Using Genetic Algorithms for Solving Hard Problems in GIS

Steven van Dijk   Dirk Thierens   Mark de Berg

Abstract

Genetic Algorithms (GA's) are powerful combinatorial optimizers that are able to find close to optimal solutions for difficult problems by applying the paradigm of evolution with natural selection. We describe a framework for GA's capable of solving certain optimization problems encountered in Geographical Information Systems (GIS's). The framework is especially suited for geographical problems since it is able to exploit their geometrical structure with a novel operator called the Geometrically Local Optimizer. Three such problems are presented as case studies: map labeling, generalization while preserving structure and line simplification. Experiments show that the GA's give good results and are flexible as well.

1 Introduction

Geographic Information Systems (GIS's for short) combine a geographical database with powerful data extraction methods and visual presentation to perform complex analysis on geographical data. However, not all tasks one would like a GIS to perform are easily automated. Label placement and certain generalization tasks are examples of this. Often these tasks are optimization problems, where a close to optimal point has to be found in a vast, complex search space. In map labeling for instance each label can be placed in several candidate positions. From all possible combinations of label placements for all features, the optimal labeling with as few intersecting labels as possible has to be found. Even in their simplest form these optimization problems are often NP-hard (the map labeling problem was proven to be NP-complete by Marks and Shieber [12]), which means that trying to find fast (polynomial time) algorithms that guarantee to find the optimal solution is an infeasible approach. In practice there are usually additional constraints that make the problem even harder. Therefore, in order to find good solutions in a reasonable amount of time, heuristic methods have to be used.

Genetic Algorithms (GA's) are combinatorial problem solvers that are capable of finding close to optimal solutions without having to perform an exhaustive search. They have proven to be successful in industry [3] in such varied fields as medicine (optimizing radio-therapy treatment), bio-informatics (molecular design) and manufacturing (optimizing car suspension design). GA's can also be used in a GIS to tackle hard problems. However, we need to be careful since the environment of a GIS poses some additional demands.

Firstly, in a GIS problems often consist of many aspects. For example, the combinatorial aspect of a map-annotation problem makes it hard, but additional aspects may need to be considered too to satisfy aesthetical or presentation demands. Therefore, it is important that additional constraints can be included in the problem definition. GA's are known to be open to adding constraints by incorporating them in the cost function. This method becomes impractical, however, when the number of extra constraints increases beyond just a few. Our method offers a way of including additional constraints by enforcing them on a local scale.

Secondly, to be practical, a GA developed to solve a certain problem should integrate well with the GIS and be user-friendly. In other words, if the GA requires excessive tuning or the setting of many parameters, it is limited in its use.

We propose a framework for developing GA's that are robust in the sense that they can be extended with additional constraints easily, do not require a lot of tuning, and use a minimal amount of parameters. As such, we offer a pragmatic approach to solve hard GIS-problems.

Fortunately, the design of a genetic algorithm for a GIS-problem is eased by the geometrical nature of the problem. The structure of the problem (expressed in so-called *linkage*) is often quite clear and the GA is able to exploit this information. To this end we introduce an operator called the *geometrically local optimizer* which allows for an efficient GA. The resulting GA shows good performance, and we demonstrate this in a number of case studies: map labeling, a generalization task, and line simplification.

The GA for the map labeling problem demonstrates the general technique. Its performance was favorably compared against other algorithms in other papers [19, 20, 22]. The GA has the additional advantage that it is designed to be easily extended to more realistic map-labeling problems.

The second case study concerns a generalization task where a minimal subset of points has to represent a large set of points. It exemplifies why it is better to use a geometrically local optimizer instead of constructing a complex cost function. The GA outperforms two greedy algorithms.

The final case study deals with line simplification and shows how to avoid infeasible solutions. An extra criterion (avoiding "spikes") was added to the basic problem as an example of the flexibility of our approach. The GA outperforms the well-known algorithm of Douglas and Peucker [2], but also the exact algorithm of Imai and Iri [10], which both needed post-processing to deal with the extra criterion.

This article is structured as follows. We start in Section 2 by explaining what genetic algorithms are. First, we give a tour around the standard components using the example of a naive GA for map labeling. Then we proceed with a bit of GA theory (in Subsection 2.2) that will show how we can improve on the naive approach. In Section 3 we will present the first case study that illustrates our approach, namely the problem of point-feature map labeling. This will show different choices for the design decisions which lead to better performance than the naive approach. Section 4 provides a more general framework for designing GA's for GIS-problems. In Section 5 and Section 6 we present two more applications of our framework: generalizing a set of points while preserving structure and line simplification without creating "spikes". We implemented these GA's and show they solve the problems satisfactorily. Section 7 is reserved for discussion about the feasibility of using GA's in a GIS. We conclude in Section 8.

2 Genetic Algorithms

The Genetic Algorithm is a heuristic solver for optimization problems. Its inspiration comes from the Darwinian theory of evolution by means of natural selection. In biology, organisms compete for resources in order to survive. For example, consider a number of trees competing for sunlight. A tree which grows taller than other trees can expose its leaves to more sunlight and will be in a better position to reproduce itself. This tree is said to be *fitter* than other trees. Assuming environmental conditions are equal for all the trees, the relative height of a tree is determined by its genetic blueprint: its DNA. Since fitter (taller) trees reproduce more, this genetically determined trait of growing high is propagated through the population of trees. In a few generations, a tree will, on average, be taller than its ancestors. This process is called natural selection. An essential ingredient of evolution is the introduction of new traits. This happens when DNA is copied and a mutation (a random change) occurs. If the mutation is beneficial it will propagate through the population. Traits can also be combined in a single individual by swapping pieces of DNA ("crossover") in the process of reproduction.

Even though this view of natural evolution is very simplified, it serves as an introduction to the *artificial evolution* of a GA. In a GA, not organisms, but solutions for a certain problem evolve. The problem is formulated using a *cost function* that attributes a cost (to be maximized or minimized) to every setting of its problem variables. In artificial evolution we use the cost function to specify fitness and call it the *fitness function.* For example, a mathematical expression could be used as a cost function in which case the GA would be a function optimizer. A "solution" need not be optimal, it just gives a setting for the problem variables. It is possible that the problem specifies constraints on the settings the problem variables may have. If the solution fulfills these constraints, it is called *feasible.* It is usually stored as a string of values. Following the analogy of biology, we call such a string a *chromosome* (a strand of DNA) consisting of a number of *genes.* Each gene stores one from a fixed number of values called *alleles.* The fitness of such a solution is a measure of how well it solves the problem. It is given explicitly by the fitness function.

A GA evolves the population of solutions by repeated selection and mating. Selection picks the individuals for which the fitness function yields a relative high result. They are "mated" by combining their genes using the operators crossover and mutation. Crossover distributes the genes of the parents over the children. Mutation randomly chooses a gene on an individual and stores a random allele there. After all children are generated, they replace the parents.

The quality of the solutions in the population will progressively get better with subsequent iterations. Eventually, the population will have *converged*: as a result of selection all individuals in the populations are similar and hardly any progress is made. If the GA is designed correctly, the best solution should be near optimal.

The algorithm for the standard GA can be seen in Algorithm 1. This algorithm uses the following subfunctions:

Initialization: start with a population of feasible solutions that cover the search space of the problem well.

Algorithm 1 GENETIC ALGORITHM

```
1: initialize population of solutions
2: **repeat**
3:     select parents from population
4:     with probability $P_c$ perform crossover and generate children
5:     with probability $P_m$ perform mutation on children
6:     replace members of population with children
7: **until** termination criterion satisfied
8: **return** best individual
```

Selection: select the individuals that are going to reproduce, based on their fitness.

Crossover: make children by swapping genes of the parents to combine their characteristics.

Mutation: change the children in a random way to introduce diversity.

Replacement: replace individuals from the old population with the children to make a new population.

Termination criterion: evaluate the state of the population (for example to measure the rate of improvement) and decide whether to stop the algorithm.

We will explain the algorithm (and its subroutines) in more detail in the next subsection where we discuss the various design decisions by using the example of a GA for labeling a map.

2.1 An example GA

First we will define the problem which we will try to solve in this section:

> **The map-labeling problem**
>
> Given are N points in the plane, for which a label can be placed in one of the positions top-right, top-left, bottom-right or bottom-left. Find a labeling (an assignment of a position to each label) such that the number of labels that do not intersect other labels is maximized.

When designing a genetic algorithm to solve this problem, we will encounter several design decisions. The choices greatly effect the performance of the GA. For now we will stick to the "standard choices" like those described in the textbook by Goldberg [5]. In the next sections we will show that more insight in the dynamics of the GA allows us to make better choices, and that these decisions depend primarily on the geometric structure of the problem.

Encoding. The GA starts by generating a population of solutions for the problem. Therefore, the first decision that has to be made when designing a GA is how one can *code* or represent the solution of a problem in an effective manner. A solution for the map-labeling problem is a map with its labels placed

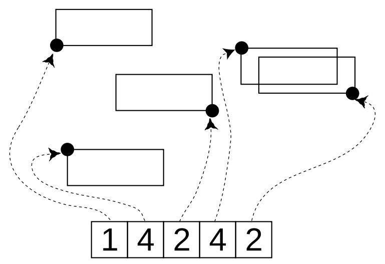

Figure 1: A map with five cities and the chromosome representing it. Each element of the chromosome is a gene corresponding with a city. The value stored in a gene is an allele corresponding with a label position. Possible alleles for a gene are 1 (top-right), 2 (top-left), 3 (bottom-left) and 4 (bottom-right).

in certain positions. We need to code this as a chromosome (a sequence of genes). Usually this is straightforward, but it is important to realize that there is an interdependency between the representation and the operators that change it (see Section 2.2). We will postpone such considerations for now, and choose the representation that suggests itself almost immediately: a chromosome with each gene corresponding to a city and its allele to the placement of the city label. See Figure 1 for a picture.

Fitness function. Now that we have the representation, we need a way of determining how well the solution solves the problem. In the case of the map-labeling problem the chromosome represents a labeling of the map, and the measure of quality (or *fitness*) will be the number of labels which do not intersect other labels. In Figure 1 for example, the map has a fitness of three. It can be improved to five (the optimum) by placing the label of the last city in the bottom-left position. The *fitness function,* which returns a fitness for each chromosome, should be fast since it will be applied often. It should also be kept as simple as possible, i.e. it should not contain a lot of parameters which have to be tuned later.

Initialization. Now the GA can start with constructing the first population and initialize it. The *initializer* should make solutions that cover the whole search space well. For example, it can choose at random an allele for every gene on every chromosome, producing a random labeling. The size of the population (the number of chromosomes it contains) is denoted with n. The question of how to choose n will be discussed later (in Section 7).

The population is now ready to evolve. This means that parents have to be *selected* to mate, children have to be produced and the parents have to be *replaced*, possibly by their children. This produces a new population and a new iteration can take place.

Selection. Selection can be done in a lot of ways, each with its own advantages and disadvantages. The aim of selection is to give a bias to solutions with high fitness, in order to promote good solutions. The assumption is that, like in natural evolution, fit parents produce fit children. A straightforward manner of selecting parents is by randomly choosing two and picking the one with

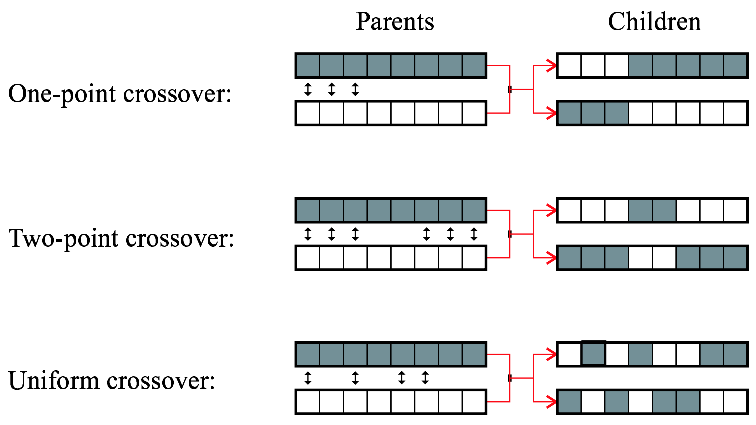

Figure 2: Recombination operators.

the highest fitness. This is called *tournament selection,* with a tournament of two. The GA keeps holding tournaments until n parents are chosen, which will produce n children.

Another selection method is *roulette wheel selection*, which selects parents one at a time, with probability in proportion to their relative fitness. This selection method has the disadvantage that the selective pressure decreases after many iterations, and we will use tournament selection for our GA.

Reproduction. For every pair of parents, two children are produced. This can be done by either mating or cloning. Mating occurs with probability P_c which is usually around 0.7. It performs *crossover* (see below) on the two parents and swaps genes. This produces two children. Cloning just makes identical copies of the parents. The children can be altered further by mutation, which picks at random a gene and chooses randomly another allele for it. Mutation is also governed by chance, with probability typically around $\frac{1}{N}$ for each gene.

Crossover. How should recombination (crossover) be done with our problem? Examples of crossover operators which are often used are one-point crossover, two-point crossover, and uniform crossover. See Figure 2 for visual examples. One-point crossover takes all genes until a randomly chosen point from one parent and takes the other genes from the other parent, creating a new child. The other child is created in a complementary fashion. Two-point crossover swaps the string of genes between two randomly chosen points. Uniform crossover decides for each gene separately (with probability 0.5) whether to swap or not. We will use one-point crossover, the standard choice, in our GA.

Replacement. The next step in the GA is *replacement.* How do we put the children in the original population? Since the population size is fixed, we are forced to kill some individuals to make room. Do we replace the whole population with all children? Or do we replace some of the worst individuals with some of the best children? Replacing the whole population is called a *generational* replacement scheme, and it is the standard choice.

Termination. The rate of improvement will gradually drop while the population converges (individuals become more similar). Therefore, one can stop the GA when little or no improvement is made. Another criterion could be to stop when a solution is found which is deemed good enough, instead of waiting for convergence.

The termination criterion we will use for our GA is a comparison of the average fitness in the population with the fitness of the best individual. When they are equal, the run is ended. The run is also ended when it continues beyond a certain limit of computational expense (see below).

Results.

Naturally we are interested in the performance of the resulting algorithm. In Figure 3 a graph is shown comparing various versions of the GA. The map used was randomly generated by placing 1000 cities on a square grid with a side length of 616. To remove edge effects the map was folded into a torus. The cities had labels with dimensions of 30 by 7 units. The cities were placed on the map with their labels and care was taken that no labels would intersect, thus creating a map of which a totally intersection-free labeling was known. Obviously, this optimal labeling was not used by the algorithm. The figure shows the number of label-intersection tests as a measure of the amount of computation. The label-intersection test was the most atomic action of the algorithm. Practically all the time the GA was running was spent doing label-intersection tests. The limit of computational expense used in the termination criterion was $40 \cdot 10^7$ label-intersection tests. Each graph in the figure shows the average of five runs on five different maps each (25 runs in total), in order to keep statistical significance.

The algorithm outlined in this section is denoted as "Tournament with 1pt crossover". To illustrate the effect of making different choices for the selection scheme and crossover, two other GA's are also shown. The first (titled "Tournament with uniform crossover") differs in the crossover used, and is slightly inferior to the GA using one-point crossover. The second (titled "Roulette with 1pt crossover") differs in the selection scheme, which is clearly inferior to the GA using tournament selection. The GA using the "standard choices" shows good performance compared to the other GA's, but improvements can still be made in the selection scheme and the recombination operator. But in order to understand the nature of these improvements, we first need to treat some background theory.

2.2 Why genetic algorithms work

In the previous subsection we illustrated the components of a GA and gave little justification. However, a fair amount of GA theory exists to guide us with the design decisions. One of the key insights of the workings of genetic algorithms is the *Building Block Hypothesis* (BBH) [9]. The BBH states that a (close to) optimal solution can be seen as a combination of smaller, more fundamental units (building blocks) which are unlikely to be split up (*disrupted)* during crossover and have a high fitness contribution. Therefore, chromosomes containing them will be selected more often and their proportions will grow.

In a more abstract view, a building block is a combination of genes which are set to specific values satisfying two conditions: 1) the genes when set to these

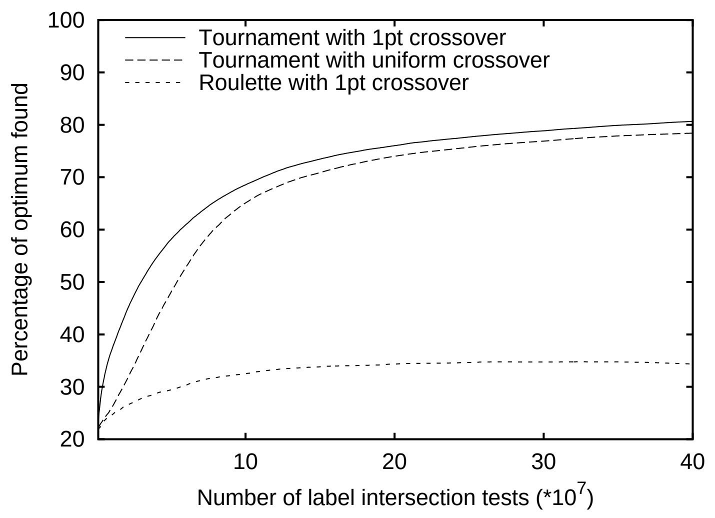

Figure 3: Comparison of several genetic algorithms: a GA using tournament selection and one-point crossover, one using roulette wheel selection and uniform crossover, and a GA using tournament selection and uniform crossover.

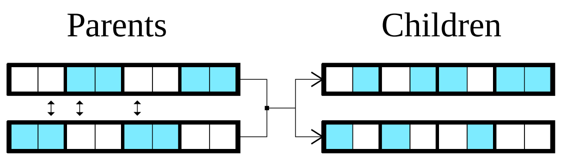

Figure 4: A disrupting crossover. Building blocks consist of two genes and are drawn shaded.

values give a high fitness contribution and 2) the genes are often transfered together during crossover.

Which genes belong "together" is determined by the *linkage*. It follows that if one knows the linkage of a problem, a crossover operator can be devised which keeps genes together. Unfortunately, the linkage of a problem is rarely known. This is why the one-point, two-point and uniform crossover operators exist: they make different assumptions about the linkage of the problem. For example the one-point crossover assumes that genes that are near each other on the chromosome belong together. This is true in biology (genes that are in a sequence often code for the same protein) and in function optimization (where each variable is coded as a sequence of bits). These representations work well with this crossover. This is however not true for a representation for a problem which is essentially two dimensional, such as map labeling. The same holds for two-point crossover. The assumption that is made with uniform crossover is that the linkage between all pairs of genes is equal, so no special bias should be used to preserve certain combinations of genes (like one-point crossover does with sequences). It is therefore vital to know the linkage of the building blocks, because if crossover splits a building block over two children, it is disrupted (see Figure 4).

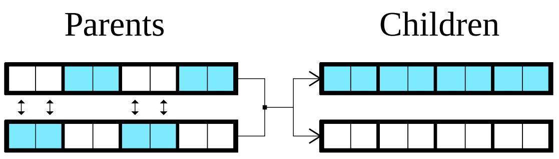

Figure 5: A perfect crossover. Building blocks consist of two genes and are drawn shaded.

Of course, we would not use any crossover operator if they only disrupted chromosomes. They do something useful as well: they *mix* the building blocks [17]. The required solution is made of these building blocks. Just having all the building blocks in the population is not good enough: they have to be brought together on one chromosome. This is the main task of the crossover operator. Ideally, if the first parent contains about half the number of building blocks needed to make an optimal solution, and the other the rest, crossover would produce a child containing all the building blocks (this situation is depicted in Figure 5).

Since GIS-problems often deal with geographical data, their linkage is likely to be determined by their geometry. The map-labeling problem illustrates this. Which genes depend on each other? These are the genes corresponding with cities that are close to each other, since their labels can intersect. Using this knowledge about the linkage of the building blocks, we can construct a crossover operator that is less disruptive than one-point crossover and mixes well. This is essential to build a GA that finds good solutions and scales up well [16, 22]. Scale-up behavior describes what happens with the amount of required computation when the input size changes.

We will apply these ideas in the next section, where we construct an efficient algorithm for the map-labeling problem.

3 A better GA for map labeling

We will use the map-labeling problem (see Section 2.1) to illustrate our general framework for GA's for geographical problems, which we will outline more formally in the next section. In subsequent sections we will then apply the framework to two other problems: generalization while preserving structure (showing how to deal with a constrained search space) and a line simplification problem (showing the flexibility and extendibility of the framework).

Encoding, fitness function and initialization. We keep the same encoding for the map-labeling problem as before: a chromosome with a gene for each city and an allele (a value between 1 and 4) for each label position. This will cover the search space well. We keep our fitness function as simple as possible, by only counting the number of non-intersecting labels. Note that additional constraints (such as preferences for positions) will not be specified in the fitness function, but in the geometrically local optimizer (described later). Since every labeling can be considered a valid solution, we can just initialize each individual by generating a random labeling.

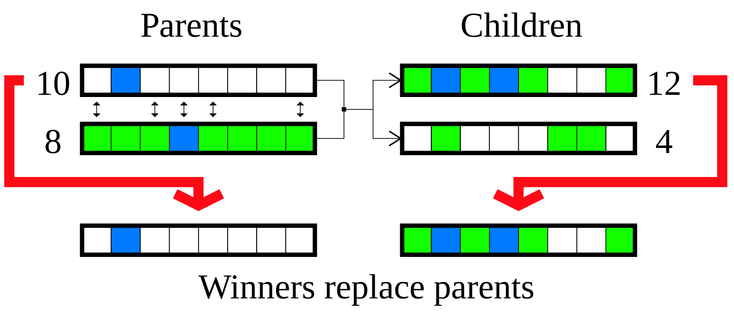

Figure 6: The elitist recombination scheme. Building blocks consist of one gene and are drawn dark shaded.

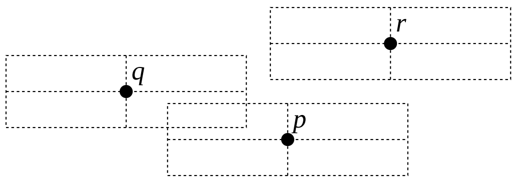

Figure 7: Cities p and q are rivals, but p and r are not.

Selection and replacement. Selection and replacement are combined using the *Elitist Recombination Scheme* [18]: choose two parents at random, generate two children and from this family of four let the two fittest individuals replace the parents in the population. See Figure 6 for an example. The selective pressure results from the biased replacement. This update scheme has the advantages that it is conceptually simple and preserves good solutions. Note that we can also safely set the probability of crossover (P_c) to 1 since disrupted children will never replace their parents.

Crossover. We already discussed (in Section 2.2) that one-point and two-point crossover are unsuitable for a two-dimensional problem and that uniform crossover disregards the linkage of the problem (making it too disruptive).

Since we have a good idea of the linkage of the building blocks, we can devise a better crossover that allows for good mixing but is not too disruptive.

Intuitively, cities that are close together depend on each other for the placement of their label, since a neighboring city can place its label such that some candidate positions are no longer possible. The genes for these cities therefore have linkage. To make this more formal, we define two cities to be *rivals* if their labels *can* intersect. See Figure 7 for an example. We now assume the genes for a city and its rivals can contain a building block. To get good building block mixing, we want to transfer roughly half the number of genes from each parent to each child (and the remaining genes from the other parent). Therefore we perform crossover as follows. We randomly pick a point on the map. We mark it and its rivals and go on picking the next point. When the number of marked points first exceeds half the total number of points, we stop picking new points. The first child is created by copying the labelings of the marked points from the first parent and copying the labelings of all the other points from the

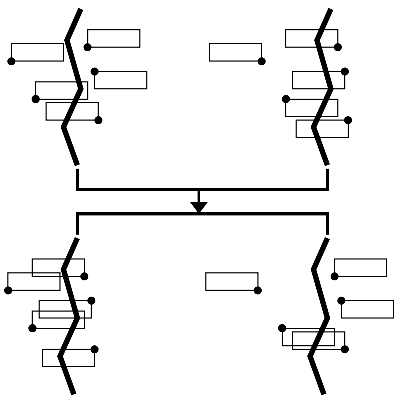
Figure 8: New conflicts can arise as a result of crossover.

second parent. The second child copies marked points from the second parent and the remaining points from the first parent. We call this a *linkage respecting crossover*, since it takes care to disrupt the linkage of building blocks as little as possible.

Our definition of a building block (a labeling for a city and its rivals) implies that building blocks overlap. As a result, after crossover, some building blocks that were not chosen to be marked can be disrupted. Figure 8 shows that this causes new conflicts where a point was marked but its rival was not, and their labels become intersected. We will try to repair these conflicts using a new operator: the geometrically local optimizer.

Geometrically local optimizer. We will not use mutation in the traditional sense ($P_m = 0$). Mutation blindly changes strings, often disrupting them, in the hope that a building block is created by chance. This may be necessary in a GA for other problems, but in this case there is a more efficient method. Since the linkage of the building blocks is geometrically determined, we can explicitly try to generate new building blocks.

On points where new conflicts have arisen we apply the *geometrically local optimizer*. It uses a procedure called *slot filling* to improve on the local labeling of a city and resolve conflicts.

Slot filling views all possible label positions as slots, which can be filled (when another label is intersecting it) or empty. After determining the status of all slots for the given point, the label is moved to a randomly picked slot which is empty. See Figure 9.

Slot filling is a simple procedure. However, the technique can also be applied when building blocks have a more complicated structure. For example, if a label and its city should not be separated by a river, the geometrically local optimizer can enforce that. The alternative would be to place an additional subfunction in the fitness function that penalizes each violation of the rule. This requires tuning of the penalty, which is computationally expensive. Also a new tuning phase is needed every time a different kind of map is used or a new rule is added.

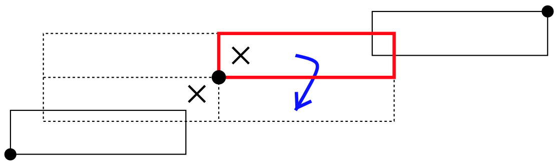

Figure 9: Slot filling places the label in an empty "slot".

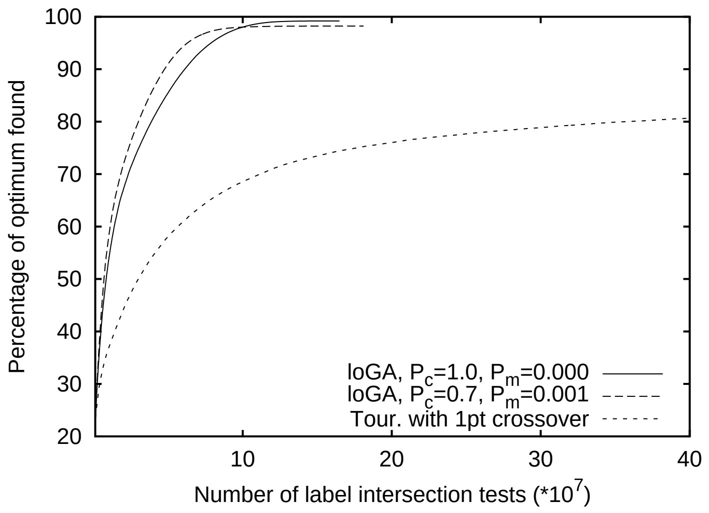

Figure 10: A comparison between the previous GA and the improved GA using Elitist Recombination, rival crossover and Geometrically Local Optimizers (denoted "loGA").

Results and comparison to other approaches.

How well does the resulting GA perform? As can be seen from Figure 10, it is a significant improvement over the conventional approach from the previous section. A close to optimal solution is found. For the sake of comparison the figure also shows a graph of the new GA with $P_c = 0.7$ and $P_m = 0.001$. Again, each graph in the figure shows the average of five runs on five different maps each (25 runs in total).

Several other algorithms to solve the map-labeling problem have been devised. The branch-and-cut technique of Verweij and Aardal [24] is an exact solver capable of solving instances up to 950 cities. However, the scale-up behavior is exponential, meaning that there is a bound on the problem size which can realistically be solved. In contrast, the scale-up behavior of the GA described was modeled in another paper [22] and experiments confirmed it is linear. Verweij and Aardal claim their algorithm could also be used as a heuristic since it finds a good solution quickly and most of the time is spent enumerating the whole search space.

Other genetic algorithms have been proposed by Verner *et al.* [23] and Raidl [13]. Christensen *et al.* [1] compared a number of map-labeling algorithms with the Simulated Annealing (SA) algorithm they devised. Figure 11

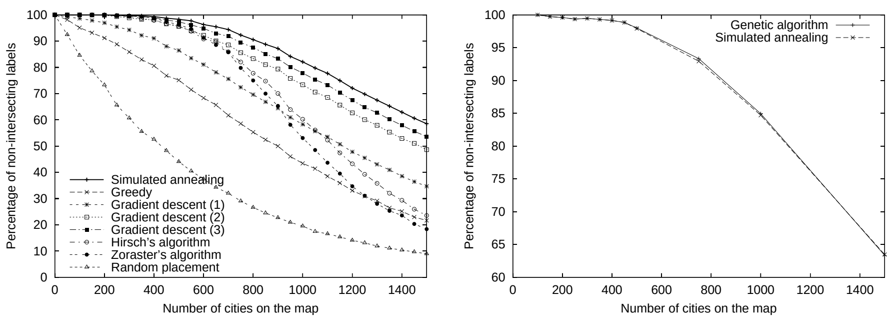

Figure 11: Left: a comparison of different map-labeling algorithms by Christensen *et al.* [1]. Right: a comparison of the GA with Simulated Annealing.

(left) shows their results, with SA producing the best results. We compared our GA with an implementation of SA [19] (Figure 11; right). The GA produces almost exactly the same results as SA.

These algorithms all reported good performance on the basic problem of map labeling, which is concerned with the placement of labels for point features on an otherwise empty map. One would naturally like to extend the problem definition to cope with more realistic maps that have to be labeled. Adding more constraints can be done in the GA described earlier by considering them in the geometrically local optimizer. This cleanly separates the combinatorial difficult part (searching for a placement with the maximum number of free labels) from the other, softer constraints (such as placing a label in water if the city is a coast town). On the other hand, the branch-and-cut algorithm of Verweij and Aardal, the GA's of Verner *et al.* and Raidl, and the Simulated Annealing algorithm of Christensen *et al.* all require that the problem be formulated as a weighted variant of the label placement problem. The optimization is then done with a cost function that has weights to specify the desirability of label placements. This is much less practical or flexible than the approach using geometrically local optimizers, where there is no need to set weights.

An extended example.

Figure 12 illustrates that the described GA is capable of handling more extended problem definitions. The map shows an example of a map that was labeled using multiple constraints, which were handled by the geometrically local optimizer. The other parts of the GA were not changed. The input set of cities consisted of 2380 cities, of which 394 had a population larger than 50,000 ("large" cities) and six cities had a population larger than 1,000,000 ("mega" cities). The following requirements had to be fulfilled by the solution:

1. When possible, a label of a city should be placed in a preferred position, top-right being the most preferred.

2. If the map is too crowded, an appropriate label should be deleted.

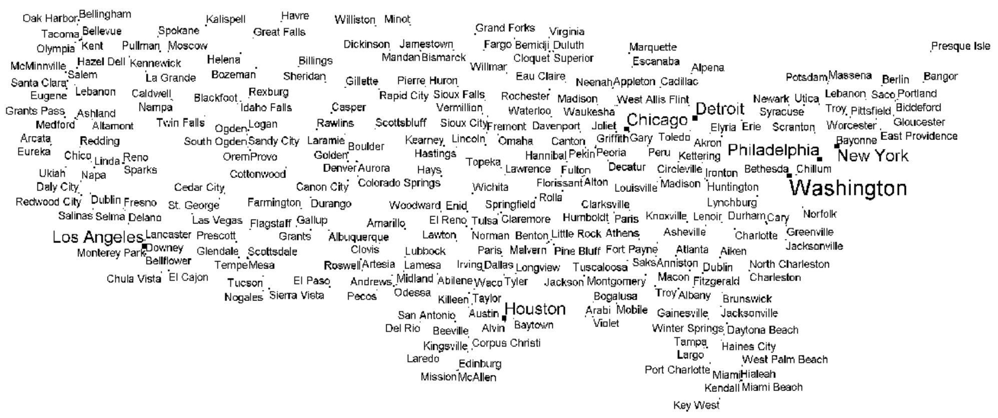

Figure 12: A labeling for the cities of the USA.

3. Labels of a "mega" city should always be placed. Also the label of the city "Washington" should always be placed.
4. The label of a "large" city should not be deleted in favor of the label of a normal city.
5. The label of a city on the coast should be placed in the water.

The last requirement was exemplified with the city of Los Angeles only, since the input data did not contain information about the surroundings of a city.

The resulting labeling can be seen in Figure 12, where "mega" cities and Washington have larger fonts and "large" cities are printed in a darker shade than normal cities. Details on the implementation of the GA that produced this map can be found elsewhere [19, 21].

4 A general framework for solving GIS-problems

Now that we have seen a concrete example of our approach (which is to use a linkage respecting crossover and a geometrically local optimizer), we will formalize the ideas in a general framework. The next two case studies will then be described in this framework.

The general algorithm takes as input two parameters specifying the length of a chromosome and population size. The algorithm proceeds from one population to the next, for as many iterations as are needed to satisfy the termination criterion. To start the search, the first population needs to be initialized to make sure it contains feasible solutions and covers the search space well. As the search continues, the search points in the population will narrow down on a specific region of the search space that contains highly fit solutions and hopefully contains the optimal solution.

Each iteration the Elitist Recombination Scheme is performed: two parents are chosen randomly, two children are generated by crossover and subsequently altered by the geometrically local optimizer, and of this family of four the best two replace the parents. Crossover is performed by first constructing a crossover mask, which specifies which genes have to be copied from which father to which child. Linked genes are placed together in the mask to make the crossover linkage respecting. After crossover the geometrically local optimizer is applied to the children to make them feasible solutions again, and possibly to improve their fitness.

Algorithm 2 GIS-GA

```
          $N$ = number of genes in a chromosome
          $n$ = number of chromosomes in the population
 1: generate $Pop$ with $n$ chromosomes                          }
 2: for $ind \in Pop$ do                                          }  Initialization
 3:    INITIALIZE($ind$)                                          }
 4: repeat
 5:    choose at random the parents $p_1, p_2 \in Pop$            }  ER Scheme - step 1
 6:    $M \leftarrow \emptyset$                                   }
 7:    while $|M| \leq \frac{1}{2}N$ do                           }
 8:       choose at random a gene $g$                             }
 9:       $M \leftarrow g$                                        }
10:       for all $g'$ such that LINKED($p_1, g, g'$) do          }
11:          $M \leftarrow g'$                                    }  Crossover
12:    $P \leftarrow$ complement of $M$                           }
13:    for $i \in M$ do                                           }
14:       $c_1[i] \leftarrow p_1[i]$ ; $c_2[i] \leftarrow p_2[i]$ }
15:    for $i \in P$ do                                           }
16:       $c_1[i] \leftarrow p_2[i]$ ; $c_2[i] \leftarrow p_1[i]$ }
17:    $B_M \leftarrow$ all genes in $M$ that are linked to a gene in $P$   }
18:    $B_P \leftarrow$ all genes in $P$ that are linked to a gene in $M$   }
19:    $B \leftarrow B_M \cap B_P$                                }
20:    for $child \in \{c_1, c_2\}$ do                            }  Geo. Local Opt.
21:       for $i \in B$ do                                        }
22:          REPAIR($child, i$)                                   }
23:          LOCALSEARCH($child, i$)                              }
24:    $sorted \leftarrow$ SORT($\{p_1, p_2, c_1, c_2\}$)         }  ER Scheme - step 2
25:    $p_1 \leftarrow sorted[0]$ ; $p_2 \leftarrow sorted[1]$    }
26: until TERMINATE()
27: return best individual
```

The algorithm is given in Algorithm 2. Compared to Algorithm 1, instead of mutation the geometrically local optimizer is used, and the elitist recombination scheme combines selection and replacement. The following subfunctions are used:

INITIALIZE(*ind*): initialize the individual *ind*, making sure it is a feasible solution which satisfies all problem constraints. In the map labeling example, every labeling was a feasible solution, so INITIALIZE could just assign a random labeling to each individual. We will see that this is not always the case in the following case studies.

LINKED(*ind*,*i*,*j*): check if gene i is linked to gene j in individual *ind*. Returns TRUE or FALSE.

REPAIR(*ind*, *i*): resolve any constraints for gene i that make the solution *ind* infeasible. We will see examples of this operator later.

LOCALSEARCH(*ind*, *i*): perform local search on gene i that makes the solution *ind* more fit, or at least not less fit. In the map labeling example the slot filling procedure performed this duty.

SORT(S): return a sequence containing the set S sorted on fitness. With equal fitness, children take precedence over parents.

TERMINATE(): check whether the algorithm should be stopped. Returns TRUE or FALSE.

5 Generalization while preserving structure

Generalization while preserving structure is the problem of choosing characteristic features of a given set of objects. Since on a large scale map not all features can be shown, a subset of features should be found that expresses more or less the same information. The specific problem we study deals with point features: from these points a minimal subset of *representatives* should be chosen such that two representatives are not too close to each other, and every point which is not a representative is close to one. We formalize our problem as follows. The input consists of a set P of points, and two parameters ε_n and ε_r. Find the smallest subset $R \subset P$ such that: 1) for every two representatives $p, q \in R$ we have $\mathit{distance}(p, q) > \varepsilon_r$ and 2) for every point $p \in P$ there has to be a point $q \in R$ such that $\mathit{distance}(p, q) < \varepsilon_n$. See Figure 13 for an example.

This problem has the interesting property that is has two equally important constraints. How can we build a GA that solves this problem? There are now two kinds of conflicts: representatives that are too close together and normal points which are too far from a representative. One approach is to define a penalty function for each type of conflict and combine them to give the fitness of the individual, which should be minimized to 0. However, this would only produce feasible solutions and not the optimal solution. So we add yet another subfunction to the fitness function which counts the number of representatives, also to be minimized. Our fitness function now has the following form:

$$\mathit{fit}(R) = w_1 \cdot \#\mathit{repConflicts} + w_2 \cdot \#\mathit{normConflicts} + w_3 \cdot |R| \,.$$

Here $\#\mathit{repConflicts}$ is the number of representatives that are too close to another representative, $\#\mathit{normConflicts}$ are the number of normal points that are too far from a representative, and $|R|$ is the number of representatives in the solution. We also need to use weighing-factors w_1, w_2, and w_3 which have to be tuned

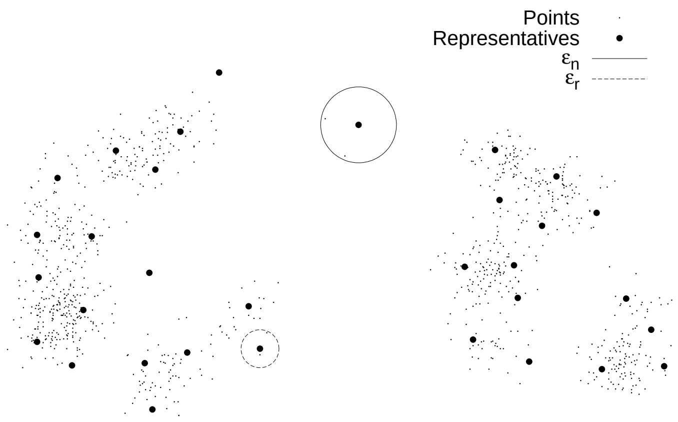

Figure 13: A set of points which is represented by a small subset. The top circle with radius ε_n contains all the points represented by the bold point. The bottom circle with radius ε_r should contain only one representative.

to balance the effect of the three constraints on the overall fitness. Our fitness-function has become unnecessary complicated however. There is an easier way to do it.

We do not let the GA search in the whole search space of all possible sets of representatives, of which many will violate several constraints. Instead, we start with feasible solutions and maintain the property of feasibility (by using the operator REPAIR) throughout the run. Then the fitness function can simply be the number of representatives ($\mathit{fit}(R) = |R|$), which is to be minimized. For the encoding we use a gene for every point and two alleles corresponding to whether the point is a representative or not.

We can fill in the framework to obtain our GA. Before we do so however, we have to decide what the linkage in this problem is. What constitutes a building block and has to be preserved during crossover? First, we define a *neighbor* of a point p to be a point that is within distance ε_n from p. Again the linkage is determined by geometry. A building block consists of a representative and "close-by" representatives. More precisely, two points are linked if they are neighbors or have a common neighbor. In other words, if there is a point q which is both a neighbor of p and a neighbor of r, then p is linked with r. See Figure 14.

Next we fill in the components of the framework:

INITIALIZE(*ind*): we have to take care that the solution is feasible after initialization. Therefore, after assigning to all genes the value specifying that the point is not a representative, all genes are traversed in random order and the procedure REPAIR(ind, i) is applied to each gene.

LINKED(*ind*,*i*,*j*): the two genes are linked if their corresponding points are linked.

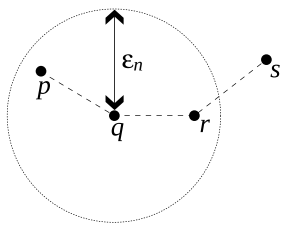

Figure 14: Points connected with a dashed line are neighbors. Point p is linked with points q and r, but not with s.

Repair(ind, i)**:** there are two cases:

1. if there is a conflict and the point corresponding to the gene is a representative, the conflict is caused by two representatives being too close. This can be resolved by making the point a normal point. For neighbors of i, new conflicts can arise and therefore Repair is applied recursively on them.
2. if there is a conflict and the point corresponding to the gene is a normal point, the conflict is caused by the fact that no representative is nearby. In this case, we can make the point a representative. Since $e_n > e_r$, this does not cause any new conflicts.

LocalSearch(ind, i)**:** in this example no local search is performed.

Note that a good solution spreads the representatives evenly to cover the input points. If more representatives are wanted where the map is dense, then ε_n and ε_r could be derived from the local density.

Results.

We generated twenty random maps similar to Figure 13. Each map was created by randomly choosing twenty "capitals" on a grid of 1000 by 500 units. Each of the capitals also contains a random weight to specify how important it is. Points were added to the map by trying to place them near a capital, until the total number of points was 1000. The decision to place a point near a capital was made by flipping a coin biased with the weight of the capital. The location of the point, when placed, was calculated using a Gaussian distribution for both x- and y-coordinates, centered at the capital. The resulting map gives a more realistic input than just randomly dispersing points.

We ran the GA five times on those maps and compared the average run against two heuristic methods one may think of when confronted with the problem. The GA was ran with a population size of 150. The two heuristics work as follows:

Greedy 1: First, all points are sorted on their distance from the center of the map[1]. The center is the average of the coordinates of all the points. Then, starting with the point closest to the center, they are examined in this order. If the point is not represented yet, it is made a representative.

[1]We also experimented with a left-to-right ordering, without observing significant differences.

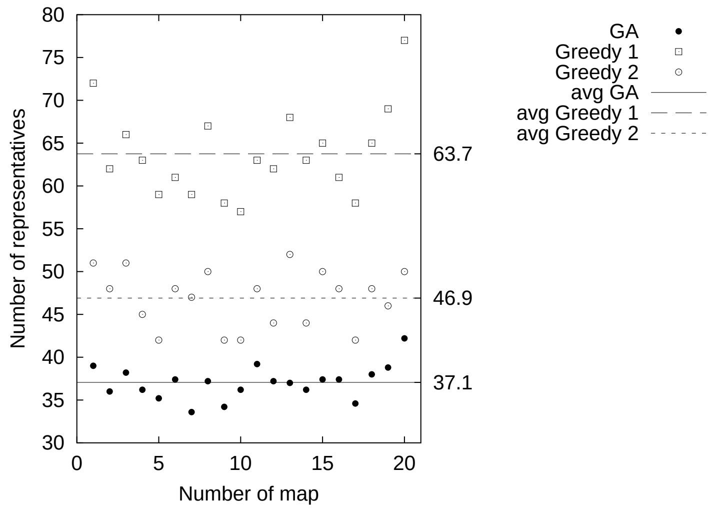

Figure 15: Comparison between algorithms for generalization ($\varepsilon_n = 50$, $\varepsilon_r = 25$).

Greedy 2: This heuristic starts with a list of the points sorted on their number of neighbors. The point which has the largest number of unrepresented neighbors is chosen and becomes a representative, after which the list is updated. This process continues until the list is empty.

The results of this comparison are shown in Figure 15 and Figure 16, illustrating that the GA performs better than the two other heuristics. However, the running times of the algorithms differ much: both the greedy algorithms took about three of four seconds to terminate, whereas the time to complete a run of the GA ranged from 90 to 200 seconds. On the other hand, it should be taken into account that the GA used was a research tool, and thus not optimized for speed.

6 Line simplification

Line simplification is the problem of removing some of the points of a polyline while maintaining approximately the same shape. See Figure 17 for an example. We consider the following variant: we are given a polyline (that is, a sequence of N points, with line segments drawn between successive points) and a parameter ε. The polyline is denoted $P_N = p_1 p_2 \ldots p_N$. None of the line segments intersect each other except when they share an endpoint. Find a polyline using a subset of the original points (including the start and end point), such that the number of points is minimal and the maximum deviation of the simplified line from the original line is smaller than ε.

The deviation is measured from a point on the original line to the line segment on the simplified line that *simplifies* (see below) that point (see Figure 18).

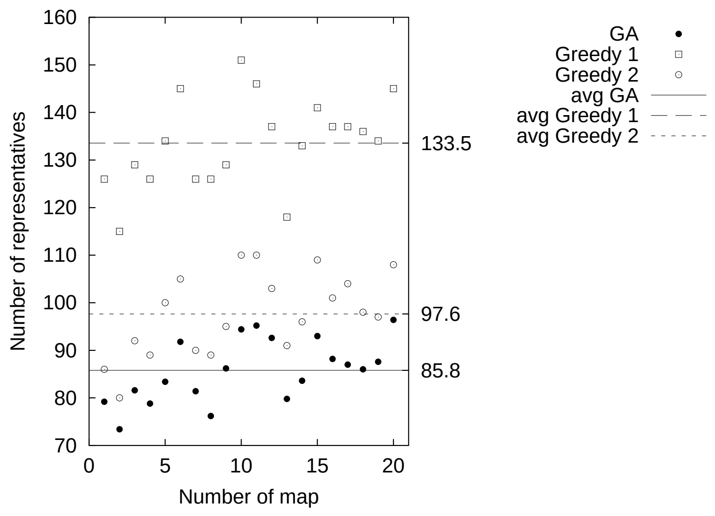

Figure 16: Comparison between algorithms for generalization ($\varepsilon_n = 30$, $\varepsilon_r = 20$).

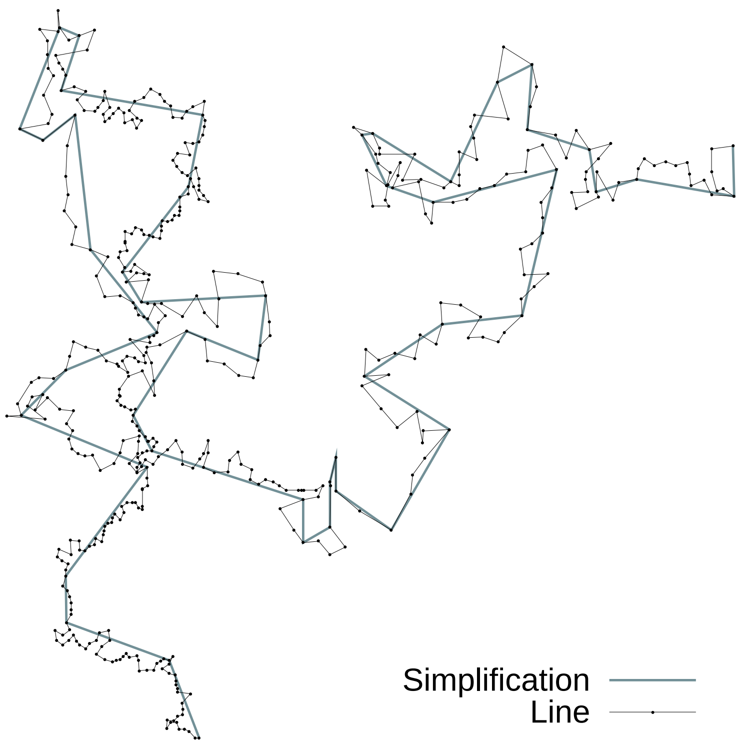

Figure 17: Simplifying a line with distance from original line less than ε.

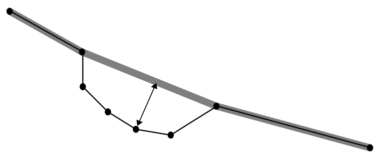

Figure 18: The deviation of the simplified line (bold) from the original line (thin).

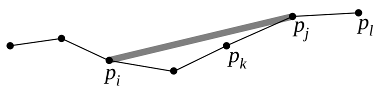

Figure 19: The bold segment simplifies points p_i, p_k and p_j, but not p_l.

A point that participates in the simplified line is called a *participant.* We call a point p_k *simplified* by segment $\overline{p_i p_j}$ (see Figure 19) if:

- p_i and p_j are participants,
- $i \leq k \leq j$
- there is no participant p_l such that $i < l < j$.

The encoding used consists of a gene for every point with two alleles corresponding to whether the point is a participant or not. During crossover we want long line segments (building blocks) to be transfered as a whole. There is linkage between two genes if their corresponding points are simplified by the same segment.

To demonstrate how one can incorporate additional constraints it is required that all angles in the simplification are larger than a certain α (for example 60 degrees). The exception is when the two segments making the angle also occur in the original line.

INITIALIZE(*ind*): choose at random alleles for all genes. To solve conflicts and make the solution feasible, we traverse all genes in random order and apply REPAIR to them. When the solution is feasible, again traverse all genes in random order and apply LOCALSEARCH to them to get a better solution.

LINKED(*ind*,*i*,*j*): as explained above, two loci are linked if their corresponding points are simplified by the same segment. Note that this linkage can change whenever the individual changes.

REPAIR(*ind*, *i*): if there is a conflict, it must be because i in the original line is at distance $\geq \varepsilon$ from the closest point on the segment in the simplification that simplifies i. Making i a participant solves the problem, but a recursive call on all genes corresponding to the points simplified by the new segments is necessary. To avoid spikes (two consecutive segments with a small angle), the procedure CHECKANGLE(i) is called when i is a participant.

LOCALSEARCH(*ind*, *i*): the simplification can be improved if i is a participant and redundant. If i can be made a normal point and no conflicts arise, the change is kept. If a conflict arises, the original situation is restored.

CheckAngle(i): if the angle between the two segments joining at point i is smaller than a given angle, rearrange the simplification. This is done by first setting all points that are simplified by the two segments and then removing points in a random order when it is possible (i.e. they are redundant as a participant and removal does not create spikes).

Results.

Several line-simplification algorithms are known, such as the iterative refinement method of Douglas and Peucker [2] and the graph algorithm of Imai and Iri [10]. The algorithm by Imai and Iri casts the simplification problem in a graph model and uses a standard algorithm for computing the shortest path to find a valid simplification with a minimal number of points. Unfortunately, both algorithms deal strictly with generating a simplification with a minimal number of points and have no provision for other constraints such as avoiding spikes.

To illustrate the performance of our GA we compared against the algorithm of Douglas and Peucker. We also compared against the algorithm by Imai and Iri, which is known to give an optimal result. To remove spikes, we applied **CheckAngle(i)** to all participants of the simplifications the algorithms found.

The algorithms were run on randomly generated lines similar to the one shown in Figure 17. The procedure that generated the polyline is recursive and starts with a straight line segment from coordinates $(0, 0)$ to $(800, 800)$. A deviation d is randomly calculated using a Gaussian distribution. The line segment is split in two new segments by adding a new point that is distance d from the midpoint of the original segment. The new segments are the input for the next recursion step. This process continues until the polyline contains 500 points. As a final step the line is made intersection free by reversing chains of segments where an intersection occurred. For example, if segments $\overline{p_i p_j}$ and $\overline{p_k p_l}$ intersect, the polyline $P_N = p_1 \ldots p_i p_j p_{j+1} \ldots p_{k-1} p_k p_l \ldots p_N$ is changed to $P_N = p_1 \ldots p_i p_k p_{k-1} \ldots p_{j+1} p_j p_l \ldots p_N$.

The GA was run five times on each line and the average was used to compare against the other algorithms.

The results are shown in Figure 20 and Figure 21. The GA finds the best solutions thanks to its capability of handling all constraints during the run. Of course, the algorithm by Imai and Iri can be extended to build a graph containing only connections which do not produce spikes. It would then again yield an optimal result. However, this is a non-trivial task that becomes harder with every constraint that is added. The genetic algorithm is much more flexible and conceptually simpler.

7 Discussion

As mentioned in the Introduction (Section 1), a solver for hard problems in a GIS context has to have the following properties:

1. The solver should be capable of giving close to optimal solutions.
2. It should be possible to extend the problem definition to solve additional constraints.

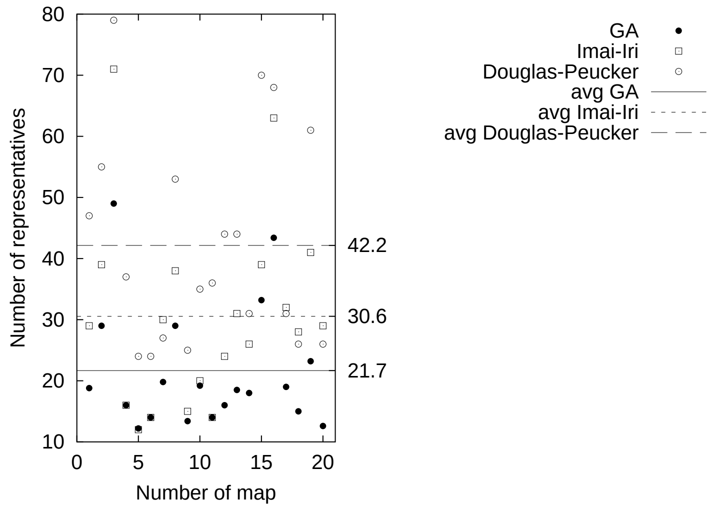

Figure 20: Comparison between algorithms for line simplification ($\alpha = 60°$, $\varepsilon = 40$).

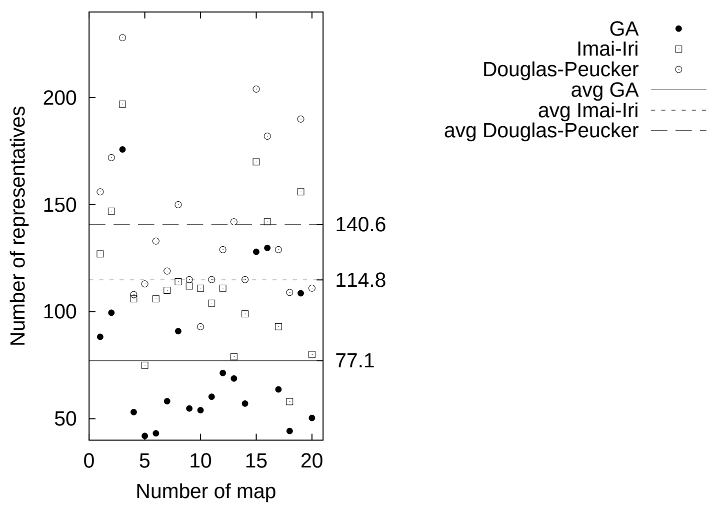

Figure 21: Comparison between algorithms for line simplification ($\alpha = 90°$, $\varepsilon = 20$).

3. The solver should integrate well with the GIS and be user-friendly.

The first requirement basically states that when simple, greedy methods fail, a more powerful solver is needed. Often the problem will be combinatorially hard, which means that trying to find a fast (polynomial time) algorithm that guarantees to give the optimum is an infeasible approach. Therefore an algorithm such as a Genetic Algorithm, Simulated Annealing [11], Tabu Search [4], etc. should be used to find good solutions in a reasonable amount of time. Genetic Algorithms have been used in industry to solve other practical problems [3], and have proven to be powerful as well as flexible.

The use of geometrically local optimizers makes it possible to extend the problem definition. This is a major advantage compared to the use of other problem solvers. Most extensions are local requirements to satisfy aesthetical demands. As such, they should not be put in the fitness function but in the geometrically local optimizer. Putting them in the fitness function introduces the need for weighting parameters, which requires a tuning phase. The GA would have to be run many times before the weighting parameters can be set to appropriate values, after which it can only be run on data which resembles the tuning data. Using geometrically local optimizers the designer of the GA can decide what is good in a local setting and let the GA construct from these local solutions a global solution. No weighting parameters will be needed.

Avoiding the use of weighting parameters is an example of the third point as well, which acknowledges the fact that the GA will be used by people who do not have much (if any) experience with GA's. Therefore it should be avoided that the user has to set parameters such as the probability of crossover. The only parameter in the proposed framework which has to be set by the user is the population size.

The population size determines the amount of computational effort that is allocated to the GA. It may be desirable to allow the setting of this parameter (although we will discuss how to eliminate it in a moment). Since the GA gracefully deteriorates in the quality of its solutions, the setting of the population size is a measure for the response time of the algorithm. In other words, if the user wants a solution fast and accepts a lower quality, he/she can set a low population size. When a production quality map is needed, the user sets a high population size and lets the computer run for a night to produce a high quality output.

However, in order to completely eliminate all GA specific parameters, a method should be found to eliminate the need of setting the population size. In general, three different methods are used to set the population size for a specific run of the GA:

1. Reckoning: here the user gets a feel of how large the population size should be simply because he/she gains experience when using the GA.

2. Analysis: by modeling the dynamics of the algorithm one can find a population sizing equation which gives recommendations on the size of the population. Examples of this method are the papers by Goldberg *et al.* [6] and Harik *et al.* [7]. An analysis of the population sizing behavior of the map labeling GA was given in another paper [22].

3. Adaptive sizing: the method which completely eliminates any user interaction is based on adaptive population sizing. Examples of this approach

are Smith and Smuda [15], Sawai and Kizu [14] and Harik and Lobo [8]. Especially the last method can be directly applied as an immediate ("off the shelf") solution to avoid setting the population size.

Summarizing, we think GA's using a linkage-respecting crossover and a geometrically local optimizer are a useful tool for GIS-developers since:

1. They are capable of solving problems that involve vast, complex search spaces.
2. They are flexible and extensible.
3. They are practically usable in a GIS since they are not limited to a specific problem and no parameters need to be tuned.

Further research would entail the application of this framework to other problems that arise in the GIS world. Problems that may lend themselves to this approach are:

- the coloring of countries on a map, where neighbors should have different colors and the total number of colors used should be minimal,
- political redistricting (accumulating areas into larger districts such that the resulting districts are as fair a representation of the whole as possible), and
- outline simplification of buildings (maintaining approximately the same shape using less points, while preserving right angles in corners and avoiding intersections).

8 Conclusion

This paper describes a general framework for solving a certain class of problems arising in geographical applications that are difficult to solve due to the large, complex search spaces involved. The method is based on genetic algorithms and is widely applicable, easy to extend with other constraints, and suitable for integrating into a GIS due to a lack of tuning and parameter setting. This was demonstrated in the three case studies.

References

[1] J. Christensen, J. Marks, and S. Shieber. An empirical study of algorithms for point-feature label placement. *ACM Transactions on Graphics*, 14(3):203–232, 1995.

[2] D. H. Douglas and T. K. Peucker. Algorithms for reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer*, 10(2):112—122, 1973.

[3] Evonet - Resources : Case studies. `http://www.dcs.napier.ac.uk/evonet/coordinator/resources/casestudies.html`.

[4] F. Glover. Tabu search. In Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems*, volume C, pages 70–141. Blackwell Scientific Publishing, 1993.

[5] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, Mass., 1989.

[6] D. E. Goldberg, K. Deb, and J. H. Clark. Genetic algorithms, noise, and the sizing of populations. *Complex Systems*, 6(4):333–362, 1992.

[7] G. Harik, E. Cantú-Paz, D. E. Goldberg, and B. L. Miller. The gambler's ruin problem, genetic algorithms, and the sizing of populations. *Evolutionary Computation*, 7(3):231–253, 1999.

[8] G. Harik and F. Lobo. A parameter-less genetic algorithm. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 258–265. Morgan Kaufmann, July 1999.

[9] J. H. Holland. *Adaptation in natural and artificial systems.* University of Michigan Press, Ann Arbor, 1975.

[10] H. Imai and M. Iri. Polygonal approximations of a curve — formulations and algorithms. *Computational Morphology*, 1988.

[11] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598), 1983.

[12] J. Marks and S. Shieber. *The computational complexity of cartographic label placement.* Technical Report TR-05-91, Harvard University, March 1991.

[13] G. Raidl. A genetic algorithm for labeling point features. In *Proc. of the Int. Conference on Imaging Science, Systems, and Technology*, pages 189–196, Las Vegas, NV, July 1998.

[14] H. Sawai and S. Kizu. Parameter-free genetic algorithm inspired by "disparity theory of evolution". In A. Eiben, T. Bäck, M. Schoenauer, and H. P. Schwefel, editors, *Lecture Notes in Computer Science, Vol. 1498: Parallel Problem Solving from Nature PPSN-V*, pages 702–711. Springer-Verlag, 1998.

[15] R. E. Smith and E. Smuda. Adaptively resizing populations: Algorithm, analysis, and first results. *Complex Systems*, 9(1), 1995.

[16] D. Thierens. Scalability problems of simple genetic algorithms. *Evolutionary Computation*, 7(4):331–352, 1999.

[17] D. Thierens and D. E. Goldberg. Mixing in genetic algorithms. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 38–45, San Mateo, CA, 1993. Morgan Kaufmann.

[18] D. Thierens and D. E. Goldberg. Elitist recombination: An integrated selection recombination GA. In *Proc. IEEE Int. Conf. on Evolutionary Computation*, pages 508–512. IEEE Service Center, Piscataway, NJ, 1994.

[19] S. van Dijk, D. Thierens, and M. de Berg. *Robust genetic algorithms for high quality map labeling.* Technical Report TR-1998-41, Utrecht University, 1998.

[20] S. van Dijk, D. Thierens, and M. de Berg. On the design of genetic algorithms for geographical applications. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 188–195. Morgan Kaufmann, July 1999.

[21] S. van Dijk, D. Thierens, and M. de Berg. Designing genetic algorithms to solve GIS-problems. In *Spatial Evolutionary Modeling.* Oxford University Press, 2000. (to appear).

[22] S. van Dijk, D. Thierens, and M. de Berg. Scalability and efficiency of genetic algorithms for geometrical applications. In M. Schoenauer, editor, *Parallel Problem Solving from Nature — PPSN VI*, 2000. (to appear).

[23] O. Verner, R. L. Wainwright, and D. A. Schoenefeld. Placing text labels on maps and diagrams using genetic algorithms with masking. *INFORMS Journal of Computing*, 9(3), 1996.

[24] A. M. Verweij and K. Aardal. An optimisation algorithm for maximum independent set with applications in map labelling. In *Lecture Notes in Computer Science, Vol. 1643: Proc. 7th Annu. European Sympos. on Algorithms (ESA'99)*, pages 426–437. Springer-Verlag, 1999.