (500,000 iterations) and 10 (10,000 iterations)

Batch	Iterations	Population	Recursion Steps	Time (best 10, in ms)	Average Weight (best 10)
11	50,000	2-50	0	683	14,309
12	50,000	8-50	4	847	13,435
13	50,000	8-512	4	898	14,629
14	50,000	16-512	6	1,268	13,801
15	50,000	20-50	6	1,934	13,648

Figure 2: Averages of the 10 best results for batches 11-15 (lower weight = better score)

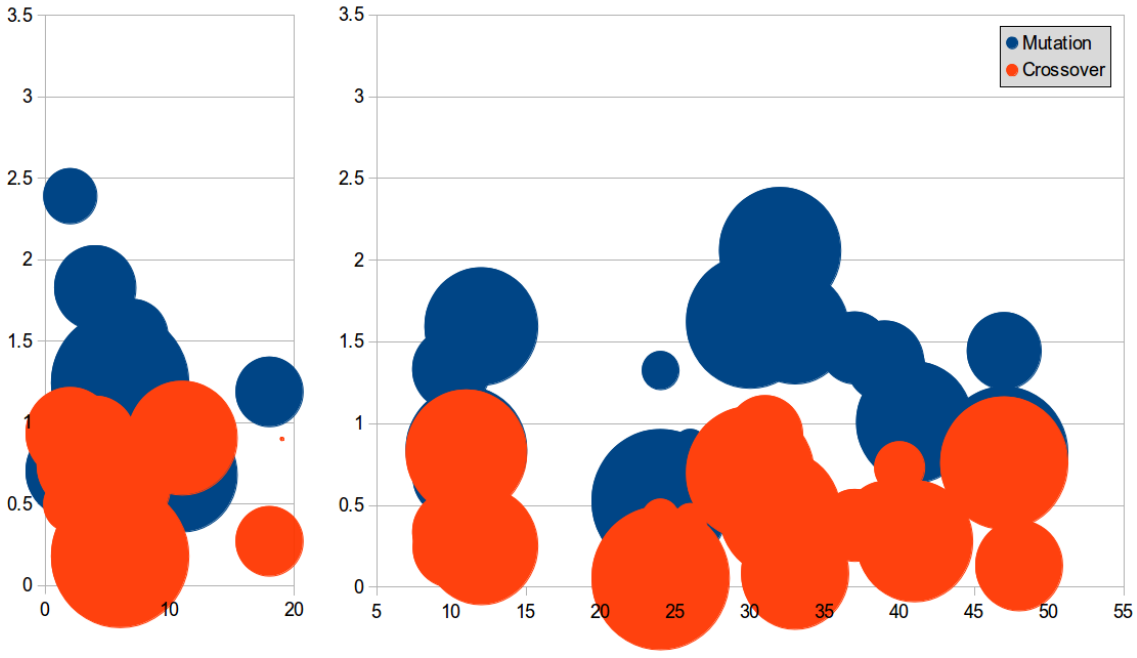

Figure 3: ch150.txt (150 cities), batch 11 (no recursion) and 12 (four recursion steps)

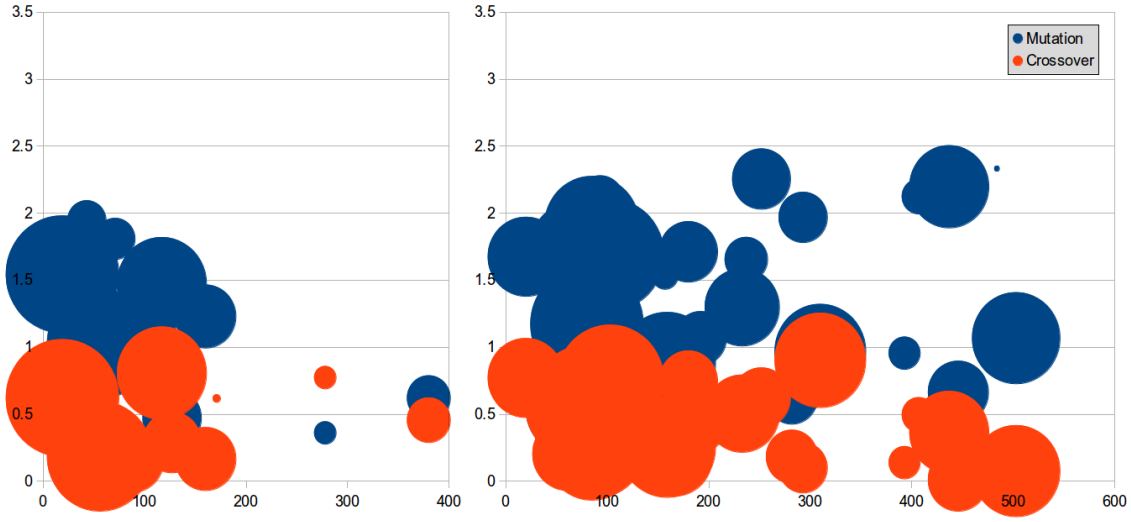

Figure 4: ch150.txt (150 cities), batch 13 (four recursion steps) and 14 (six recursion steps)

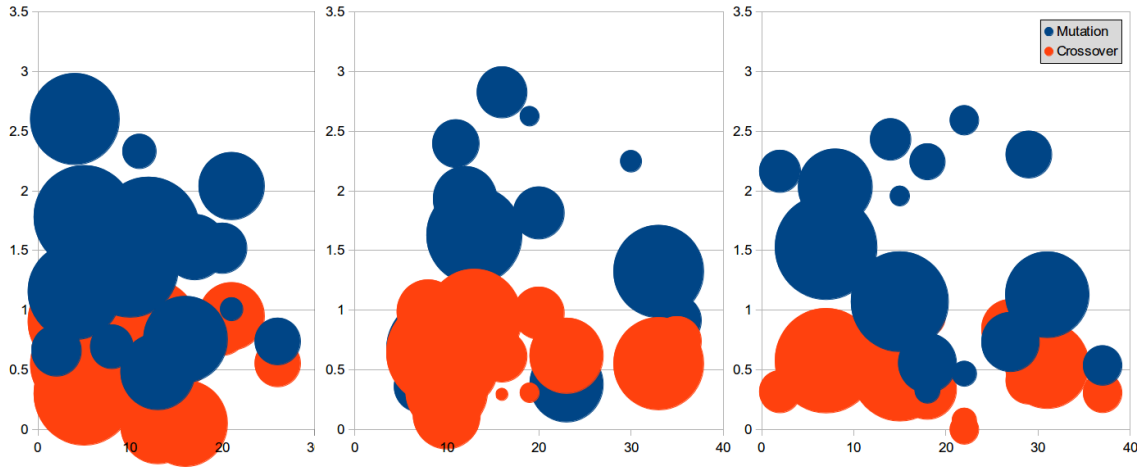

Figure 5: Berlin52.txt (52 cities), runs of 1.000, 3.162 and 10.000 iterations

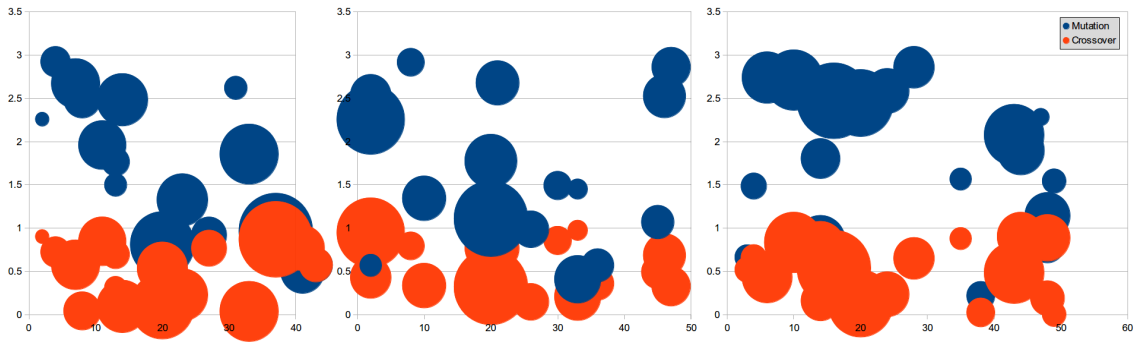

Figure 6: Berlin52.txt (52 cities), runs of 31.620, 100.000 and 316.200 iterations