Genetic Algorithms for Map Labeling

Genetische Algoritmen voor het Plaatsen van Namen op Kaarten

(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de Rector Magnificus, Prof. Dr. W. H. Gispen, ingevolge het besluit van het College voor Promoties in het openbaar te verdedigen op maandag 26 november 2001 des middags te 12:45 uur

door

Steven Ferdinand van Dijk
geboren op 14 april 1974, te Amsterdam.

promotor:	Prof. Dr. M. H. Overmars Faculteit Wiskunde & Informatica
co-promotor:	Dr. Ir. D. Thierens Faculteit Wiskunde & Informatica
co-promotor:	Dr. M. T. de Berg Faculteit Wiskunde & Informatica

ISBN 90-393-2864-1

The research in this thesis was supported by the Netherlands Organization for Scientific Research (NWO) .

Dedicated to:
my father, from whom I inherited curiosity, and
my mother, from whom I inherited perseverance.

Contents

Contents **v**

List of Algorithms **ix**

1 Introduction **1**
1.1 Map labeling 4
1.2 GAs for map labeling 10
1.3 Main results and overview of thesis 12

2 Genetic Algorithms **15**
2.1 The algorithm 16
2.1.1 Population size 20
2.1.2 Encoding 21
2.1.3 Initializers 22
2.1.4 Fitness function 23
2.1.5 Selection schemes 24
2.1.6 Crossover 27
2.1.7 Mutation 29
2.2 Theory 29
2.2.1 Schemata and linkage 29
2.2.2 The schema theorem and adequate mixing 32

3 A GA for point-feature map labeling **35**
3.1 A simple GA 36

3.2 Applying theoretical insights . . . 42
3.2.1 Analysis of the simple GA . . . 42
3.2.2 Other GAs . . . 45
3.2.3 A GA conforming to theory . . . 48
3.3 Comparison with other techniques . . . 59
3.4 Discussion . . . 71
3.5 Conclusion . . . 74

4 Scale-up behavior of the GA 75
4.1 The models . . . 76
4.1.1 Determination of t^* . . . 77
4.1.2 Determination of n^* . . . 81
4.2 Adherence of assumptions . . . 87
4.3 Results . . . 89
4.4 Discussion . . . 97
4.5 Conclusion . . . 98

5 A design framework for GIS problems 101
5.1 A general algorithm for GIS problems . . . 103
5.2 Point-feature map labeling . . . 106
5.2.1 Results . . . 112
5.3 Line simplification . . . 117
5.3.1 Results . . . 121
5.4 Generalization while preserving structure . . . 125
5.4.1 Results . . . 129
5.5 Discussion . . . 130
5.6 Conclusion . . . 133

6 Point and line feature labeling 135
6.1 Measuring distances and proximity constraints . . . 136
6.2 The GA . . . 139
6.2.1 Finding candidate label positions . . . 140
6.2.2 The number of labels for a line feature . . . 146
6.2.3 Initialization . . . 147
6.2.4 Geometrically local optimizer . . . 152
6.3 Comparisons . . . 154
6.4 Discussion . . . 158
6.5 Conclusion . . . 163

7 Conclusion 165

A Notation **169**

Acknowledgements **173**

Samenvatting **175**

Curriculum Vitae **179**

Bibliography **181**

Index **193**

List of Algorithms

1 The standard genetic algorithm. 19
2 Genetic algorithm with elitist recombination. 26
3 GA for GIS problems. 105
4 LOCALSEARCH(ind, x_i) 109
5 CHOOSESLOT(ind, x_i) 111
6 Finding label position candidates. 146
7 Initialization of a line feature. 152

CHAPTER 1

Introduction

This thesis is (mostly) about making maps. Specifically, it is concerned with the problem of automating the placement of names on them. Maps are a very convenient means of presenting all kinds of information in a way that is easily understood. Making a good map is hard, though. There is a limit on the amount of information one can display in a fixed area, because beyond a certain point the map will become unreadable. The art of map making lies in providing as much relevant information as possible, while keeping the map easy to use. Displaying all the objects (or *features*) on the map is not always feasible. Sometimes unimportant features have to be removed, or merged with other features on the map. On the other hand, important features may need to be exaggerated to enhance their visibility. However, just displaying all the geometry is not enough to make a usable map. The user also needs a way of identifying the features. Usually, this is done by writing the name of each feature next to it.

We can divide the features in three classes, based on their dimension. There are zero-dimensional features called *point features*, like cities and small symbols. Then there are one-dimensional features, like highways and rivers, which are called *line features*. Finally, there are two-dimensional features such as countries and lakes, which are known as *area features*. Note that this classification depends on scale, since a city can be a point feature on a small-scale map, but on a large scale it is best classified as an area feature. A feature can have a name; the graphical depiction of the name on the map is called a *label*. The process of putting labels near their features is called *label placement* (see Figure 1.1).

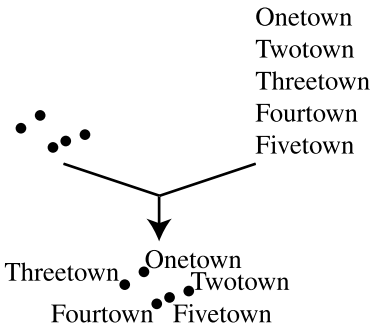

Figure 1.1: The map-labeling problem.

Label placement is usually done manually by cartographers, who spend much time on it. It is both a craft and an art, since a map should be both practical and aesthetically pleasing. A practical map has labels which are easy to read, clearly associated with their features, and easy to spot. The overall impression should be harmonious, balanced, and not too crowded.

Although cartographers can make great maps, there is a need for *automated* label placement. It becomes increasingly the case that maps are generated on demand, with the use of a computer, which means that labels have to be placed automatically. Geographical information systems (GISs) are software systems which can store a huge amount of geographical data, and allow the user to analyze, process, and view the data. When the final result is presented as a map, it has to be completely computer-generated. Storing hand-crafted maps is simply not possible, since the user can select and combine which information to display in many different (and unpredictable) ways, creating any map he likes. Therefore, automated label placement is a desirable capability of a GIS, since without names it can be hard to make sense of a map.

In cartography, the label-placement problem is typically expressed using a large number of rules. These rules are almost all local in nature.* If a label satisfies these rules, then its quality is high. A good map has labels for each of which the quality is high. Furthermore, label placement can also be seen as an *optimization problem.* Such problems are defined by a *cost function* which has to be optimized (either maximized or minimized). The cost function is used to evaluate the map as a whole, contrasting with the previous view where the scope of a rule is local. We will say more about these two perspectives on the hardness of the problem shortly.

Genetic algorithms (GAs) are heuristic solvers for optimization problems. Based on the theory of Darwinian evolution, they are able to "evolve" solutions

*A few *global* cartographic rules exist, such as the rule that labels in dense areas should be "centrifuged" outwards.

using a process similar to adaptation in biology. A GA uses a population of solutions from which it selects the best ones: in a maximization problem, those for which the cost function gives high values. From these the next population is generated, by altering and combining them. This two-step cycle (selection and generation) is iterated until the population is converged (all members are very similar). The best member of the final population is the solution returned by the algorithm. Hopefully, it is close to optimal.

In this thesis we will explore the possibilities of labeling maps using genetic algorithms. When applied to map labeling, the solutions of a GA will be labelings for a given map. A good labeling places as many labels as possible that are not intersected by other labels. But a good labeling also places each label well with respect to its surroundings. For example, if a city is adjacent to a coastline, its label is best placed in the water. The GA should be able to handle both aspects of the problem.

The goals of this thesis are to study the use of GAs to solve the map-labeling problem, and the application of theoretical insights about GAs to a real-world problem. Paralleling these two goals, the thesis is aimed at two different audiences, broadly divided in those with a cartographic interest and those interested in GAs.

Readers with a background in cartography can find a new approach to solve the map-labeling problem. Other GAs that try to solve the problem* are known,[18, 107, 79] but we try to surpass them by offering a framework for reliably and efficiently solving the problem in a more general setting. We will describe a GA that can label point and line features (and it should be relatively easy to incorporate area features). Special care has been taken to make the GA flexible and easily extendible with additional constraints, without resorting to a weighted cost function. In fact, our techniques are more broadly applicable than just the map-labeling problem, and we describe in one chapter how GAs can be applied to other GIS problems, like line simplification and certain generalization tasks.

Researchers in the field of genetic algorithms can find something of their liking, too. We will design a GA based on GA theory. However, we will also take care to make the GA *practical*. We try to minimize the number of arcane parameters that have to be tuned or set beforehand, keep the cost function simple, and provide a way to easily extend the problem with additional cartographic rules. Our faithful application of theoretic principles allows us to give a theoretical analysis of the scale-up behavior of our GA. We devote an entire chapter to this analysis, in which evidence is given that the GA described is efficient. We also describe an extension of the classical GA—the geometrically local optimizer—that acts as a source of building blocks and combats the disrupting effect of crossover.

*These GAs are described in detail in Chapter 3.

We will make few assumptions about prior knowledge; map labeling and GAs will be explained in sufficient detail (in this chapter and Chapter 2) to understand later chapters. However, some basic knowledge about statistics is assumed in Chapter 4.

The remainder of this chapter is organized as follows. First, in Section 1.1, we explore the map-labeling problem in more detail. We give an overview of different variants, discuss why the problem is hard, and then describe the relevant literature. In Section 1.2 we argue why GAs are an interesting candidate for solving the problem. We end the chapter by giving an outline of the thesis, and describing our main results.

1.1 Map labeling

The map-labeling task can contain many aspects which can add to the complexity of the problem. For example, different instances of the problem can include different kinds of feature. Some problems only deal with point features, for example labelings of drill sites[119,110] (where a measurement has to be shown next to the location) or statistical plots (where data is printed in a scatter plot). Other problems, such as graph labeling,[52] include line features. The full cartographic problem has to deal with all three kinds of features: point features, line features, and area features.

The problem can also have a time constraint. For example, for dynamically generated maps,[74] where a user is quickly changing the region of interest, there is little time to generate the labeling. Other problems, such as the production of paper maps, allow for a considerable amount of time (and computational resources) in order to obtain a final result that looks good.

Another source of variety is the number of labels for a feature. For point features usually a single label suffices, but for some problems there is a need to place multiple labels.[78,77,53] This happens, for example, with multi-lingual maps, or when labeling a mountain peak with both its name and its elevation. For line features it is sometimes required to repeat labels, spread out over the feature.[8]

The features themselves can have different priorities, which influences the labeling.[88] Different cities can differ in importance due to their size. Also, different types of feature can be assigned different priorities. For example, the roads on the map can be more important than the cities. Labels of features with a low priority should not obstruct the labels of high-priority features.

The problem can be further complicated by the shape of the label. Often, the label is considered to be an axis-parallel rectangle in which the name of the feature is written, but sometimes curved labels are allowed. Instances where the labels are circles[19,90] or squares[28] have been studied too. A practical extension to the fixed-

sized rectangle is the *elastic label* as introduced by Iturriaga and Lubiw,[48] where the label can have different shapes depending on how the name of the feature is written. Long names can be split over multiple lines, which results in a differently shaped label.

Additionally, the placement model—the specification of the candidate positions for a label—used is a factor of importance. For point features there often is a standard placement model, where a label can be placed in four (or sometimes eight) predefined positions. Other models have more freedom in the placement of the label. For example, in the problem studied by Hirsch[43] the placement of labels is only constrained by the fact that a label should touch (but not overlap) a circle around the point feature. The sliding-label model, introduced by Kreveld et al.,[104] is similar; it only constrains the placement of the label by the requirement that it should touch the point feature. There are no standard placement models for line and area features, since those features can differ in shape considerably.

Furthermore, the requirements on the output can differ. On geographical maps, the dimensions of the labels are usually given, and we are concerned with placing the maximal number of labels that do not intersect other labels. Another possibility[110] is to find the maximal size of the labels, under the condition that all labels have to be placed.

Lastly, there is an interesting connection between the map-labeling problem and the *name-selection* problem. The latter problem, also called *settlement selection*,[105,61] is concerned with finding a subset of features on the map, when it is not feasible to label all features. Selection of an appropriate subset can be tricky, since one needs to deduce how important a certain feature is in relation to its surroundings and the use of the map. Name selection can be done before labeling, in which case only the names of the selected features have to be placed. However, there is no guarantee that it is possible to label all selected features without intersecting labels. On the other hand, name selection may be too selective and remove too much. It makes sense, therefore, to integrate map labeling and name selection.

In this thesis we focus on the case of cartographic maps, where the number of non-intersecting labels should be maximized. Labels will be rectangles of fixed dimensions (but they can be different for each label). We will develop an algorithm that handles both point and line features. For point features, we will use a placement model with fixed positions. No such placement model exists for line features, so we will describe a procedure that generates candidate positions. We will discuss a way to integrate name selection with the labeling process, and in the case of line features we will study the problem of placing multiple labels for a single feature. We will also describe a flexible approach to add cartographic constraints. This is exemplified by adding preferences for the positions of point-feature labels and categorizing the features of the map in different levels of im-

portance. For example, the label of a major city should not be omitted in favor of the label of a small town.

The application we have in mind is where a user of a GIS creates an (unlabeled) map, then presses a button to label it, and finally prints it on paper. This means that the speed of the algorithm is not our main concern, although we will take care to make the algorithm run efficiently. It also means that we avoid as much as possible the use of parameters that are difficult to set by the user of the GIS. Examples of such parameters are parameters specific to the GA (like the probability of crossover) and weighting factors in the cost function.

We expect the input of our algorithm to consist of a list of features classified by type. Also, the *scale* of the map—the ratio between real-world coordinates and map coordinates—has to be specified. Figure 1.2 demonstrates some of the elements of the maps that we want to make:* different levels of importance (indicated for cities by font size), preferred positions for point-feature labels, and multiple labels (with appropriate spacing) for line features.

Why is map labeling so difficult? The difficulty of the problem stems from the following two factors.

Firstly, even basic instances of the problem have been proven to be NP-hard. An example is the problem where the number of non-intersecting labels has to be maximized, given a set of point-features whose labels can be placed in one of four fixed positions.[64, 58, 28] NP-hardness proofs for other instances[51, 49, 104, 47] were found also.

The second reason why map-labeling is hard, is that many additional constraints have to be considered when placing a label.[46, 3, 116] For example, for a point-feature label the top-right position is the best placement, not considering the surroundings of the feature. On the other hand, if the label were separated from its feature by a river when placed there, the position is considered not so good. This kind of constraints are almost always very local in nature. We will refer to them as the "cartographic rules", although strictly speaking the combinatorial part also deals with a cartographic rule ("a label should not obscure another label"). We can divide the cartographic rules in three classes:[103] *association*, *visibility*, and *aesthetics*. Association rules express the fact that the user of the map should be able to associate clearly the label with its feature. Therefore, the label should be placed near its feature. A visibility rule states that a label should be clearly visible on the map. For example, a label should not be placed on a background with low contrast. The third class, aesthetics, is concerned with how nice

*The map was made using $\varepsilon_{close} = 12$, $\varepsilon_{far} = 30$, $\varepsilon_{medium} = 80000$, $\varepsilon_{mega} = 250000$, and medium scale. We discuss the constants $\varepsilon_{medium}, \varepsilon_{mega}$ in Section 5.2 on page 107, the constants $\varepsilon_{close}, \varepsilon_{far}$ in Section 6.1 on page 138, and the scale in Section 6.2 on page 144.

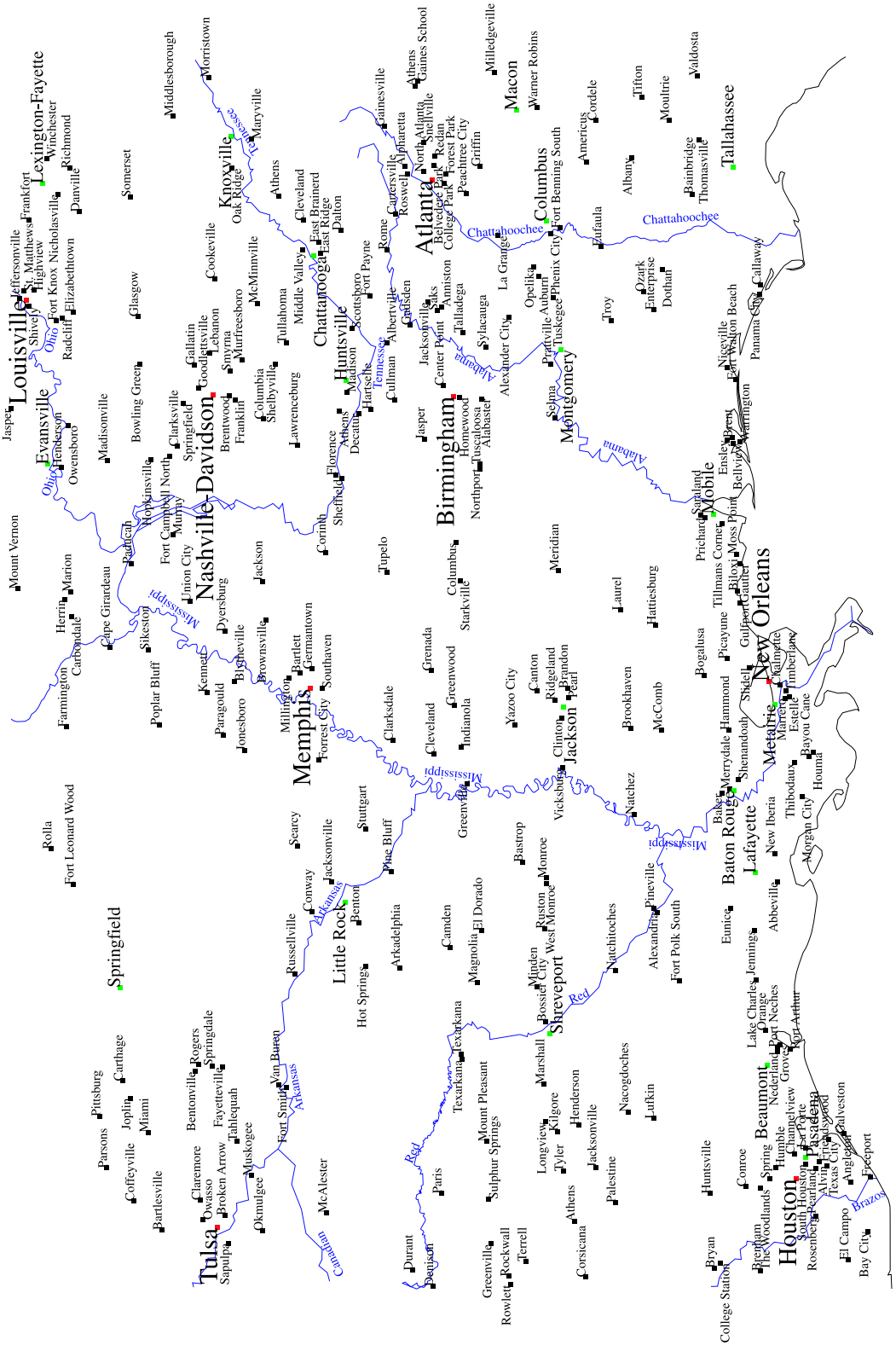

Figure 1.2: Example of a map (Mississippi area).

the label looks. An example of such a rule is that a curved label should not have too many inflection points.

Some cartographic rules are relatively strict ("a line-feature label should be repeated at suitable intervals"), others are less important ("a point-feature label should preferably be placed in the top-right corner"). Rules also often conflict with each other, and require a suitable compromise to be found.

Previous work

A lot of research has been done on various instances of the map-labeling problem. Initially, the main focus was on satisfying the cartographic rules as they were written down by cartographers like Imhof,[46] Alinhac,[3] and Youli.[116] This resulted in many rule-based systems (for example by Doerschler and Freeman[20] and by Jones and Cook[50]), which performed backtracking when a label could not be placed without breaking the rules. At the same time, heuristics like the label-repulsion method of Hirsch[43] and integer programming with relaxations by Cromley[15] and Zoraster[117] were devised. Later, NP-hardness proofs for the basic map-labeling problem were given independently by Marks and Shieber,[64] Kato and Imai[58] and Forman and Wagner.[28] Still, Verweij and Aardal[109] showed that they could solve instances with up to 950 point features optimally before the required computation time became too demanding. Recently, much research has been done in applying heuristic combinatorial problem solvers to the map-labeling problem. Methods like simulated annealing (by Christensen et al.[13]), evolutionary algorithms (by Djouadi,[18] Verner et al.,[107] Raidl,[79] and Preuss[76]), and tabu search (by Yamamoto et al.[115]), known for successfully solving other hard combinatorial problems, were able to find good solutions for instances of the point-feature map-labeling problem. They usually focused on finding labelings with as few overlapping labels as possible, and mostly disregarded the other cartographic rules that add to the complexity of the problem. Heuristics for maximum independent set were applied (first by Agerwal and Kreveld,[1] and recently by Strijk et al.[89,87,108]) to the same problem and compared favorably with other methods. They too disregarded other rules.

Edmondson et al.,[23] however, extended the simulated-annealing approach of Christensen et al.[13] by using a cost function that sums a large number of measurements of different cartographic rules. They also considered line features with a single label and area features, making their algorithm a general one. Another general approach that considered all types of feature was given by Wagner and Wolff.[111] They reduced the problem to a constraint-satisfaction problem by using the features as variables, the candidate positions as their domains and overlaps between positions as constraints. This method is general in the sense that it can also solve problems with line and area features, provided that candidate positions for

the label are calculated. Additional cartographic rules seem difficult to express. Kakoulis and Tollis[54] described a method which formulates the problem in graph theory, after which a bipartite matching algorithm is applied. They assign each candidate position a cost (which can express its adherence to cartographic rules) and minimize the total cost. The method can handle all feature types, but again it seems difficult to add additional constraints (except by incorporating them into the cost function).

In developing a general algorithm for the map-labeling problem, a difficult task is to combine cartographic rules with the combinatorial problem of maximizing the number of non-intersecting labels. As stated above, they are essentially very different. The combinatorial aspect is global, whereas the cartographic rules are mostly local in nature. All the algorithms that were successful in solving the combinatorial part did so by using a cost function that counted the number of non-intersecting labels. A possible method for dealing with the cartographic rules is to simply express them in the cost function. The cost function, however, evaluates the whole map, and therefore is global in nature. For example, the cost function for the most basic problem (maximizing the number of non-intersecting point-feature labels) can be expressed as $f_{cost}(\mathbf{x}) = \mathit{free}(\mathbf{x})$. Here, the cost function uses a function $\mathit{free}(\cdot)$ that counts the number of *free* (non-intersecting) labels. The parameter $\mathbf{x}$ denotes the solution that is evaluated. Next we can add the constraint that point-feature labels have preferred positions (with the top-right position as the best). This is expressed using the function $\mathit{pref}(\cdot)$, which measures how well all labels are placed with respect to the preference in positions.

How should the function $\mathit{pref}(\cdot)$ be combined with the function $\mathit{free}(\cdot)$? Multi-attribute utility theory (MAUT)[59] provides a framework in which different attributes of a system can be combined to yield a utility function that captures the interactions between the attributes. For the map-labeling problem, the attributes are measurements of properties of the labeling, like $\mathit{pref}(\cdot)$ and $\mathit{free}(\cdot)$. The cost function is the utility function that combines these attributes in the MAUT framework. Informally, the framework dictates that attributes for which *trade-offs* can be made can be expressed using an additive function. A trade-off means that the decrease of one attribute can be compensated by the increase of another, yielding a solution which is equally good. In such a case, the cost function would be a summation of the subfunctions. Under other circumstances, it may be necessary to use more complicated forms, like a multiplicative function such as the following: $f_{cost}(\mathbf{x}) = (1-w_1)\,f_1(\mathbf{x}) \cdot (1-w_2)\,f_2(\mathbf{x})$.

For our example, a complication that arises is that in most cases one usually wants the number of free labels to be maximal, and does not want to make a trade-off between free labels and preferred positions for the labels. The combinatorial aspect of the problem has the highest priority, and should not be affected by the application of the cartographic rules. For attributes that measure different

cartographic rules, the trade-off condition seems more appropriate. The MAUT framework can, in those cases, probably help to design a sensible utility function.

In any case, in literature, it is customary to use an additive function to combine multiple cartographic rules. We will follow this practice to keep our examples familiar and intuitive. Note that we don't use cost functions consisting of multiple subfunctions ourselves—we handle the cartographic rules in a different way, as explained later. Additionally, the following discussion applies equally well to other kinds of combination of subfunctions.

The cost function is thus extended with the new constraint by summing the result of both functions. A problem is that every constraint needs a weighting factor to balance the priorities of the constraints. In our example, the number of free labels is more important than the number of labels in a preferred position. The resulting cost function takes the following form: $f_{cost}(\mathbf{x}) = w_1 \cdot free(\mathbf{x}) + w_2 \cdot pref(\mathbf{x})$. The constants w_1, w_2 are the weighting factors used. These weights have to be set to sensible values, which is not a trivial task. For example, Edmondson et al. used such a cost function in which $w_1 = 40$ and $w_2 = 1$. They say: "Suitable values for the weights were created intuitively and refined empirically."[23] They have to be set by trial and error, that is, by running the algorithm many times and evaluating the output. Each time the map has to be evaluated by hand, by looking at it, to see if all constraints are balanced properly. Even if one succeeds in setting the weights to good values, they are dependent on the maps used as test data. Different maps may yield unexpected results. Adding another constraint can therefore be very difficult.

Finding a good way to combine cartographic rules with the combinatorial aspect of the problem was an important goal when we designed the GA described in this thesis. In the next section we will discuss the reasons for considering GAs to solve map-labeling problems.

1.2 GAs for map labeling

In this thesis we will explore a new approach to the automated map-labeling problem that uses genetic algorithms (GAs). GAs are powerful, combinatorial problem solvers based on the theory of Darwinian evolution. We will discuss how GAs work in more detail in Section 2.1.

GAs are a promising candidate for solving the map-labeling problem. Two relevant characteristics of the map-labeling problem, and indeed of a lot of GIS-problems in general, are:

- The problem constraints can be viewed as belonging to different classes. As described in Section 1.1, the map-labeling problem has a clear global

combinatorial part and also has a part that deals with more local constraints. Since both parts have different kinds of constraints, they should be treated differently.

- The structure of the problem is determined by the geometry of the features and the labels. Placing a label near a feature is possible when the labels of surrounding features don't get in the way. Features which are further away, however, have less influence on the placement of the label.

These characteristics match well with GAs, as will be explained shortly. There are several reasons why it is interesting to investigate the use of GAs for solving the map-labeling problem.

GAs are powerful and flexible problem solvers. The map-labeling problem is hard in a combinatorial sense, so it makes sense to use a technique that is capable of solving hard problems. When the problem size becomes large we have to use a heuristic like genetic algorithms, simulated annealing or tabu search, to avoid an exhaustive search that takes too much time. GAs have shown to be successful in many different problem domains and have been used to solve various hard problems.

The clear geometrical structure of the map-labeling problem, as described above, provides another reason to use GAs. This property translates nicely to the *linkage* (see Section 2.1) of a problem. A GA finds good solutions by combining the best parts of other solutions. Exactly what a "part" (also called a *building block*) is, is determined by the linkage of the problem. Building blocks in the map-labeling problem are spatially local regions, and this information can be exploited in the design of a GA that searches for good solutions efficiently.

Finally, solving a hard combinatorial problem invariably requires a lot of computational effort. GAs are no exception to this rule. Fortunately, GAs are algorithms whose population-based nature makes them easy to parallelize.[12] Close to linear speed-ups can be achieved this way, provided that the evaluation of the cost function is the most time-consuming part of the algorithm.

A difficulty with using GAs for the map-labeling problem seems to be that all the cartographic rules have to expressed by using a cost function that is a combination of weighted subfunctions. The disadvantages of that approach were discussed before, in Section 1.1. We show that this is not necessary, and develop a GA where the cost function is deliberately kept simple; it only counts the number of free (non-intersecting) labels. The division of the problem constraints in two different classes is reflected in the design of our GA. The combinatorial constraint is expressed in the fitness function and is handled by the normal operation of the GA. All other constraints are handled in a novel operator, the geometrically local optimizer.

1.3 Main results and overview of thesis

As stated earlier, the goal of this thesis is twofold: to solve the map-labeling problem with genetic algorithms, and to use theoretical insights into the way GAs work for the design of a GA for a real-world problem. Our main contributions are thus as follows.

Firstly, we develop a GA capable of solving the map-labeling problem. We will start with the basic problem of maximizing the number of non-intersecting point-feature labels. Comparisons with existing algorithms show that our GA gives equally good, or better, results than the current state-of-the-art. We will then add additional constraints; for example, we show how to have preferred positions for labels, how to enforce that important cities always get labeled, and how to do integrated name selection. We will start with randomly generated data, but will also use real-world data of portions of the USA. The GA will be extended to handle both point and line features. The latter can have multiple labels. This is one of the few algorithms that allows for multiple labels for line features.

We tried to build a GA that is *practical* in that one can actually envision it being used in a GIS. We imagine that users just want to press a button and let the system label their map, and therefore the GA should integrate well in a GIS and a minimum of parameters should be set. Our GA differs from more conventional GAs in that we tried to avoid using a weighted cost function (the cost function gives a measure of how well a map is labeled).

Secondly, the other main theme of this thesis is the use of theoretical insights into how GAs work for the design of GAs for real-world problems. Theory should help the design of GAs for practical problems: a GA designer should try to apply the lessons from theoretical analysis, and not rely too much on craftsmanship.

The design of the map-labeling GA is founded on the identification of the so-called building blocks of the problem, and recognizes the need to obtain good mixing, minimize disruption, and avoid genetic drift and hitchhiking—see Chapter 2 for an explanation of these and other concepts from GA theory. This results in a GA that is robust in the sense that it reliably finds solutions for the map-labeling problem, and is extendible with new constraints. In fact, the techniques are useful for more problems with a geometrical structure, and we show how to apply them to two other GIS-problems (line simplification and a certain generalization task).

Adding more cartographic rules to the problem is done by modifying the geometrically local optimizer, a new operator that is introduced to explicitly generate candidates for building blocks. Building blocks in the map-labeling problem are geometrically local regions of the map which have a good labeling. Thus, a new cartographic rule can be expressed by adding it to the geometrically local optimizer.

We used theoretical models for the population size (the gambler's-ruin model)

and the number of generations before convergence to describe the scale-up behavior (the relation between input size and computational cost) of our GA. These models have been able to predict the scale-up behavior of GAs for artificial problems quite accurately. As far as we know, this is the first time they are applied to analyze a GA for a real-world problem. We were able to satisfy the conditions for these models largely because of our use of the geometrically local optimizer. For example, one of the most important conditions states that the fitness function should be additively decomposable (see Section 2.1.4, page 23). Since we were able to keep the fitness function simple by putting the cartographic rules in the geometrically local optimizer, this condition holds for our GA.

The rest of this thesis is organized as follows. We start by providing a tutorial on GAs in Chapter 2 for those for whom GAs are new or who want a refresher. A GA for the "basic" point-feature map-labeling problem (without the additional cartographic constraints) is designed in Chapter 3. In this chapter we describe the core of the GA, which we designed carefully to make it robust and extendible. We also introduce the geometrically local optimizer, which is used to incorporate cartographic rules into the algorithm, and discuss its advantages. From this result, we continue in three different directions.

Firstly, we discuss the efficiency of the GA in Chapter 4. The scale-up behavior of any algorithm is very important for its practical significance. We show that the GA has quadratic scale-up behavior, which means that doubling the input of the algorithm requires four times the amount of computational effort. As stated above, this is, as far as we know, the first time that GA theory has been successfully applied to describe the behavior of a GA for a real-world problem.

Secondly, in Chapter 5, we present a framework to design GAs for GIS problems. We show how the techniques we used to design the GA for point-feature map labeling, which were derived from theoretical insights, can be used in a broader context. We extend the GA for point labeling in several ways to demonstrate how to incorporate additional cartographic rules. We also present two other GIS problems (line simplification and a generalization task) as case studies of our approach.

Lastly, in Chapter 6, we expand our framework for map labeling by adding another feature type. We extend our GA to also handle line features with multiple labels that are placed at proper intervals along the line feature.

We conclude in Chapter 7.

Parts of this thesis have appeared elsewhere.[100, 102, 101, 98, 99]

CHAPTER 2

Genetic Algorithms

In the following chapters we will develop several GAs. In this chapter, some understanding of the way GAs work is provided. This background knowledge is given here for those who do not possess it, or who do not mind a quick refresher. Readers that are already familiar with GAs may want to skip this chapter.

A genetic algorithm is a heuristic solver for optimization problems. Its inspiration comes from the theory of Darwinian evolution by means of natural selection. The interest in biology by computer scientists can easily be understood. In biology incredibly complex structures have arisen that are capable of solving the problems inherent to life and survival very well. Biologists have also identified the mechanism that leads to these remarkable feats of problem solving: evolution by means of natural selection. Computer scientists on the other hand have encountered problems that are so hard that it is believed that no fast (polynomial time) algorithm can be found that solves them optimally. They are in need of ways of (approximately) solving such problems in a reasonable amount of time. It seems interesting, therefore, to apply the highly successful mechanism of natural evolution to computer-science problems. Today, several types of *evolutionary algorithm* are known, for example genetic algorithms, evolution strategies[7,80,84] and evolutionary programming.[27,26] In this thesis, we will use genetic algorithms, as originally described by Holland.[44] Good textbooks on GAs[32,67,65] also cover the material from this chapter.

This chapter is organized as follows. First, in Section 2.1, we will introduce the basic genetic algorithm. It consists of several components, which will be

discussed in turn. In Section 2.2, we will briefly introduce some theory on GAs that explains why GAs are able to solve many problems efficiently.

2.1 The algorithm

In biology, organisms have to compete for scarce 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. Over time organisms will become progressively fitter, until the environment changes (humans start breeding bonsai trees) or a physical boundary is reached (trees can only grow so large until they collapse under their own weight).

Another 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 is likely to propagate through the population. Traits can also be combined in a single individual by swapping pieces of DNA in the process of reproduction. This is called crossover.

Summarizing, the key elements of Darwinian evolution are the following:

- Traits (like height, shape of foliage, etc.) are genetically determined and can be passed on from parents to offspring.
- Certain traits are important for the success in reproduction. In the competition for limited resources an organism can be more or less fit, compared to other organisms of the same species. The process of selective reproduction as a result of differences in fitness is called *natural selection*.
- There is a source of genetic variation that produces new traits. Mutation introduces random variation, and crossover provides new combinations of genes.

Organisms with the best traits are able to reproduce more, and pass the traits on to their offspring. New traits are continually introduced and subsequently subjected to natural selection. As a result, traits that are beneficial get propagated through the population, and the species evolves towards a state of adaptation to its environment.

Even though this view of natural evolution is much simplified, it serves as an introduction to the *artificial evolution* as used by a GA. In a GA, not organisms, but solutions for a certain problem evolve. As a running example of a very simple problem, we consider the following, artificial problem. Given $2n_p$ binary variables, $x_1 \ldots x_{2n_p}$, we want to maximize the number of pairs x_i, x_{i+n_p} such that $x_i = x_{i+n_p}$. We will call this problem the *corresponding-bits problem.* Since it is so simple—setting all bits to 1, for example, gives an optimal solution—it will not distract us from demonstrating how it can be solved using a genetic algorithm. (Good problems to solve with a GA are usually combinatorial in nature, and have a search space with a complex structure. A GA to solve this problem would, obviously, be overkill.)

The problem is formulated using a *cost function* that attributes a cost (to be maximized or minimized) to every *solution* of the problem. A solution gives a *setting* of the *problem variables* of the cost function. A setting is an assignment of values to the problem variables. Note that a solution doesn't have to be optimal. For our running example, the cost function becomes $f_{cost}(\mathbf{x}) = \sum_{i=1}^{n_p} match(x_i, x_{i+n_p})$, where $\mathbf{x}$ denotes a solution and $match(x_i, x_j) = x_i \cdot x_j + (1 - x_i) \cdot (1 - x_j)$. An optimal solution $\mathbf{x}^*$ has $f_{cost}(\mathbf{x}^*) = n_p$.

In artificial evolution we use the cost function to specify fitness and call it the *fitness function*. It is possible that the problem specifies constraints on the settings the problem variables may have. If a solution fulfills these constraints, it is called *feasible*.

To make artificial evolution work, similar elements as for natural evolution are needed:

- A solution is encoded, for example as a string of values, similar to DNA. Offspring is made by copying parts of the strings from the parents.
- Each solution has a certain fitness, which specifies how well the solution solves the problem. Selection is mimicked explicitly using a selection scheme, which chooses the individuals with the highest fitness for reproduction.
- Genetic variation is provided by mutation and crossover operators similar to those of natural evolution.

Following the analogy with biology, we call the string of values that specifies a solution a *chromosome* (a strand of DNA) consisting of a number of *genes*. Each gene stores one from a finite number of values called *alleles*. The fitness of such a solution is given explicitly by the fitness function.

A GA evolves a population of solutions by repeated selection and mating. Selection picks individuals out of the population for which the fitness function yields

a relatively high result. These individuals are placed in an intermediate population called the *mating pool*, which has the same size as the original population. Selection is done with replacement. As a result, very fit individuals may have multiple copies in the mating pool. The members of the mating pool 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. Two (different) members of the mating pool are randomly chosen and removed from the pool. They are then mated, and produce two children. After all children from all pairs of parents are generated, they constitute a new population that replaces the old population.

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 population are similar in quality and hardly any progress is made. If the GA is designed correctly, the best solution from the final population should be near optimal.

We will refer to the analogy with natural evolution at various points in this chapter to make concepts intuitively clear. It should be understood, however, that the analogy should not be taken too far. For example, in biology a population is almost always largely converged and well adapted to its environment. Further adaptation occurs when the environment changes and a new niche is discovered. In artificial evolution the environment (simulated by the fitness function) is static and the GA terminates when the population is converged.

Pseudocode for a genetic algorithm is given as Algorithm 1, which takes as input the population size n, the probability Pr_c that crossover is performed, and the probability Pr_m that mutation is performed. The size of the population n is assumed to be even. Since many researchers have been inspired by natural evolution differently, there are many variants of the genetic algorithm. Our description of the GA is based on the algorithm as developed by Holland[44] and discussed by Goldberg.[32] This algorithm uses the following subfunctions:

INITIALIZE(*ind*): Initialize an individual.

SELECT(*Pop*): Select an individual from the population *Pop* with above-average fitness.

CROSSOVER(p_1, p_2): Perform crossover on p_1 and p_2, yielding children c_1 and c_2.

MUTATE(*ind*, g): Mutate gene g in the solution *ind*.

TERMINATE(*Pop*): Evaluate the state of the population and decide whether to stop the algorithm.

```
1:  generate Pop with n chromosomes                          } Construct initial
2:  for ind ∈ Pop do                                         } population
3:      INITIALIZE(ind)
4:  repeat
5:      Pop_mat ← ∅                                          } Fill mating pool
6:      repeat                                               } with selected
7:          ind ← SELECT(Pop)                                } individuals
8:          add ind to Pop_mat                               }
9:      until |Pop_mat| = |Pop|
10:     Pop ← ∅                                              } Generate new
11:     repeat                                               } population from
12:         randomly choose and remove two                   } mating pool
              individuals p_1, p_2 from Pop_mat              }
13:         with probability Pr_c do                         }
              {c_1, c_2} ← CROSSOVER(p_1, p_2)               }
14:         otherwise: c_1 ← p_1; c_2 ← p_2                  }
15:         for child ∈ {c_1, c_2} do                        }
16:             for each gene g in child do                  }
17:                 with probability Pr_m do MUTATE(child, g) }
18:             add child to Pop                             }
19:     until |Pop_mat| = 0
20: until TERMINATE(Pop)
21: report best individual in Pop
```

Algorithm 1: The standard genetic algorithm.

All parts of the algorithm are further explained in the following subsections. We first need something to evolve, so the algorithm starts by generating and initializing a new population. How to set the population size n is discussed in Subsection 2.1.1. Each solution in the population is a string of values (see Subsection 2.1.2). After a solution is generated, it has to be initialized to give it proper settings (see Section 2.1.3). What follows is the main loop in which a population evolves to a new population. First, a mating pool is filled with highly fit individuals from the old population. A fitness value is given by the fitness function, discussed in Subsection 2.1.4. How individuals are selected is described in Subsection 2.1.5. Parents are paired by randomly choosing two members of the mating pool. Parents can either produce exact copies of themselves, or perform crossover (see Subsection 2.1.6) to yield a new combination of their genes in their children. The offspring can subsequently be mutated (see Subsection 2.1.7), before they are placed in the new population. After some time, the population will only contain solutions which are very much alike, and little progress is made. A termination criterion that evaluates the state of the population then signifies the main loop should end. The choice of termination criterion depends on what the user wants. Examples of termination criteria are a limit on the amount of computation, a goal for the fitness of the best solution in the population, or a measure of convergence. The latter can be done by examining the differences in fitness in the population, or by following the increase in fitness over time.

The following subsections will explain the various components of the genetic algorithm in more detail. Descriptions will, for the sake of simplicity, appeal to intuition, and will be exemplified with the corresponding-bits problem. In Section 2.2, more formal concepts will be introduced.

2.1.1 Population size

The population size determines the amount of information the GA can exploit. Each member of the population is a solution, and each solution is a point in the search space. A larger population means the GA's view of the search space is more detailed, and deductions about its structure are more likely to be correct. In other words, larger populations are likely to give better final solutions. On the other hand, maintaining a large population takes a lot of computational effort. A trade-off has to be made between the quality of final solutions and the amount of computation one is willing to do. Since a GA deteriorates gracefully in the quality of its solutions, there is some slack in the setting of the population size.

How does one set the population size? In general, three different methods are used to set the population size for a specific run of the GA:

1. Reckoning: the user gets a feel of how large the population size should be,

simply based upon experience when using the GA. This is by far the most common method.

2. Analysis: by modeling the dynamic interactions of the algorithm one can find a population-sizing equation which gives recommendations on the size of the population. Examples of this method are in papers by Goldberg et al.[35] and Harik et al.[40] While this would seem to be the ideal solution, doing the analysis is very hard due to the complex behavior of genetic algorithms. As a result, analysis has so far only been done of GAs for artificial problems with known properties. One of the contributions of this thesis is to perform the first such analysis for a GA solving a real-world problem (namely the map-labeling problem)—see Chapter 4.

3. Adaptive sizing: a method that completely eliminates any user interaction is adaptive population sizing. Examples of this approach are described in papers by Smith and Smuda,[85] Sawai and Kizu[82] and Harik and Lobo.[42] Especially the last method can be directly applied as an immediate, off-the-shelf solution to avoid setting the population size. It works by running multiple GAs with different population sizes in parallel. GAs with smaller population sizes get more computation time, until GAs with larger populations find better solutions. In the worst case, the time complexity of the number of evaluations becomes $E = O(E_{opt} \log E_{opt})$, where E_{opt} denotes the number of evaluations that a GA using an optimally sized population would take.[73,62]

2.1.2 Encoding

One of the first decisions that has to be made in the design of a GA, is what encoding to use. As described before, solutions are encoded as chromosomes, strings of genes which can hold any of a finite number of alleles. Solutions give settings for the problem variables, and it has to be decided how to store a solution as a string of values. How long should the string be? Which alleles can be stored at a gene? How are genes and problem variables related? The set of alleles the encoding will use is called the *alphabet* of the solution. The cardinality of the alphabet is the number of alleles. An encoding can be as simple as a concatenation of the problem variables, or as elaborate as coding a tree structure in a string of integers.[81] In the problem of matching corresponding bits, each problem variable can be encoded by a single gene that stores a bit. The encoding becomes a bit string of $2n_p$ genes long, with the alphabet $A = \{0,1\}$.

The choice of the encoding is important for the success of the search of the GA. There exists an interdependency between the choice of the encoding and

the design of the operators that act on it (especially crossover). Depending on the crossover operator used, it is probably a bad idea to use an encoding for the corresponding-bits problem that follows the same ordering as for the problem variables. We have pairs of two variables, x_i and x_{i+n_p}, that depend on each other, because they have to match. If the same ordering is used in the encoding, then there are always $n_p - 1$ genes between two genes that "belong together". It is likely that crossover will transfer their settings to different children. This will become more clear when we discuss crossover. For now, it is sufficient to realize that variables that have a mutual dependency should be kept close together in the encoding. Therefore, we encode a solution for the corresponding-bits problem as a string of corresponding pairs that have to match: $\mathbf{x}' = x_1 x_{1+n_p} \ldots x_{n_p} x_{2n_p}$.

The *building block hypothesis*[32] states that GAs find good solutions by combining smaller parts called *building blocks*. Such a part consists of only a few genes with (near-)optimal settings, and gives a large contribution to the fitness of the chromosome. It is crucial to the success of the search that these parts are inherited by the children, since that is what makes them fit. If crossover, for example, splits a building block that existed in a single parent over two children, the building block is lost. The encoding and crossover should be designed together, to ensure good inheritance of building blocks. In the running example, a building block is a combination of a matching combination of bits. In other words, building blocks consist of specific combinations of genes with specific values. We will discuss this in more detail in Section 2.2.

A nice property that the encoding can have is when it only expresses feasible solutions. If that is not possible, the GA needs other ways to cope with infeasible solutions.

2.1.3 Initializers

The initializer provides the initial population, which will evolve during the course of the GA. A good initial population has to be diverse, to give the GA as much raw material to process as possible. The population is often generated by providing randomly chosen alleles for all genes, for each chromosome in the population.

An intuitive way of looking at GAs is by picturing the search space as a landscape. The *fitness landscape* is derived from the search space, with an additional dimension. The additional dimension is used for giving each search point a "height", its fitness value. If the encoding has two genes, the fitness landscape is a surface in a three-dimensional space with mountains, ridges, pits, and valleys. In a maximization problem, a *local optimum* is a point where all points in the neighborhood are lower (their fitnesses are smaller). A *global optimum* is a point with a height greater than or equal to all other points. The GA tries to find a highest

point on the landscape.

Since there usually is no a-priori knowledge about the location of the largest peak in the landscape, the initial population has to cover the whole landscape to make sure enough information about all regions in it is provided. Using these points, the GA deduces where the highest regions are, by selecting solutions with above-average fitness. Crossover and mutation provide new points to guide the search.

If the problem is defined with constraints that make certain regions of the search space infeasible, the initializer has to take care that each solution is feasible, or the GA will have to cope in some other way with infeasible solutions.

The initializer can also be seen as a source of building blocks. Additionally, these can be formed during the course of the algorithm, which becomes more likely if the initializer provided enough variation.

A proper initialization for the running example is done by generating a random bitstring. These random bitstrings will contain building blocks, generated by chance, at different places. The task of the GA is to bring the building blocks together in a single individual.

2.1.4 Fitness function

A GA uses a cost function to evaluate how good a solution is. In GA terminology, the cost function is called the fitness function. Note, however, that the fitness function is not necessarily completely determined by the problem definition. For example, a GA might deal with infeasible solutions by including a *penalty function* in the fitness function, to reduce their fitness. For most problems, the major part of computation in the standard genetic algorithm is spent in evaluating the fitness function. Therefore, it is important to design the fitness function such that it can be evaluated efficiently.

A special kind of fitness function is the *additively decomposable function* (ADF). Such a function can be expressed as the summation of the contributions of the parts of the solution. More precisely, the function is a summation of partial fitness functions that only depend on a few genes each. For example, given a solution $\mathbf{x} = x_1x_2x_3x_4x_5$, the function $f_{fit}(\mathbf{x}) = f_1(x_1x_2x_3) + f_2(x_2x_3x_4) + f_3(x_3x_4x_5)$ is an ADF. If the different functions $f_i(\cdot)$ all depend on different genes, the ADF is called *separable*. If each partial fitness function is defined over the same range, the ADF is called *uniformly scaled*.

Note that the ADF is different from the fitness function with weighting factors that we described in Section 1.1 on page 9, which uses subfunctions that all provide a global measure of the solution, whereas the ADF uses subfunctions that depend on only a few genes. The properties of an ADF will prove essential to the analysis given in Chapter 4.

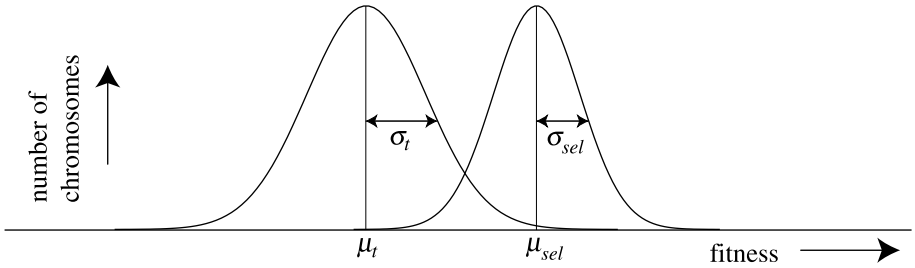

Figure 2.1: The effect of selection on a normally distributed population.

The fitness function for the running example of matching corresponding bits takes as input the encoding we described in Section 2.1.2, and is therefore different from the cost function given before: $f_{fit}(\mathbf{x}') = \sum_{i=1}^{n_p} match(x'_{2i-1}, x'_{2i})$. Note that this is a separable, uniformly scaled ADF.

2.1.5 Selection schemes

A GA uses a selection scheme to give a bias to good solutions in the population. In nature, the same thing happens when organisms compete for a limited resource. Those that are best equipped for obtaining the resource have a better chance to reproduce. In a GA this competition has to be simulated in the selection scheme. Selection can be characterized by the strength of the bias it gives to good solutions. This is called the *selection pressure*. It varies between randomly choosing an individual (zero pressure) and always selecting the best individual (maximal pressure). A balance between these two extremes has to be struck.

Selection pressure can be quantified by specifying the *selection intensity*. If μ_t and σ_t are the mean and the standard deviation of the fitness of the population at generation t, respectively, and μ_{sel} is the mean fitness of the selected individuals, then the selection intensity I_t at generation t is defined as[11] (see Figure 2.1):

$$I_t = \frac{\mu_{sel} - \mu_t}{\sigma_t},$$

If the selection intensity I_t is equal for all t, it is just denoted I and the selection pressure is called constant. Note that for a population whose population is normally distributed with zero mean and unit standard deviation, the selection intensity is equal to the expected fitness of the selected individuals.

Fitness-proportionate selection[44] gives each individual a probability for selection that is proportional to its fitness. For example, if there are only three members in the population, with fitnesses of 1, 1, and 2, the former two have a 25% chance of being selected, while the latter individual has a 50% chance of being selected.

One problem with this scheme is that it is sensitive to the actual fitness values used. As a result, when fitness values become more similar (at the end of the run), the selection pressure drops. The opposite effect occurs at the start of the run, where a superior solution can get an overly large bias, causing the GA to converge prematurely.

Another well-known selection scheme, called *tournament selection*,[10] works by holding a competition among several randomly chosen members of the population. The one with the highest fitness wins and is selected. This method has constant selection pressure, because instead of using differences in absolute fitness values, it uses differences in the *ranking* of fitnesses. As a result, it avoids the problem with variable selection pressure of fitness-proportionate selection. The strength of the selection pressure can be tuned by making the tournament size (the number of individuals taking part in the competition) smaller or larger. A problem with this scheme (which also occurred with fitness-proportionate selection) is that the genes of a particularly good individual can be lost if it produces inferior children that will not get selected in the next generation. One way to preserve highly fit individuals (also called *elitism*[16]) is to copy the best k individuals from the old population to the new population.

In this thesis we will mostly use the *elitist recombination scheme*.[96] This scheme combines selection, recombination (that is, crossover), and replacement. The GA no longer explicitly calls a procedure for selection, but takes the form of Algorithm 2, which takes as input the population size n and the probability Pr_m that mutation is performed. Parents are randomly chosen from the population, with no bias. Crossover produces two children. From this family of four—two parents and two children—the two best ones are chosen and replace the parents. In the case of ties, children precede parents. The selection pressure results from the biased replacement. See Figure 2.2 for an example where one child replaces a parent.

This selection scheme has the following advantages:

- The scheme is conceptually simple, easily implemented, and simplifies the structure of the GA.
- It preserves good solutions by having elitism on the family level.
- The parameter Pr_c, which denotes the probability that crossover is applied, is set to 1.0, because there is no danger of losing fit parents.*
- The selection pressure is constant and can be tuned by using tournament selection for one parent.[92]

*The parameter Pr_c is further discussed in Sections 2.1.6 (page 28) and 2.2.2 (page 33).

1: generate *Pop* with n chromosomes
2: **for** $ind \in Pop$ **do**
3: INITIALIZE(ind)

} Construct initial population

4: **repeat**
5: randomly choose two individuals p_1, p_2 from *Pop*
6: $\{c_1, c_2\} \leftarrow$CROSSOVER(p_1, p_2)
7: **for** $child \in \{c_1, c_2\}$ **do**
8: **for** each gene g in *child* **do**
9: with probability Pr_m do MUTATE$(child, g)$
10: replace p_1, p_2 with the best two from $\{p_1, p_2, c_1, c_2\}$

} Elitist recombination scheme

11: **until** TERMINATE(*Pop*)
12: report best individual in *Pop*

Algorithm 2: Genetic algorithm with elitist recombination.

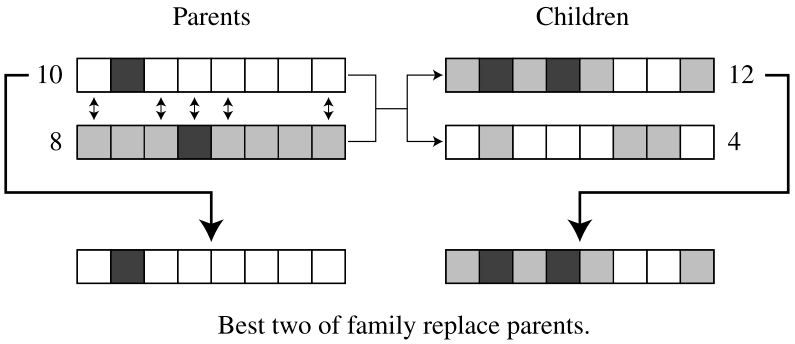

Figure 2.2: Elitist recombination, with building blocks darkly shaded. The numbers denote the fitness of the strings.

The selection pressure of a selection scheme should be carefully set. A selection pressure that is too high or too low can have undesirable effects.[36,94] If the selection pressure is set too low, *genetic drift*[4,37] will take over the convergence of the population. Genetic drift is essentially a random walk with absorbing barriers. Because with zero selection pressure samples are randomly chosen from a finite set, stochastic effects will build up until the population has converged, by chance, to a single individual.

The other extreme—a selection pressure that is too high—is just as bad. A disproportionate large bias is given to a solution that has above-average fitness. Presumably, that solution had a higher fitness because it contains a building block. Then why is it so bad to have a high selection pressure? Firstly, in order to allow crossover to produce new combinations of building blocks, the diversity of the population should not be depleted too quickly. It takes some time to exchange building blocks and propagate them through the population, and convergence should not be reached before a near-optimal solution has been constructed. Secondly, the solution that was selected may contain a building block, but it can also contain parts that are inferior. These inferior parts will be propagated with the building block because they are on the same string, and fitness evaluation is on the level of strings. This is called *hitchhiking*,[68] since the inferior part travels along with the building block. The effect is stronger when different building blocks have widely different fitness contributions.[97]

2.1.6 Crossover

Selection only *exploits* the information present in the population. The GA also needs a mechanism of obtaining new information (*exploration*), which can subsequently be exploited. Exploration is done by the operators crossover and mutation. With crossover, we hope that each parent possesses a part of the optimal solution, which will be combined in the children. Another word for crossover is *recombination*. In biology, crossover occurs by swapping pieces of chromosomes. This inspired the one-point crossover operator.[44] It is depicted in Figure 2.3 (top). The one-point crossover randomly chooses a point on the string, and swaps the parts of the parent strings before that point. Two-point crossover[16,86] (in the middle of the figure) is similar, but uses two randomly chosen points. Uniform crossover[91] (at the bottom of the figure) tosses a coin for each gene and swaps the genes depending on the outcome.

Which crossover is best? It depends on the structure of the problem. As we explained when we discussed the encoding, some genes “belong together” because their optimal values should be transfered together. After initialization, the building blocks are dispersed in the population. Somehow all building blocks have to end up together in a single individual to compose a (near-)optimal solution.

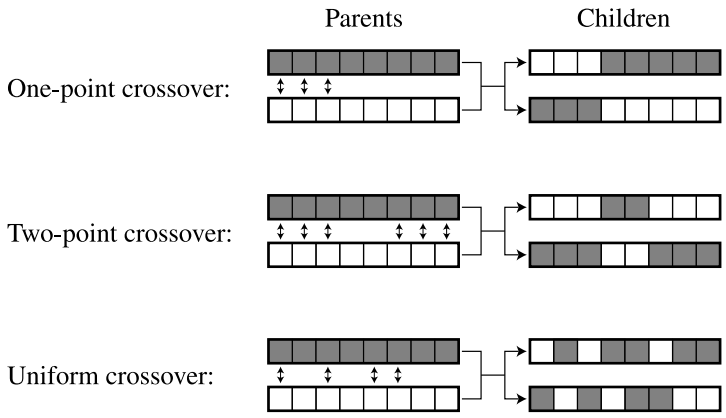

Figure 2.3: Crossover operators.

Crossover has to perform two goals at once, which can be contradictory. Firstly, it has to *mix*[94] the building blocks. Children become better by inheriting building blocks from both parents. Mixing is the shuffling of the building blocks present in the parents to get new combinations in the children. Secondly, it has to do this with minimal *disruption*. If a building block is composed of two genes and each gene is distributed to a different child, then it is possible that the building block is not present in either child; the building block is said to have been disrupted. Different crossover operators make different assumptions about the structure of the problem. One-point crossover preserves building blocks well (provided that their genes are close together in the encoding), but mixes rather slowly. It also has a positional bias against genes that are far apart. The first and the last gene will almost always end up in different children. Two-point crossover is similar to one-point, but alleviates that problem. Uniform crossover mixes very aggressively, but can also be very disruptive.

Disruption could be minimized if it was known which genes, together, make up a building block. For example, when a building block consists of a single gene, uniform crossover can never disrupt it, and is therefore the best choice, since it has the best mixing properties. In our running example with corresponding bits, the two genes that encode the matching bits “belong together”. They can hold four values, and there exist two building blocks: the setting where both bits are set, and the one where both bits are cleared. Unfortunately, in realistic problems it is often not known which bits belong together (the so-called *linkage*).

The purpose of the parameter Pr_c is to tune the amount of mixing and disruption that occurs due to crossover. Crossover should be applied often enough to make sure good mixing occurs, but not too much to avoid unnecessary disruption. Empirical studies[16,83] suggest values between 0.6 and 0.9. When we use the elitist recombination (ER) scheme, however, we can safely set Pr_c to 1.0. In the

ER-scheme, elitism (the preservation of good individuals) is built in by the biased replacement. This means that a disrupting crossover can produce an inferior child, but the child will not replace the parent.

2.1.7 Mutation

Another mechanism for exploring the search space is given by mutation. Each gene is mutated with probability Pr_m. This is done by storing a randomly chosen allele at the gene. Mutation is intended to provide diversity and reintroduce possibly lost alleles. Diversity can be used in crossover to create new building blocks. Additionally, mutation can get lucky and create a building block immediately.

The trade-off that has to be made to set Pr_m is between the benefit of "getting lucky" and the disruptive effect mutation usually has (when it isn't lucky). Since a GA works with a population, a little disruption is acceptable because the disrupted part will probably also occur in some other individual. A good value for Pr_m is $1/l$,[70,5] where l denotes the length of a chromosome (the number of genes).

For our running example, we can use bit flipping as the mutator. A gene is mutated by flipping (changing from 0 to 1 and vice versa) the bit that it holds.

2.2 Theory

The previous section introduced the genetic algorithm and discussed its various components. Some ideas about the workings of the GA were introduced, for example the concepts of building block, mixing and disruption. In this section, we will investigate deeper why GAs work and look at these matters more formally.

We start by introducing the concepts of schemata and partitions. This makes it possible to define linkage. Linkage determines what GAs "should be" processing—that is, which combinations of genes should be protected against disruption during crossover. After that, we discuss the schema theorem, which states what a GA actually *is* processing. The goal of GA design is to make these two perspectives match.

2.2.1 Schemata and linkage

A chromosome is a string of genes, each of which stores one from several possible alleles. More formally, we can define a chromosome as a string of values taken from an alphabet A:

Definition 1 ***Chromosome***

A chromosome *is a string* $\mathbf{x} \in A^l$*, where A is called the alphabet and l denotes the length of the chromosome.*

If we want to talk about *similar* chromosomes, we need a way of specifying similarity. This is done with a *schema* (*schemata* in plural). Schemata are strings of the same length as the chromosomes. They use the same alphabet, but extended with the "don't care"-character, denoted by "#". A chromosome *matches* a schema when for each gene, they either have the same allele or the schema has the special value "#" for that gene.

Definition 2 ***Schema***

A schema *is a string* $\mathbf{s} \in (A \cup \{\#\})^l$. *A chromosome* $\mathbf{x}$ *matches a schema* $\mathbf{s}$ *if, for all* $1 \leq i \leq l$, *we have:* $x_i = s_i$ *or* $s_i = \#$.

For example, given a binary alphabet, the chromosome 010110 matches the schema 01##10. A schema can also be seen as defining a set that holds the chromosomes that match it. The fitness of a schema is defined as the average fitness of all the strings that match it.

Schemata can be further classified in *partitions*:

Definition 3 ***Partition***

A partition *is a string* $\mathbf{p} = \{\mathrm{F}, \#\}^l$. *A schema* $\mathbf{s}$ *matches a partition* $\mathbf{p}$ *if, for all* $1 \leq i \leq l$, *we have: either* $p_i = \mathrm{F}$ *and* $s_i \in A$, *or* $p_i = s_i = \#$.

A partition is also a set that holds the collection of schemata that match it. The character "F" denotes a fixed position, where an element of the original alphabet is used. A schema matches a partition when they have the special value "#" for exactly the same genes. For example, the partition FF#### holds the following four schemata: 00####, 01####, 10####, and 11####. The idea of a partition is useful to formalize what before was called "a part of the solution" or a "subsolution".

Obviously, many different partitions of an encoding can be considered. The question is which ones are important. In the previous section, we have demonstrated with the running example of corresponding bits that some genes "belong together". When this is the case, we say that these genes are *linked*. The concept of linkage can be formalized using the so-called Walsh decomposition.[33,31,30,93] For example, consider the corresponding-bits problem, when the number of pairs is one ($n_p = 1$). Recall that the encoding consists of two bits, which can hold four values: 00, 01, 10, and 11. Using a Walsh decomposition, any fitness function that is defined on two genes can be rewritten in the following form:

$$\begin{aligned} f_{fit}(x_1x_2) = \quad & \omega_0 + \\ & \omega_1 \cdot sign(x_2) + \\ & \omega_2 \cdot sign(x_1) + \\ & \omega_3 \cdot sign(x_1) \cdot sign(x_2)\,, \end{aligned}$$

x_1	x_2	$f_{fit}(x_1x_2)$	
0	0	$\omega_0+\omega_1+\omega_2+\omega_3=1$	$\omega_0=\frac{1}{2}$
0	1	$\omega_0-\omega_1+\omega_2-\omega_3=0$	$\omega_1=0$
1	0	$\omega_0+\omega_1-\omega_2-\omega_3=0$	$\omega_2=0$
1	1	$\omega_0-\omega_1-\omega_2+\omega_3=1$	$\omega_3=\frac{1}{2}$

Table 2.1: The corresponding-bits problem for $n_p = 1$. Left: chromosomes and their fitnesses. Right: the Walsh coefficients.

with

$$sign(x) = \begin{cases} 1 & \text{if } x = 0 \\ -1 & \text{if } x = 1\,. \end{cases}$$

The constants ω_i are specific for a certain fitness function and are called the *Walsh coefficients*. They signify the linkage between genes. Table 2.1 shows the fitnesses of the strings, expressed as the sum of the Walsh coefficients. Note that $\omega_3 = \frac{1}{2}$, which shows that there is a non-linear interaction between the first and the second gene. For a chromosome of length l, the fitness function is a summation of 2^l terms, where each term contains a Walsh coefficient. However, if there is no linkage between genes, the Walsh coefficient is zero. A full treatise of Walsh decompositions and linkage is beyond the scope of this thesis. What is important to remember is the intuition behind linkage, namely that genes are linked if their optimal setting can only be found by looking at them together, as a result of the non-linear interactions between genes.

This helps us in understanding why problems can be difficult. Good solutions are found by searching for the best schemata defined over genes that are linked together. However, linkage can be weak or strong. If it is weak, a relatively small fitness contribution can be sacrificed by allowing schemata defined on weakly linked genes to be disrupted during crossover. The cost is that the final solution found will be worse, but it will be found faster. Suppose, for example, that the user is only satisfied with the optimal solution. It is possible that the only way this solution can be found is when schemata defined over all genes of the chromosome are not disrupted during crossover. In this extreme case, there is linkage between all genes, and chromosomes have to be kept intact. In other words, crossover can not be applied and we have to resort to enumerative search. Such problems are often called needle-in-a-haystack problems. If, on the other hand, the user is satisfied with a sub-optimal solution, the groups of genes which have to be considered collectively shrink in size. The main power of a GA comes from the fact that it can quasi-independently search for the best schema in all partitions defined over genes which are strongly linked. We will exemplify these ideas in

Chapter 4, where we look at the problem of finding a suitable population size, given a specified measure of required quality.

Finding the linkage of an encoding can be hard. Educated guesses about the linkage can influence the choice of one of the standard crossovers. If more insight in the structure of the problem exists, the crossover operator can be designed to be *linkage respecting*, by mixing on the boundaries of the partitions defined over strongly-linked genes. If no relevant domain knowledge exists a priori, one can attempt to learn the linkage,[56, 39, 41, 38, 72] and exploit it with a linkage-respecting crossover.

2.2.2 The schema theorem and adequate mixing

The *schema theorem* is a fundamental theorem that gives insight in the way a GA works. It offers a different perspective on the GA by considering how it processes schemata (instead of chromosomes):

Theorem 1 ***Schema theorem***[44]

Let $\mathbf{s}(t)$ *be the number of chromosomes matching a schema* $\mathbf{s}$ *in generation* t*, let* $\phi(\mathbf{s},t)$ *be the growth rate of the schema in generation* t *due to selection, and let* $\varepsilon(\mathbf{s},t)$ *be the decay rate of the schema in generation* t *due to disruption by crossover and mutation. Then the following holds for every schema* $\mathbf{s}$ *in the population:*

$$\mathbf{s}(t+1) \geq \mathbf{s}(t) \cdot \phi(\mathbf{s},t) \cdot \varepsilon(\mathbf{s},t)\,.$$

The schema theorem states that the number of chromosomes matching a schema will increase due to the effect of selection if the schema has above-average fitness, but it will decrease as a result of the disruptive effects of crossover and mutation. The inequality in the formula results from the fact that crossover can not only disrupt the schema, but it can also form new chromosomes matching the schema. The theorem holds for all schemata present in the population, in parallel. In other words, the number of chromosomes matching a certain schema will *increase* if the following holds (dropping identifiers for schema and generation for simplicity):

$$\phi \cdot \varepsilon > 1\,.$$

The final solution found by the GA is composed of highly-fit schemata that were processed during the run of the algorithm. These so-called *building blocks* are highly-fit, to be picked up by selection, and are composed of only a few genes, to avoid disruption. They get propagated through the population, as predicted by the schema theorem. The *building block hypothesis*[32] states that, eventually,

these building blocks come together in a single individual, resulting in a solution that should be close to optimal. This happens when the building blocks that the GA processes are the best schemata of partitions defined over genes with strong linkage. Such schemata guide the search towards a good solution. This condition is satisfied when the crossover operator respects the linkage by trying to keep linked genes together.

Careful reading of the above definition of building blocks (highly fit, unlikely to be disrupted) reveals a discrepancy with the intuitive notion of building blocks as we have used them until now (a "part of a (near-)optimal solution"). We implicitly assume the GA is designed well, and that the linkage of the encoding is respected by the crossover operator. As a result, we use these two explanations of a building block interchangeably, although it only holds when two conditions hold. Firstly, the GA uses a linkage-respecting crossover. Secondly, the GA is able to mix fast enough.

An additional condition that needs to be satisfied in order to find good solutions, is that *mixing* of building blocks is done *adequately*.[94] In a GA with no crossover or mutation, selection will eventually have filled the population with copies of a single individual. That individual may contain one or two building blocks, but it is unlikely that it is the optimal solution. The other building blocks needed were present in other individuals that didn't get selected. The exchange of building blocks during crossover, where building blocks from different parents end up in the same child, is called mixing. Mixing is adequate when building blocks are exchanged quickly enough to be able to converge to a (sub-)optimal solution containing building blocks from all partitions that are defined over strongly-linked genes. The amount of mixing can be tuned with the parameter Pr_c (the probability of crossover). If Pr_c is low, few building blocks are mixed, but disruption is low. If Pr_c is high, more building blocks will be mixed but the amount of disrupted building blocks increases. If selection pressure is high, mixing has to occur faster because there is less time before the population is converged. Therefore, Pr_c is also dependent on the selection pressure.

The question in GA-design is how to find an encoding and suitable operators such that building blocks are mixed adequately, disruption is minimized, all in as little time as possible. Knowledge about the linkage of the encoding can be used to design a crossover that mixes the elements of the right partitions and avoids disrupting building blocks.

CHAPTER 3

A GA for point-feature map labeling

In this chapter we will develop a genetic algorithm for the basic map-labeling problem:

> **The basic map-labeling problem:**
>
> Given are n_{feat} point features on a map. Each point feature has a label of fixed dimensions, and can place its label in one of four positions (see Figure 3.1): the top-right, top-left, bottom-right and bottom-left position. Produce a *labeling* for the point features, which assigns a position to each label, maximizing the total number of non-intersecting labels.

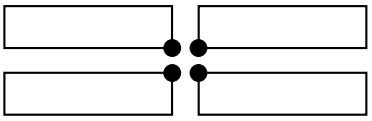

Figure 3.1: The four-position placement model.

This instance of the problem has been shown to be NP-hard.[64, 58, 28] We will not consider any other cartographic rules, although we will compare our GA against a simulated-annealing algorithm for the case of integrated name selection (where labels can be deleted and no intersections should remain). However,

since it is our goal to eventually offer an algorithm that solves map-labeling problems that include cartographic rules, we will design the GA such that it is easily extended with additional rules. How the GA can be extended and generalized will be discussed in Chapters 5 and 6. When comparing with the GA by Verner et al.,[107] we use the eight-position model, in which the four positions of Figure 3.2 are added to the four-position placement model.

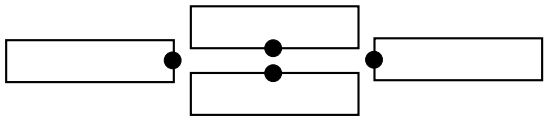

Figure 3.2: The additional four positions for the eight-position placement model.

In order to fully appreciate the design decisions we will make (most notably the inclusion of an extra operator), we take a bit of a round-about route to our destination. First, in Section 3.1, we will treat the problem like a black box: we pretend we do not know anything about the structure of the problem. As can be expected, this leads to a less than optimal GA. This GA will exemplify the difficulties in design that need to be overcome. In Section 3.2, we will analyze why the first attempt fell short, to gain an understanding of the problem. Standard GA theory (as described in Chapter 2) is used for the analysis. We will also look at other GAs for the same problem, and see how they cope with the difficulties. Finally, we present our own approach, based on the insights that we achieved in the previous discussion. Section 3.3 is devoted to a comparison of our algorithm with other algorithms from literature. In Section 3.4, we discuss the potential of the algorithms to be used in a realistic setting (in a GIS, for example). We conclude with Section 3.5.

3.1 A simple GA

In the absence of any knowledge about the structure of the problem, one can use the standard GA, described by Holland[44] (and in Section 2.1). It uses fitness-proportionate selection and one-point crossover. Suitable values for Pr_c (probability of crossover) and Pr_m (probability of mutation) were found in empirical studies.[16,83,70,5] Later studies[71,95,34,112] revealed that fitness-proportionate selection needed a scaling method or should be replaced with a selection scheme with constant selection pressure like tournament selection. Two-point[16,86] and uniform crossover[91] were invented and were found, under certain circumstances, to yield good performance.

The first GA developed in this thesis is a simple GA for the basic map-labeling problem. We will review the various design choices and examine the standard

options available. The reader is referred to Chapter 2 when unfamiliar concepts are encountered.

Encoding. The only prerequisite for the encoding is that it is composed of a string of values over some finite alphabet. In the case of map labeling, an encoding suggests itself almost immediately: a chromosome with a gene for each point feature, and an allele for each label position (see Figure 3.3). The alphabet is $A = \{1,2,3,4\}$, where 1 stands for a label in the top-right position, 2 for a label in the top-left position, 3 for a label in the bottom-left position, and 4 for a label in the bottom-right position.

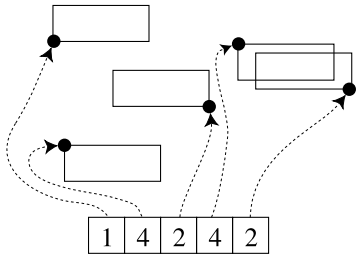

Figure 3.3: The encoding for a map.

Initialization. The initializer should produce solutions which cover the search space well. Here, we simply choose a random allele for every gene in every chromosome. Since no infeasible labelings exist, the solution will always be feasible and no special action is required.

Fitness function. We want to maximize the number of non-intersecting labels. A labeling with many non-intersecting labels is better than one which has less. Therefore, for the fitness function we can simply use the number of labels in the labeling that are not intersected by any other label. The function, to be maximized, becomes: $f_{fit}(\mathbf{x}) = free(\mathbf{x})$, where $free(\mathbf{x})$ is the number of non-intersecting labels, and $\mathbf{x}$ is a solution. For example, the fitness of the small map in Figure 3.3 is 3.

Selection. Next, we have to choose a selection method. Selection fills a mating pool with individuals that will produce the next population. Recall* that fitness-proportionate selection has variable selection pressure, so we do not expect good performance without a scaling method. Therefore, tournament selection is probably the best choice. We will try both selection schemes. The next question is

*See Section 2.1.5 on page 24.

what the selection pressure should be. Fitness-proportionate selection derives its selection pressure from the distribution of fitness values in the population. For tournament selection, we use a tournament size of two, which usually gives good results.

Crossover. After selection, the mating pool is paired and for each pair it is decided whether crossover should be applied. This is done with a probability of Pr_c; if no crossover is performed, exact copies of the parents are made. Values for Pr_c that are generally regarded to be acceptable range from 0.6 to 0.9. We choose $Pr_c = 0.7$.

Nothing can really be said about which crossover is better, if one doesn't know anything about the structure of the problem, as captured by the linkage of the encoding (explained in Chapter 2). Each crossover works well for different encodings and therefore for different problems. Therefore, we just have to try them out and pick the one with favorable performance.

Mutation. On the children (or copies) that result from crossover, mutation is then applied with a probability of Pr_m for each gene. Mutation is performed by randomly choosing an allele and storing it at the gene. A suitable value for Pr_m is $\frac{1}{l}$, where l is the length of the chromosome. In our case, $l = n_{feat}$. On average, one gene gets mutated in every chromosome in the new population.

Results

To quantitatively evaluate the performance of the GA, we use artificial maps with known properties. The maps used were randomly generated by placing 1000 cities on a square grid with a side length of 650 units. To remove boundary effects the map was folded into a torus. The cities had labels with dimensions of 30 by 7 units. Each city was placed on the map by randomly selecting a location for its point. If the label of the point can be placed in one of its positions without intersecting other labels, it is placed. Otherwise, we systematically search for another point where it is possible to place the label. Starting with the location that was randomly selected, points are examined in order of increasing x-coordinate. At the right edge of the map, the x-coordinate is reset to zero and the y-coordinate is increased. Similarly, at the bottom edge of the map, the y-coordinate is reset to zero.

After all points and their labels were placed, all labels were removed, and the GA was used to find a new labeling. This way we were certain that it was possible to place all labels without intersecting other labels, and that the optimum was always the number of cities on the map. See Figure 3.4 for an example of such a map.

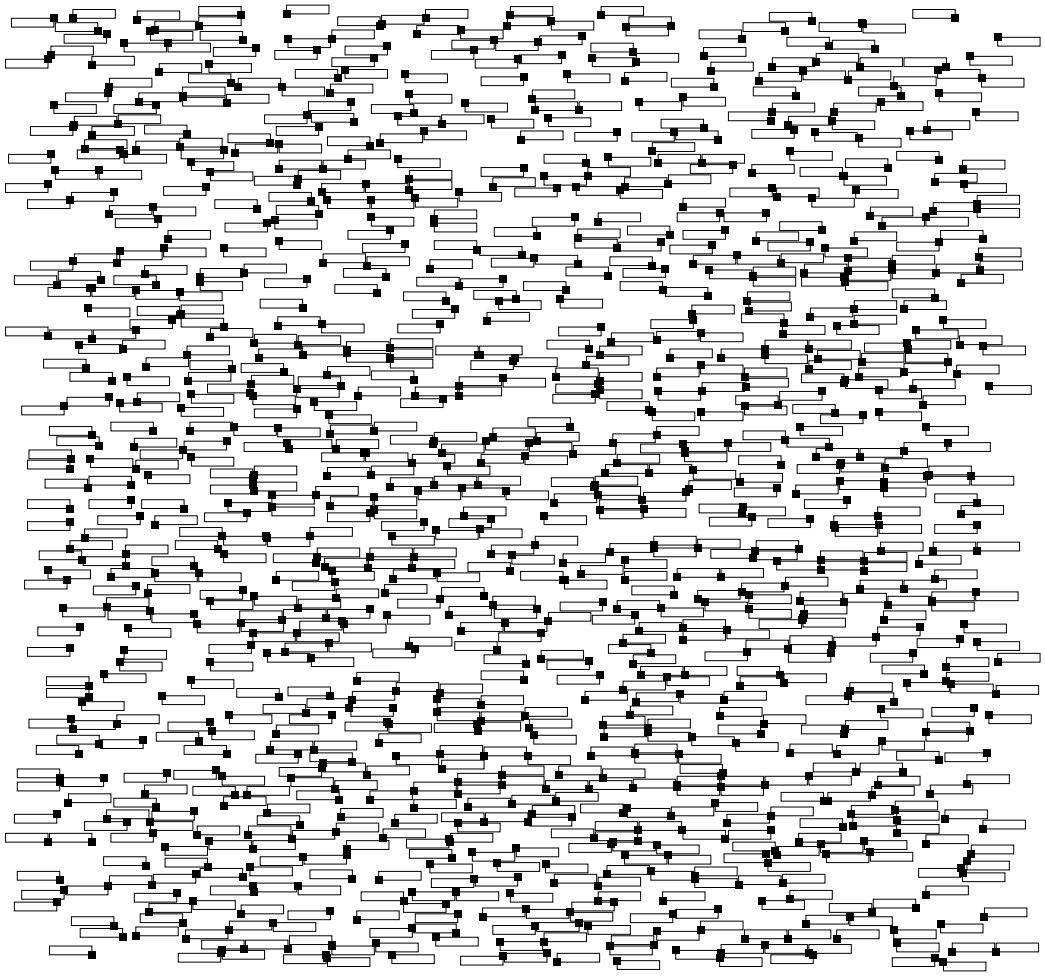

Figure 3.4: An example of a map with 1000 points where all labels can be placed.

In Figure 3.5, the results of four different versions of the simple GA are plotted. The population size for all GAs was 200. The GAs differ in the selection scheme used (tournament selection, or fitness-proportionate selection), and the crossover used (one-point or uniform crossover).

The figure shows the quality of the solution against the amount of computational effort spent by the algorithm. The former is measured as the percentage of non-intersecting labels (recall that it is guaranteed that it is possible to place all labels without intersections). The latter is measured in terms of the number of label-intersection tests performed by the algorithm; practically all the time the GA was running was spent doing label-intersection tests.* Each run of the GA was terminated when it converged (the average fitness of the population was equal to the fitness of the best individual), or when a limit of computational expense was reached. The limit used 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. Each of those runs records the average fitness of the population at regular points in time with the amount of label-intersection tests that have been performed up to these points. At the end of the runs, the population was largely converged; the fitness of the best individual in the population differed about 0.4% from the average fitness. For example, at the end of a typical run of the GA using uniform crossover and tournament selection the average fitness was 831.32 (that is, on average a solution placed 831.32 labels without intersections). The best individual at that point had fitness 834. The standard deviation of the final solutions is shown at the end of each graph.

Several interesting observations can be made of Figure 3.5. The most striking one is that no GA is able to find a solution close to the global optimum (1000 cities labeled, or 100%). Also interesting are the differences in the performance of the GAs. As was expected, fitness-proportionate selection does not work well without a scaling method. The difference between the crossover operators is significant for the GAs using tournament selection: the GA using uniform crossover performs better than the GA using one-point crossover.

Summarizing, the best GA we can make using standard techniques still performs poorly. Additionally, we notice that the choice of crossover operator can have a great impact on the performance of the GA. It is time to use the theoretical insights into how GAs work, to understand these differences and design a GA that gives better results.

*Note that a conflict for a city was checked by performing label-intersection tests between the label of the city and the labels of neighboring cities, the so-called rivals (discussed later). As a result, the same label-intersection test is performed twice for a pair of labels.

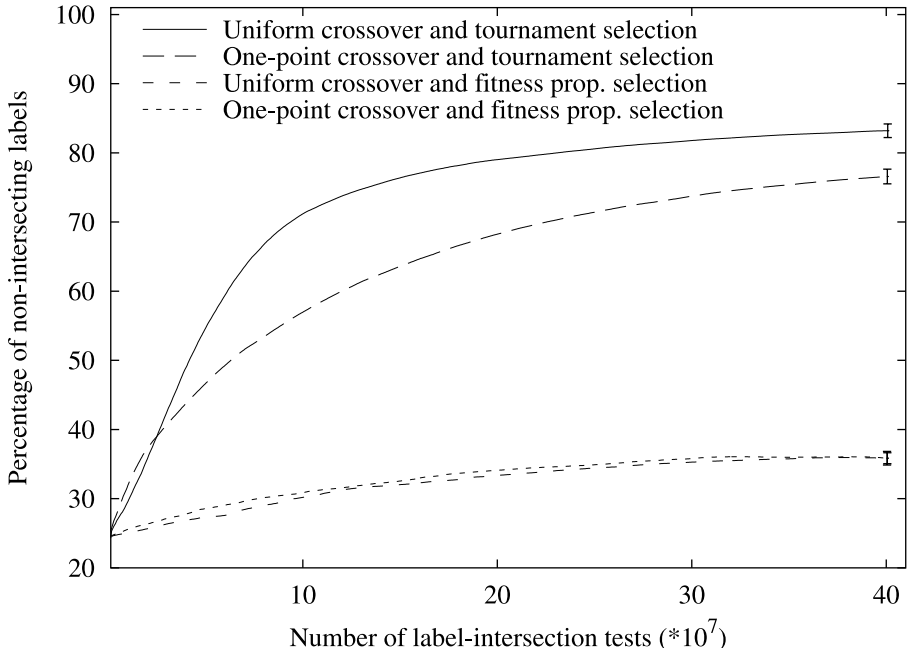

Figure 3.5: Comparison between different versions of the simple GA using different selection schemes (tournament and fitness-proportionate selection) and different crossovers (one-point and uniform crossover). $Pr_c = 0.7$ and $Pr_m = 0.001$.

3.2 Applying theoretical insights

The previous section deliberately turned a blind eye to theoretical insights into how GAs work (specifically concepts like building blocks, mixing and disruption). It employed a straight-forward approach, by applying the "standard" arsenal of operators. Often the situation is like this, when one doesn't have any idea about the structure of the problem. The use of GAs can then be an attractive option to pursue, because they are good at exploiting the structure present, even when the designer has no idea what it looks like. Note that a problem always needs to have some structure, from which inductions can be made, otherwise one has to resort to random search.[55] To obtain better performance, we have to incorporate our knowledge about the structure of the problem into the GA. This will help the algorithm to search better.

3.2.1 Analysis of the simple GA

Since we treated the map-labeling problem as a black box when we designed the simple GA, let's now inspect its structure to understand why the GA did not work so well. A point on the map has to place its label where it does not intersect other labels. Therefore, the placement of the label depends on the positions of other labels in the neighborhood. It depends less on labels that are some distance away. It follows that if a label does not intersect other labels in the parent map, then it is probably advantageous to transfer it (during crossover) to a child together with the labels in its immediate vicinity. A good solution will be composed of such patches of labels that are placed well with respect to each other.

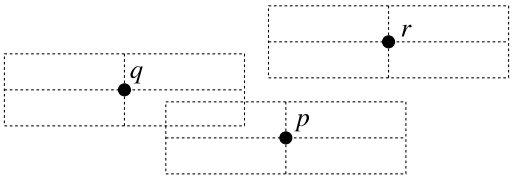

Figure 3.6: Cities p and q are rivals, but p and r are not.

To make this more formal, we define two points to be *rivals* if their labels *can* intersect. See Figure 3.6 for an example. A *rival group* is a certain point, called the *central point*, together with its rivals. A good placement of labels for a rival group can be considered a *building block* for the GA. Or, more accurately, the rival relationship defines the *linkage* of the encoding (which genes "belong together", see Section 2.2).* A GA finds good solutions by combining such building blocks.

*Actually, it is more accurate to say that the rival relationship is a good assessment of the true linkage of the encoding. We will discuss this further in Section 5.1 on page 106.

If one parent contains a building block for a certain region, and the other parent contains a building block for another region, we would like crossover to produce a child which contains them both. This exchange of building blocks is called *mixing* and is one of the fundamental mechanisms of the GA. Since rival groups overlap, it is inevitable that some rival groups will get split over both children after crossover. For example, suppose point q is a rival of point p, and p has no other rivals. A parent $\mathbf{x}$ contains corresponding genes x_p and x_q. Points p and q together form a rival group, and their genes can therefore store a building block. After crossover, the value in x_p can be copied to the first child, and the value in x_q can be copied to the second child. Assuming the first child holds another value for x_q (and similarly for the other child), the building block is *disrupted*.

In order to make the GA find a good solution, it has to mix building blocks and minimize the amount of disruption. We can now interpret the results of the simple GAs and understand why they performed so poorly. Both crossovers used by the simple GAs (one-point and uniform) have different mixing and disruption properties, which are intimately bound to the encoding used. We will discuss one-point crossover first.

The encoding used stores a labeling as a list of label positions. Two adjacent genes correspond with cities that can be anywhere on the map. One-point crossover works under the assumption that genes that are adjacent belong together (are linked). As a result, one-point crossover is biased to keep certain combinations of genes together. Unfortunately, it is very unlikely that the corresponding cities are actually near each other, and therefore the bias is inappropriate. We can demonstrate this more clearly by slightly altering the input of the algorithm. Instead of feeding it a random list of cities, we provide as input a list that is sorted on their x-coordinate. In the case of ties, the sorting is done on the y-coordinate. Now it makes more sense to transfer adjacent genes together, since they are more likely to be near each other on the map. However, crossover is still handling a two-dimensional problem in a one-dimensional way. The results improve (see Figure 3.7), but the end result is still poor.

Uniform crossover works under the assumption that the linkage between each pair of genes is equal. It mixes very aggressively, and it is hoped that the amount of disruption will still be acceptable. In the case of map labeling, a rival group will very likely be distributed over both children. The probability of a rival group being transfered intact decreases exponentially with the size of the group. On the maps used to compare the simple GAs, each point has on average roughly seven rivals. Therefore, uniform crossover is very disruptive.

It is important to realize how the GA gets supplied with building blocks and what happens to them. In Figure 3.8, we show four runs of different GAs, that differ in their crossover (one-point or uniform) and their setting for Pr_m (either 0.001, as before, or 0.000). Pr_c was kept at 0.7. The runs with one-point crossover show

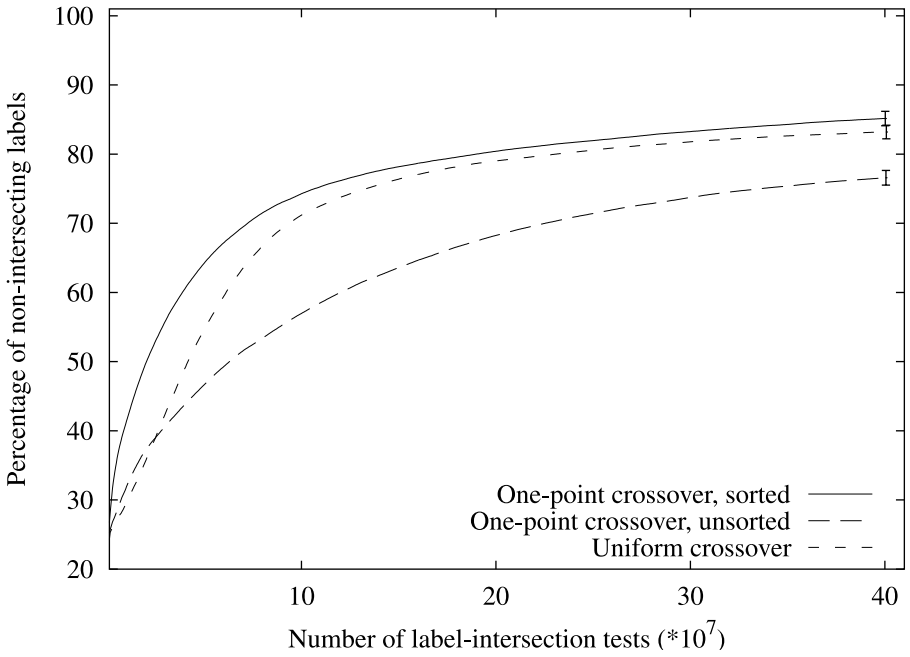

Figure 3.7: The same GA using one-point crossover on a random and a sorted list. Also shown is the GA with uniform crossover. Selection scheme is tournament selection, $Pr_c = 0.7$, $Pr_m = 0.001$.

that mutation largely supplies what crossover mixes; if mutation is turned off, premature convergence occurs. This happens because one-point crossover mixes quite slowly, so the population converges rather fast. This effect is less noticeable with uniform crossover, as can be observed in the figure. Uniform crossover mixes harder, and is able to combine the building blocks that were present after initialization.

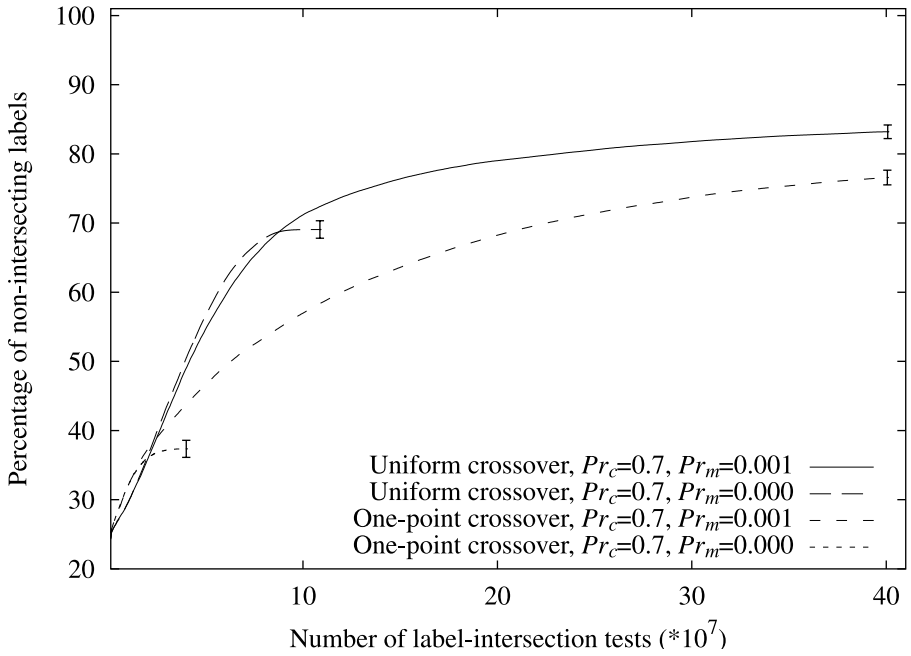

Figure 3.8: The interplay between mutation and crossover. Selection scheme is tournament selection.

3.2.2 Other GAs

It is also helpful to see how other GAs that try to solve the map-labeling problem cope with the problems sketched above.

Three other GAs that solve the map-labeling problem are known: the GAs by Djouadi,[18] by Verner et al.,[107] and by Raidl.[79] We will not discuss the GA by Djouadi in this thesis, since he uses rather unorthodox methods and does not give results which can be used to compare against.

In the light of the analysis of the previous subsection, it is interesting to see how the GAs of Verner et al. and Raidl cope with the problems of disruption and obtaining good mixing. Both algorithms report good performance, similar or slightly better than the simulated-annealing algorithm of Christensen et al. We

will say more about the latter study in Section 3.3, where we will discuss comparisons with the GAs.

The GA of Verner et al.

The GA of Verner et al. uses fitness-proportionate selection and the same generational replacement scheme that was described in Chapter 2 (shown in Algorithm 1 on page 19). The encoding is the same as described in Section 3.1. The fitness function (to be minimized) is a sum of several subfunctions:

$$f_{fit}(\mathbf{x}) = w_1 \cdot \mathit{overlap}(\mathbf{x}) + w_2 \cdot \mathit{area}(\mathbf{x}) - w_3 \cdot \mathit{distFact}(\mathbf{x}) + w_4 \cdot \mathit{pref}(\mathbf{x}) .$$

The number of overlapping labels is measured by $\mathit{overlap}(\cdot)$ and the total area of all overlapping labels is given by $\mathit{area}(\cdot)$. The function $\mathit{distFact}(\cdot)$ measures the *distance factor* of a point feature with an overlapping label, defined as the sum of distances from the center of its label and the centers of the labels from the four nearest features. The distance factor of a point feature with a non-intersecting label is zero. The last subfunction, $\mathit{pref}(\cdot)$, gives a measure of quality for the placement of the label with respect to the preferred positions. The weighting factors $w_1 \ldots w_4$ were empirically set.

Crossover occurs with probability $Pr_c = 0.9$. After crossover, mutation stores a randomly-chosen allele in a randomly-chosen gene with a probability of 0.1. The crossover operator battles disruption by using *masking* to preserve building blocks. Each chromosome has a bit string, or *mask*, associated with it. A bit is set to 1 for a specific gene if the corresponding point has a good labeling. A labeling for a point is considered *good* when the label does not intersect another label, and the point is not a *neighbor* of a point with an intersecting label. Otherwise, the labeling is considered bad. A neighbor of a point p is defined as one of the four points that are closest to p (using the Euclidean metric). Note the contrast with our use of a rival, where the number of rivals of a point is related to the density of the map.

Crossover is performed by looking at a random bit mask and the two masks of the parents (see Figure 3.9). For every location an allele is copied to a child from either the first or the second parent, depending on the mask. If the mask of the first parent signifies that the location is good, the contents are copied to the first child. If the mask of the second parent signifies a bad location, the contents of the first parent is also copied to the second child. If the masks of both parents signify a bad location, the random bit mask determines whether to copy from the first or the second parent. The procedure is symmetric for the other child. So if a location is considered good in one parent and bad in the other parent, both children get the information from the same parent.

0	0	1	0	1	Mask from parent 1 (p_1, 1 means "good")
0	0	1	1	0	Mask from parent 2 (p_2)
1	0	-	-	-	Random bitmask ('-' means that the value is irrelevant)
p_1	p_2	p_1	p_2	p_1	Child 1 inherits from...
p_2	p_1	p_2	p_2	p_1	Child 2 inherits from...

Figure 3.9: The crossover operator of the GA by Verner et al.

This scheme, although very exploitive, works quite well. Results are given in Section 3.3. If masking is turned off, the crossover is simply uniform crossover and performance drops. It is clear that knowledge about the structure of the problem is used to make the GA work well. It is used when it is decided whether an allele should survive crossover, which depends on the placement of the labels of close-by points. Disruption is reduced by preserving good labelings.

The GA of Raidl

The GA of Raidl uses tournament selection and an incremental replacement scheme. In this scheme, an iteration starts with selecting two parents. Crossover is applied with probability Pr_c. If no crossover is performed, two copies are made of the parents. The resulting individuals are immediately placed back in the population, replacing the worst individuals. The encoding is again the same as described in Section 3.1. The fitness function (to be minimized) is expressed as follows:

$$f_{fit}(\mathbf{x}) = overlap(\mathbf{x}) + \sum_{i=1}^{l} normPref(x_i),$$

where $normPref(\cdot)$ gives a penalty for the position of a label, and l is the length of the chromosome. The penalty is normalized, which means that the improvement of the fitness by placing an intersected label in a free position is never less than the penalty for that position. This way the GA also optimizes for position preferences, without degrading the number of free labels. Note that the use of weighting factors is carefully avoided.

Uniform crossover is always applied, and mutation is applied with probability 0.01 for each newly created individual. Mutation stores a randomly-chosen allele at a randomly-chosen gene.

The disruption of uniform crossover is repaired by using a *local optimizer*. The local optimizer examines all the labels of the map and, if possible, moves each to

a more desirable position. If the label intersects another label, any position where the label can be placed freely is more desirable. A position is also more desirable when it is more preferred in the order of positions.

3.2.3 A GA conforming to theory

We can improve on the simple GA from Section 3.1, by applying the insights from the preceding discussion. We have to ask ourselves the following questions:

- What do building blocks look like in the map-labeling problem?
- How can good mixing of building blocks be assured?
- How is disruption of building blocks minimized?
- How can we keep the fitness function simple?

We will start with the last point first, since it has consequences for the design of the rest of the GA. We will return to the other points when we discuss the crossover operator.

The GA uses the same encoding, and the same initializer, as before.

Fitness function

As described in Section 1.1, a good labeling is able to place many free labels and adheres to cartographic rules. The most important aspect is combinatorial, namely to maximize the number of non-intersecting labels. This aspect is global in nature and requires a global evaluation of the map. The cartographic rules are of a more local nature. In this chapter we concentrate mainly on the combinatorial aspect and try to develop a GA that can solve it well. However, our goal is to use the GA to solve map-labeling problems that also contain cartographic rules. We need to consider at this point how we can facilitate adding such rules later on.

As we saw in Section 1.1, a straightforward technique is to express the cartographic rules in the cost function. The best example of this approach is the simulated-annealing algorithm of Edmondson et al.[23] Such a function could have the following form:*

$$f_{cost}(\mathbf{x}) = w_1 \cdot f_1(\mathbf{x}) + w_2 \cdot f_2(\mathbf{x}) + \dots ,$$

where $f_i(\mathbf{x})$ provide global measures of different cartographic rules. We will first discuss the problems with using a cost function containing many subfunctions, and then propose an alternative.

*As discussed in Chapter 1, other forms are possibly better. However, the additive form is the one most commonly used.

Since cartographic rules are local in nature, they present problems when they are evaluated globally using the cost function. When a local rule is added to the combinatorial constraint in the cost function, it can deteriorate the combinatorial part, that is, it can cause fewer non-intersecting labels to be placed. The contribution from a function measuring a cartographic rule can make up for the decrease of the contribution from the function enforcing the combinatorial aspect. One can try to avoid this by using a large weight for the most important rules. Unfortunately, in a GA, large differences in fitness contributions may lead to hitchhiking. When a rule is locally hard, it becomes especially inappropriate to put it in the global evaluation measure. For example, making sure a capital is always labeled is hard to express in a global measure.

A genetic algorithm is particularly sensitive to the effects of a large summation of subfunctions, since it doesn't use neighborhood search. The simulated-annealing algorithm of Edmondson et al. is an example of an algorithm that uses neighborhood search. It tries to move a single label to a new position and then evaluates the difference in quality according to its cost function. Since the rest of the map stays the same, their contributions to the cost cancel out. The difference in quality can be calculated by only looking at the local changes. For example, if the cost function only counts the number of free labels, the change in cost can be deduced by looking at the surrounding labels of the changed label and checking whether their status has changed. It is not necessary to check all the labels on the map. In a genetic algorithm, a solution can be very different from its parents. Therefore, the whole fitness function has to be evaluated to calculate its fitness. The contributions of all the subfunctions are blended in a single fitness measure. There is the danger of degrading the combinatorial aspect to improve on aesthetic rules.

There are other problems with using a cost function with multiple subfunctions, that arise both with simulated annealing and genetic algorithms. To start, finding suitable values for the weighting factors is hard. The weights have no meaning in themselves, so they have to be set by repeatedly running the algorithm on the same map with different weights. Then the map has to be inspected (by hand) to see whether the algorithm did a good job. Evaluation of a whole map for the effect of a single weighting factor is difficult, since all factors are blended into one. Even when no inspection by hand is needed, this *tuning* of the weights takes a lot of time. In addition to the loss of computational resources, this makes it difficult to change the problem definition by adding a cartographic rule. It requires recalibrating all the weights or the whole algorithm has to be revised.

Additionally, tuning makes the algorithm only suitable for a particular type of map. The weights become specific for the maps on which the algorithm was ran when it was tuned. Running the algorithm on other maps (of a different type, a different scale, or a different density) may produce inferior results. The tuning

of the weights is typically done on small maps, to minimize the amount of time needed to complete a single run.

These problems can be exemplified by the study of Verner et al.[107] As described before, they propose the following fitness function:

$$f_{fit}(\mathbf{x}) = w_1 \cdot overlap(\mathbf{x}) + w_2 \cdot area(\mathbf{x}) - w_3 \cdot distFact(\mathbf{x}) + w_4 \cdot pref(\mathbf{x}) .$$

After introducing this fitness function they make the following remark:

> "The w_4 factor in the fitness function was not used in our analysis. The various combinations of values for w_1, w_2 and w_3 were arrived at by *numerous* trial and error testing to determine the best combination to use."
>
> (Verner et al.,[107] emphasis added and notation changed)

As Verner et al. demonstrate, tuning the weighting factors takes a lot of time. This time is spent in calibrating the GA itself; no problem is yet being solved. This would be acceptable if it only had to be done once. Verner et al. derive values of 1, 0, 0.0001 for w_1, w_2, and w_3 respectively from runs done on small maps (w_4 was always set to 0). Then they look at a more dense map:

> "Dataset R500 proved to be a very interesting problem. ... In this example a w_3 value of 0.00001 was used to keep the factor in line with the weight of the other parameters."
>
> (ibidem)

Unless one wants to do a lot of runs under nearly identical circumstances—an unlikely event in real GIS-use where users generate very different maps for different uses—a new tuning phase is needed for every fresh problem. This dramatically increases the time needed to solve a given problem.

We propose an alternative way of enforcing the cartographic rules. The fitness function should only express the combinatorial aspect. The cartographic rules are expressed in a new operator, the geometrically local optimizer. It has limited scope, and improves the labeling in its scope according to the cartographic rules. The GA gets provided with locally optimized pieces of the map (building blocks) and constructs a globally good map. Therefore, we will use the following fitness function:

$$f_{fit}(\mathbf{x}) = free(\mathbf{x}) ,$$

where $free(\mathbf{x})$ counts the number of non-intersecting labels in solution $\mathbf{x}$. By keeping the fitness function simple, we avoid the use of weighting factors that are needed to combine the combinatorial constraint with the cartographic rules. As a result, the design of the GA requires no tuning phase, which is a major advantage.

Additionally, this function has the important property that it is additively decomposable (see Section 2.1.4 on page 23): it can be expressed as a summation of partial functions which only depend on a few genes each. This property makes the analysis in Chapter 4 possible. We will keep using this fitness function in Chapters 5 and 6, were we extend the GA to handle cartographic rules and add line features to the problem. The geometrically local optimizer, which will handle the cartographic rules, will be discussed below, after discussing the crossover operator.

Rival crossover

The experiments with the simple GA from Section 3.1 show that one-point and uniform crossover by themselves both don't work well for the map-labeling problem. The GAs by Verner et al. and Raidl demonstrated that GAs using these crossovers can still perform well, provided that an additional mechanism is used—a masking technique in the GA of Verner et al., and a local optimizer in the case of the GA of Raidl. We choose a different approach, and design our own crossover that has the desired properties of good mixing and little disruption.

We would like to have something that mixes as hard as uniform crossover, but is much less disruptive. We observed in Section 3.1 that the position of a label is most influenced by the positions of the labels of its rivals. It is therefore reasonable to assume that two genes are linked when their corresponding cities are rivals. We will try to keep linked genes together during crossover. Our crossover is similar to uniform crossover, except that it works on the level of rival groups, instead of individual cities. We construct a *crossover mask* that determines which genes get transfered to which child. The crossover mask is a bit string, of the same length as a chromosome. If a bit in the crossover mask is 1 for a certain location, the allele stored at the gene with the corresponding location is copied from the first parent to the first child, and the second parent donates its allele to the second child. If the bit is 0, the first child inherits the gene from the second parent, and the second child inherits from the first parent. If we use a random bit string as a crossover mask, we obtain uniform crossover.

We start with a crossover mask filled with 0's. We repeatedly choose a point on the map and set its bit and the bits of its rivals in the crossover mask to 1. We keep doing this until the amount of 1's exceeds the amount of 0's. This way we transfer approximately half the map to each child, in order to obtain good mixing. We battle disruption by transferring whole rival groups. Unfortunately, rival groups

overlap, so a lot of rival groups will still be disrupted. Still, this is a good first step. We can see in Figure 3.10 that the new crossover, called *rival* crossover, improves on uniform crossover. Note that in this figure, we use tournament selection. Recall that a run is terminated when either the average fitness in the population is equal to the fitness of the best individual, or the limit of computational expense ($40 \cdot 10^7$ label-intersection tests) is reached. Rival crossover performs better, because it is less disruptive, while still mixing well.

Also shown in the figure is a run with rival crossover, where mutation has been turned off. Recall that we looked at the effect of mutation on GAs using uniform and one-point crossover in Subsection 3.2 (see Figure 3.8 on page 45). The effect of turning off mutation was less pronounced for the GA using uniform crossover, because it was able to mix the building blocks supplied in the initial population. Figure 3.10 shows that the effect of turning off mutation for the GA using rival crossover is similar as for the GA using uniform crossover.

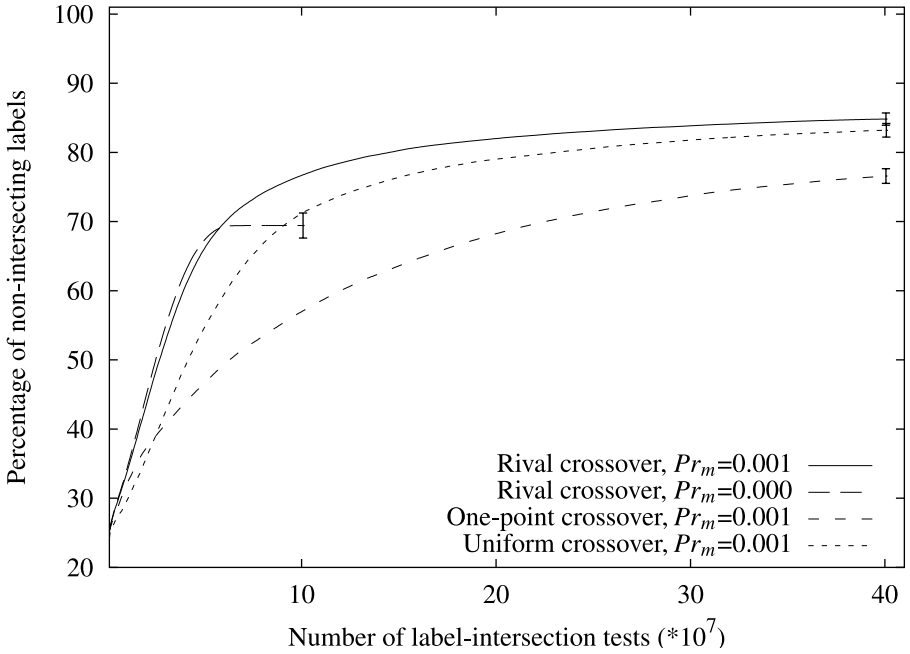

Figure 3.10: Comparison of rival with other crossovers. Selection scheme is tournament selection, $Pr_c = 0.7$.

Why do we use rival groups of this size? Alternatively, we could, for example, use rival groups composed of a certain point, its rivals, and the rivals of the rivals. Or we could go even one step further. In Figure 3.11, the results of four different GAs are shown, using different sizes for their rival groups. Note that mutation is turned off, because we want to isolate the performance of the crossover operator.

If the size is zero, the rival group is composed of a single point. It is similar to uniform crossover. It is not the same, since rival crossover always transfers about half of a parent to a child, while the proportion of the parent that uniform crossover transfers to a child is binomially distributed. If the size of the rival groups is one, we get the regular rival crossover. As the size increases, the patches which are copied from the parents get larger. The selection scheme used is again tournament selection.

The GA with size zero produces a result slightly inferior to the GA using size one, and takes more computation time. Sizes which are too large result in a drop in performance. This is to be expected, since with larger rival groups, genes are mixed less well. We can conclude that a rival size of one is best. Disruption is still quite high, though, which has a negative impact on the quality of the solutions found. To resolve this, we introduce a new operator: the geometrically local optimizer.

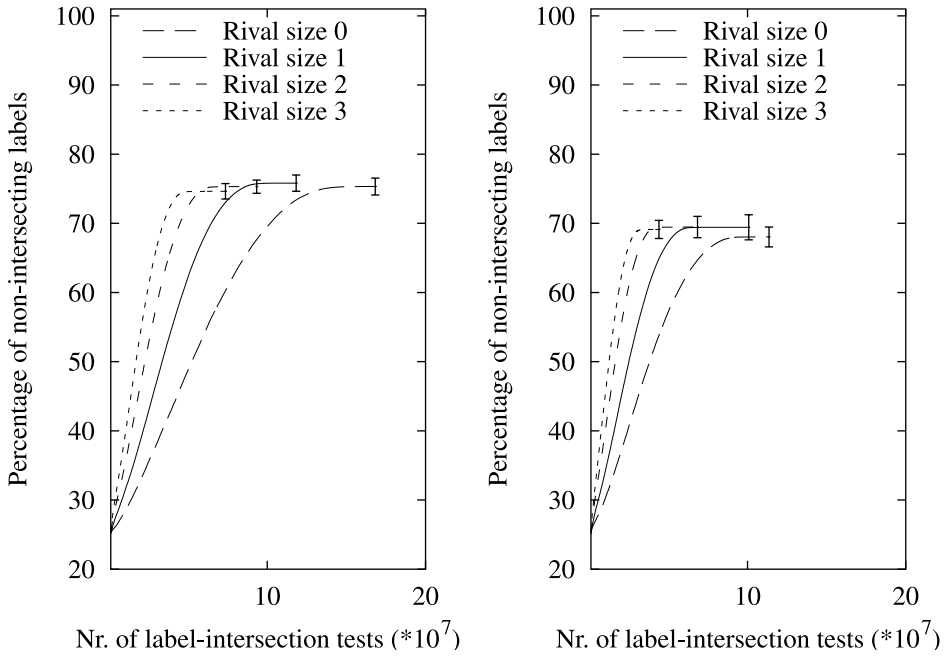

Figure 3.11: Comparison between different rival group sizes. Selection scheme is tournament selection. On the left, $Pr_c = 1.0$ and $Pr_m = 0.0$. On the right, $Pr_c = 0.7$ and $Pr_m = 0.0$.

Geometrically local optimizers

Rival crossover mixes well, but is still very disruptive. In order to get good solutions, we need to repair disrupted building blocks. This is done with the *geometrically local optimizer* (or GLO). It is applied to every point on the map that is the central point of a possibly disrupted rival group* (see Figure 3.12). Using the rival relation, we can give a good estimate of the *linkage* of the encoding (which specifies which genes "belong together"): two genes are linked if their corresponding points are rivals. When two genes that are linked inherit alleles from different parents, there is the possibility of disruption. Therefore, the GLO is applied to every gene that inherited its allele from parent p_1 and is linked to a gene that inherited from parent p_2 (or vice versa). The GLO is applied to these points in the order in which they occur in the input.

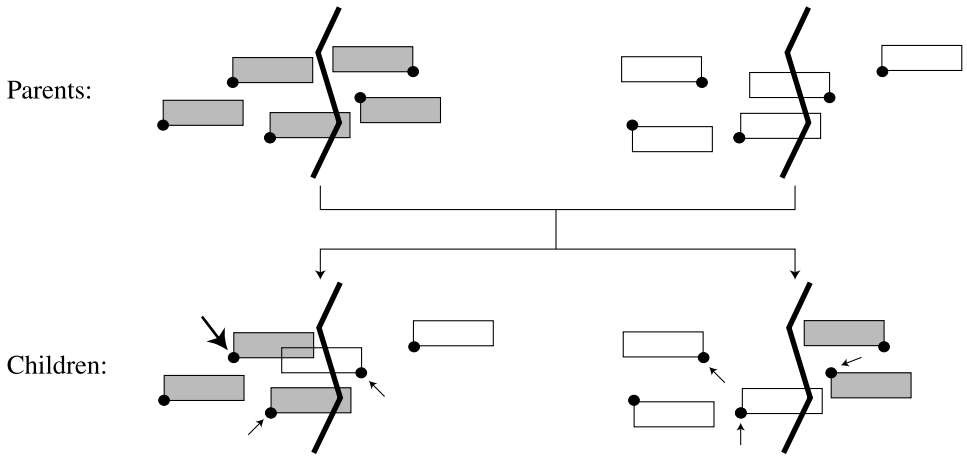

Figure 3.12: After crossover, the geometrically local optimizer is applied to points (indicated with arrows) that may be part of a disrupted rival group. In Figure 3.13, slot filling is applied to the point indicated with the largest arrow.

The GLO checks whether the label of the point it is applied to intersects another label. If this is the case, it then uses a procedure called *slot filling* to improve on the local labeling of a city and resolve conflicts. Slot filling views all possible candidate label positions as *slots*, which can be assigned attributes corresponding with their status on the map (see Figure 3.13, which shows the application of slot filling to the point in Figure 3.12 that is indicated with the largest arrow). A slot can be in two states: EMPTY and FULL. EMPTY signifies that no label of the rivals of the point is intersecting the candidate position corresponding with the slot. FULL signifies the opposite, and shows that the label of the point would intersect

*Recall that a rival group consists of a central point and its rivals (see page 42).

another label when placed there. After determining the status of all slots, a free slot is picked at random and the label of the point is placed in the corresponding label position. If no free slot is available, the label remains in its initial position.

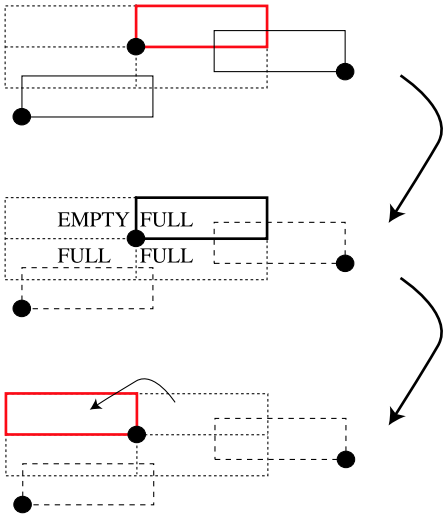

Figure 3.13: Slot filling. Top: original situation. Middle: assignment of status to slots. Bottom: label has been moved to a free position.

How well does this strategy work to reduce disruption? In Figure 3.14, we have compared three different GAs. They all use rival crossover and tournament selection. The first one uses slot filling as geometrically local optimizer. The second GA uses a different procedure: the label (of the point corresponding to the gene the GLO is applied to) is moved to a random position, and the change is kept only if it improved the fitness of the solution. The last GA uses a geometrically local optimizer that does nothing after the check for a conflict, and serves as a point of comparison. All GAs have mutation turned off to show the effect of the GLO on the performance of rival crossover. As we can see, the use of a geometrically local optimizer, and slot filling in particular, dramatically improves the performance of the GA. Indeed, the quality of the solution increases from less than 80% when no GLO is applied, to more than 99% when slot filling is used.

The use of a GLO does have a price, because label-intersection tests have to be performed when checking for a conflict, during slot filling, and after a random repositioning to check whether the labeling has improved. This can be observed by comparing the GA with rival size 1 from Figure 3.11, and the GA with the GLO which does nothing after the conflict check from Figure 3.14. The difference in the amount of computation is due to the fact that the GA using the GLO has to spent label-intersection tests to check whether the label considered by the GLO is intersected by another label.

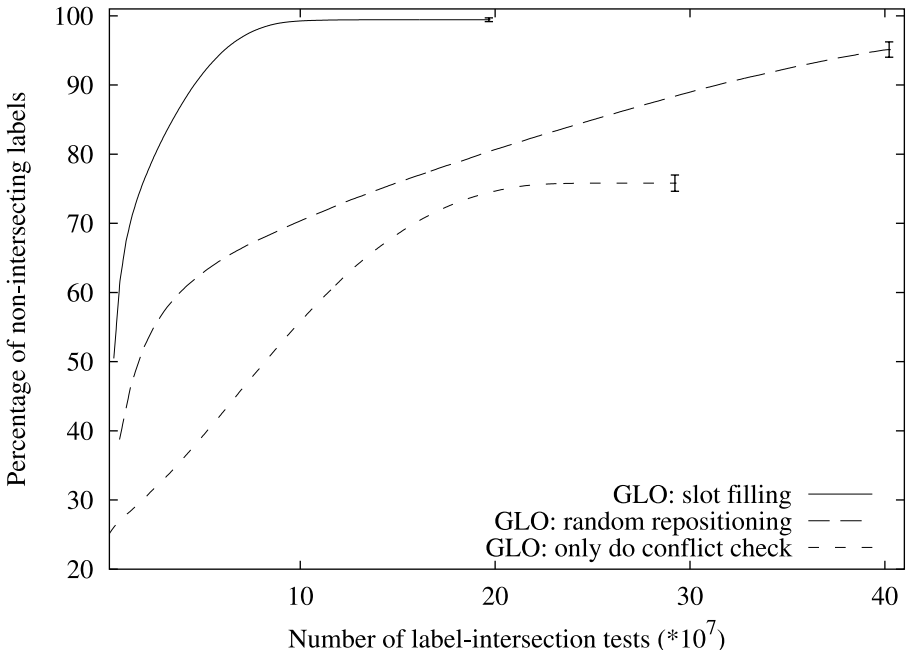

Figure 3.14: GA using different GLOs. Crossover is rival, selection scheme is tournament selection, $Pr_c = 1.0$, and $Pr_m = 0.0$.

The geometrically local optimizer is characterized by the facts that it has very limited scope, and that it tries to improve the part of the solution in its scope. This scope is geometrically determined: in this case it consists of a point and its rivals. Moving labels that intersect another label to a free position is an obvious example where the geometrically local optimizer improves the solution. We will see in Chapter 5 how cartographic rules can be incorporated using the geometrically local optimizer. This will demonstrate that improvements can be more subtle, because the GLO doesn't necessarily require the quality of the solution to be quantified. For example, locally it can be decided, if there is room, to place a label in the most preferred position, without having to quantify this. This is an important difference with the method of expressing cartographic rules as additional functions in the fitness function.

The effect of the geometrically local optimizer on the way the GA works can also be discussed in terms of building blocks. The genetic algorithm needs a supply of building blocks that will eventually be combined in a single individual. Sources of building blocks are typically the initializer and the construction of a building block by chance, due to crossover or mutation. The geometrically local optimizer is another source of building blocks, which is made possible by our knowledge about the (geometric) structure of the problem.

Note the difference between the GLO and a "normal" local optimizer. A local optimizer is a procedure that finds a better point by looking in the neighborhood of the search point on the fitness landscape. Our use of "local" refers to the locality in the geometry of the problem. Of course, the combined applications of the GLO on certain points after crossover is a local optimizer in the usual sense. Note the resemblance with the approach used in the GA of Raidl (see Section 3.2.2, page 47).

Normally, a GA experiences a bias towards solutions with high fitness values, as a result of selection. In the GLO, certain configurations of labels are changed in other configurations, introducing a new bias in the search process. As a result, we have to be careful when we design the GLO that it acts in concert with selection. For example, if the geometrically local optimizer would move labels from free positions to intersecting ones, it would have a bias opposite to the one from selection. Fortunately, for the map-labeling problem we can decide with reasonable confidence which configurations are good and which are bad, just by looking at the local situation.

Another important point in the design of the GLO is that the GA should be able to construct a good solution using the locally good configurations generated by the GLO. The GA will need some choice in the configurations it can combine. Hence, the GLO should be careful to produce configurations with enough variation, so that enough diversity is kept in the population. Since the geometrically local optimizer is applied to rival groups with their labels in different configurations, this won't normally be a problem.

Summarizing, the uses of the geometrically local optimizer are:

- It makes it possible to exploit the knowledge about the structure of the problem.
- It can repair rival groups which were disrupted during crossover.
- It generates building blocks.
- It facilitates adding cartographic rules to the problem, while keeping the fitness function simple. This has the important benefit that the fitness function remains additively decomposable.

Selection scheme

We use the *elitist recombination scheme*[92,96] to perform selection. In this scheme, two parents are randomly chosen from the population. Crossover is always performed, and two children are generated. From this family of four, the two best individuals replace the parents in the population. In the case of ties, children

precede their parents. The advantages of the elitist recombination scheme were discussed in Subsection 2.1.5 (page 25). The selection pressure can be tuned by holding a tournament for one of the parents. We use the scheme described above, which gives reliable results. The selection pressure that the standard elitist recombination scheme uses is equal to that of tournament selection using a tournament size of two.[9,92,95]

We empirically demonstrate the usefulness of the scheme in Figure 3.15. A GA using elitist recombination is compared with similar GAs using tournament selection, and different settings for Pr_c and Pr_m. The GAs all use rival crossover and the geometrically local optimizer. The GA using elitist recombination finds a slightly better solution in less time than the other schemes, due to its elitist strategy.

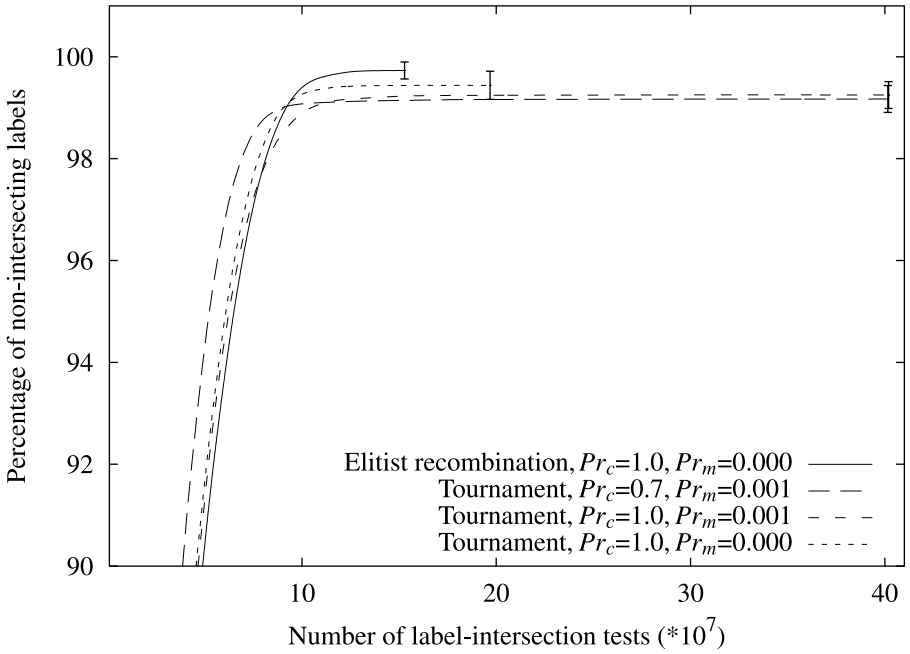

Figure 3.15: Comparison of the elitist recombination scheme with tournament selection. The GAs use rival crossover and the geometrically local optimizer. Note that the scale for the percentage of non-intersecting labels is between 90 and 100.

Overview of the GA

We propose a GA that uses the elitist recombination scheme, rival crossover, and the geometrically local optimizer. We use $Pr_c = 1.0$ and $Pr_m = 0.0$. We do not

use mutation in its traditional form. Instead, the geometrically local optimizer explicitly tries to create improvements to geometrically local parts of the solution. Our fitness function is deliberately kept simple to avoid tuning, and only counts the number of non-intersecting labels.

How well does this GA perform, compared to the simple GA we described in the previous section? As can be seen in Figure 3.16, the new GA is able to find significantly better solutions. The next step is to compare our GA against other methods from literature.

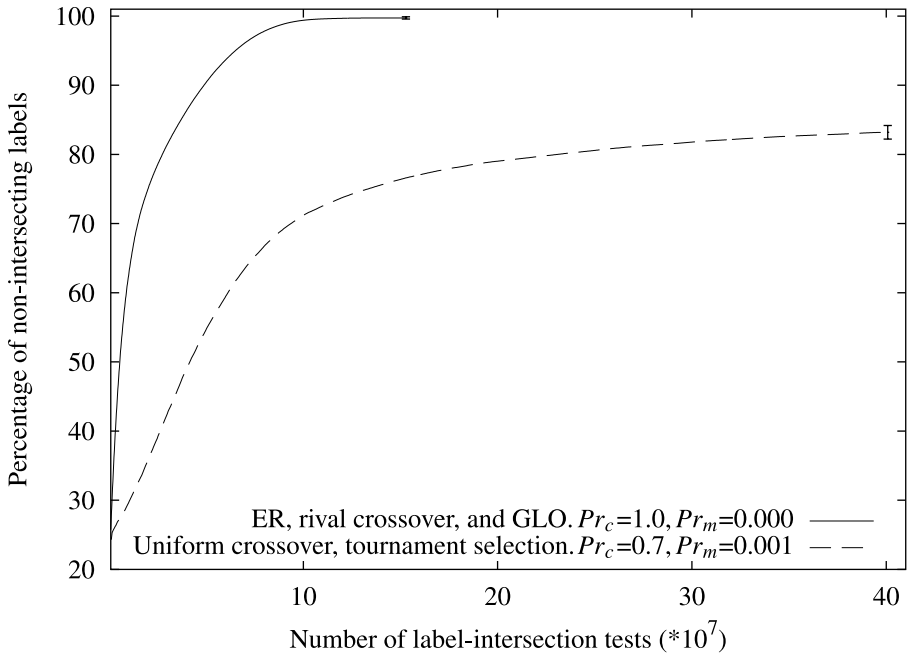

Figure 3.16: The new GA compared with the simple GA.

3.3 Comparison with other techniques

In the previous section, we presented a genetic algorithm to find good solutions for the map-labeling problem. It is based on known GA-theory, leading to a GA that performs well. In this section we compare the resulting GA with methods from literature that reported good results.

A comprehensive study that compared various different map-labeling algorithms on a class of randomly generated maps was performed by Christensen et al.[14] First, they argued that algorithms using a rule-based approach that performed

back tracking[20,50] could never be computationally efficient, due to the combinatorial explosion that is inherent of NP-complete problems. They then considered the following algorithms, which represented the state-of-the-art at the time (1993):

- Greedy algorithms: to test a "greedy" approach, a rule-based algorithm was used, but no back-tracking was performed when a label intersection could not be resolved.
- Hirsch's algorithm: Hirsch[43] provided a heuristic based on a system of mutually repulsing labels, in which a minimal-energy state is sought.
- 0/1 integer programming: Zoraster[117,118] formulated the problem as a 0/1 integer-programming problem, which subsequently was solved using Lagrangian relaxations.
- Random placement: maps with a random placement, on which no optimization was done, were used as a baseline to compare against.

They proposed two algorithms of their own devise:

- Gradient descent: this hill-climbing algorithm considered all possible alterations that could be made to a solution, choosing the best one. The algorithm terminated when no improvements could be made. Several variants of this method were tested, which differ in the amount of look-ahead (subsequent alterations) they had. More look-ahead enabled the algorithm to avoid local optima, at a computational cost.
- Simulated annealing: since they recognized the combinatorial nature of the problem, they employed simulated annealing, a heuristic combinatorial problem solver, to find solutions for the map-labeling problem. It is able to avoid local optima by not always choosing the best option available. In fact, the solution is allowed to become worse at times. A detailed discussion of simulated annealing is given later.*

For the comparisons Christensen et al. used randomly generated maps. Each map consisted of a grid of fixed dimensions (792 units by 612 units) on which n_{feat} points were placed. Each point possessed a label of fixed dimensions (30 units by 7 units). They used maps that were progressively more dense, by ranging n_{feat} from 100 to 1500. An example of such a map with 500 points is shown in Figure 3.17. Note that beyond a certain point it becomes impossible to place all labels on the map without intersections. In this aspect they differ from the maps we

*Edmondson et al.[23] later extended this algorithm to include more cartographic rules.

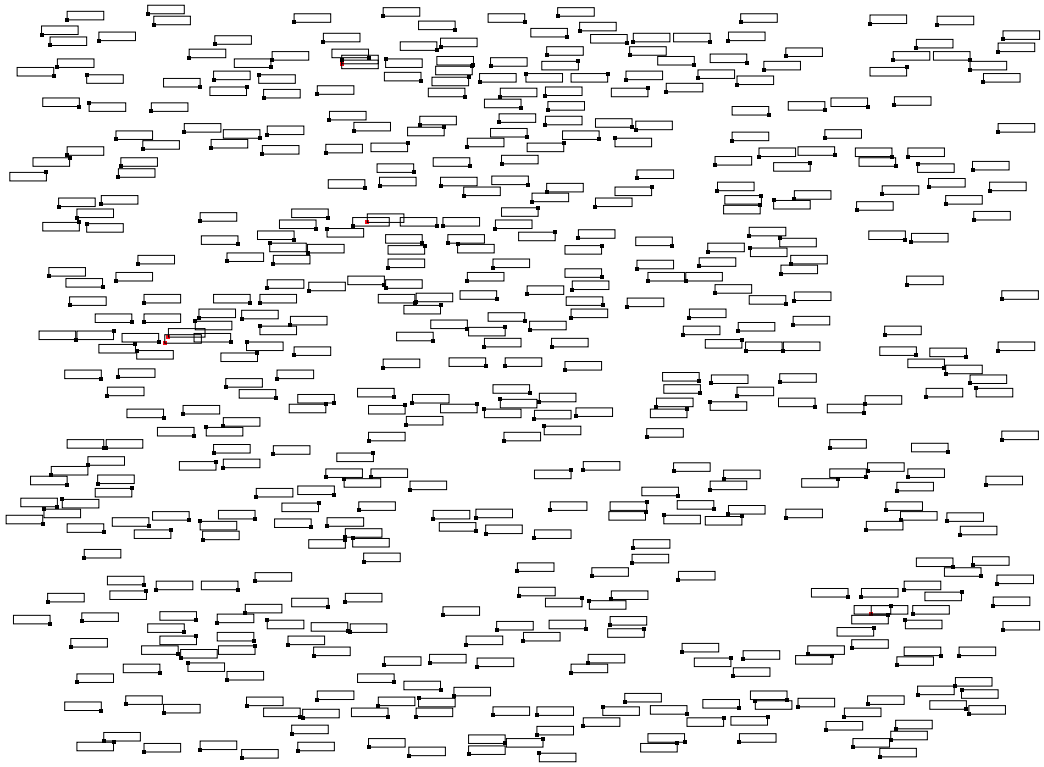

Figure 3.17: An example of a map with 500 points, as used by Christensen et al. Note that some irresolvable intersections remain.

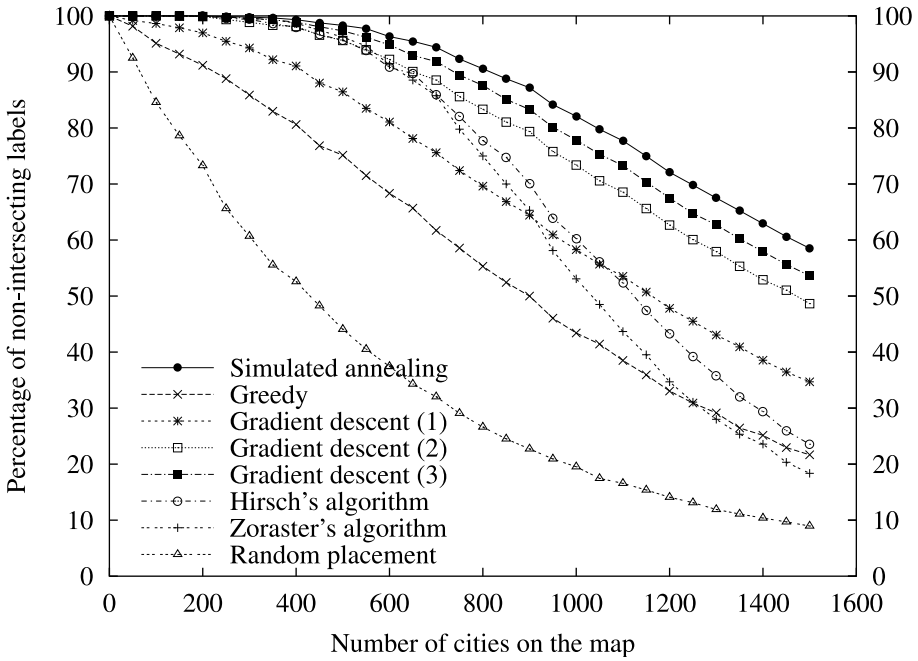

Figure 3.18: Results from the paper by Christensen et al.[14]

used until now. The problem they studied was the basic map-labeling problem, in which the number of non-intersecting labels has to be maximized, and each label can be placed in one of four positions. The results are given in Figure 3.18. They concluded that simulated annealing gave superior results in terms of quality.

Verner et al.[107] proposed a genetic algorithm for the basic map-labeling problem, and compared it with the simulated-annealing algorithm of Christensen et al. They report favorable results. We will compare our GA with the algorithm of Christensen et al. and their algorithm.

Recently, several new algorithms have been proposed that focus on the basic map-labeling problem: the GA by Raidl,[79] the tabu-search heuristic by Yamamoto et al.,[114] and heuristics for maximum independent set by Strijk et al.[89,87,108] We did not have the opportunity to perform experiments that compare our GA with these algorithms. All algorithms report similar, or slightly better results than simulated annealing. The genetic algorithm we propose also produces results with the same quality as simulated annealing. It is therefore also important to look beyond the combinatorial aspect of the map-labeling problem, and discuss how one can incorporate cartographic rules in a map-labeling algorithm. We will postpone this discussion until Section 3.4.

Another, equally important property of any algorithm is its scale-up behavior:

how does the required amount of computation relate to the input size? We discuss the scale-up behavior of our GA in Chapter 4.

map size	100	150	200	250	300	350
our GA	150	150	150	150	150	150
GA of Verner et al.	200	250	250	250	400	400

map size	400	450	500	750	1000	1500
our GA	150	150	150	300	500	1000
GA of Verner et al.	400	400	400	400	500	1500

Table 3.1: Population sizes used by the GAs for the different maps.

The population sizes used for the genetic algorithms are given in Table 3.1. The population sizes for the GA of Verner et al. are the same as they used, except for the map with 1500 points. They did not experiment with such maps, so we chose a population size which is certainly large enough. The population sizes for our GA were intuitively chosen (the "reckoning" method discussed in Section 2.1.1 on page 20). It is unlikely that they are optimal. We will return to the question of finding a good population size in Chapter 4.

An interesting property of the randomly generated maps Christensen et al. used is that they can be decomposed into separate problems, which can be solved independently. As described before, the placement of a label of a point depends on the placement of the labels of its rivals. Since they depend on the labels of *their* rivals, the rival relationship connects points which may need to be considered collectively to find the optimal solution. As such, it induces a graph:

Definition 4 ***Rival graph***

The rival graph *of a map of points P is the graph $G_r = \langle P, E \rangle$ which has a node for each point $p_i \in P$. There exists an edge $\overline{p_i p_j} \in E$ when the points p_i and p_j are rivals.*

The separate problems mentioned above correspond with *connected components* in the graph. A connected component is a set of nodes in the graph for which there exists a path between each pair of nodes in the set. The connected components of the map from Figure 3.17 are shown in Figure 3.19.

The map-labeling problem can be solved by breaking down the rival graph into its connected components and solving these independently, afterwards merging the results. We did not use this property, and always solve the entire map as one problem. Our reasons for this are as follows:

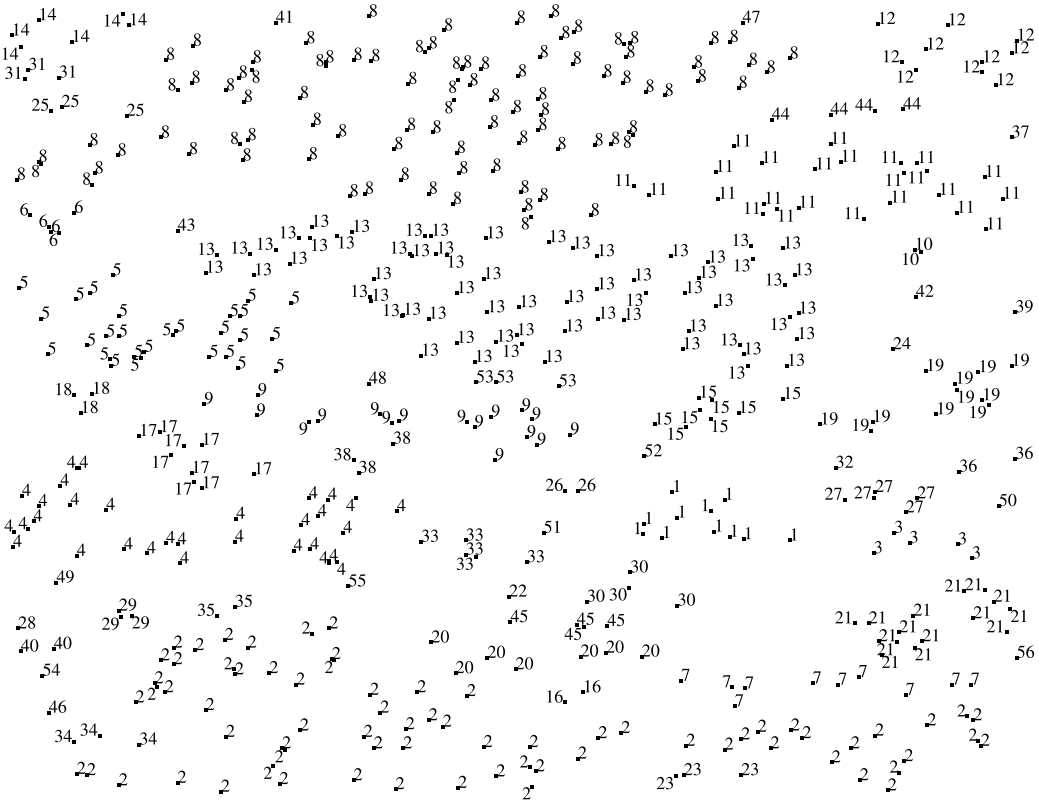

Figure 3.19: Connected components in a map (two points are in the same component if they are labeled with the same number). Compare with Figure 3.17.

- The algorithms we compare our algorithm with (simulated annealing and the GA of Verner et al.) also did not solve connected components independently. Comparisons with these algorithms are clearer if we use the same conditions those algorithms used.

- Our genetic algorithm does not find better solutions if the problem is split into small components. The GA benefits from solving each component separately in terms of speed only.

- In Chapter 6, we extend the map-labeling problem to include line features as well. Line features can span a large portion of the map, and will have many rivals. Therefore, the map will very likely contain very few connected components. Including an optimization that will be useless in the future would be bad design.

- As the map density increases, the number of connected components decreases. As a result, the benefits of solving each connected component independently are quickly lost when the maps get more dense. This would make comparing the results on sparse maps with those on dense ones more difficult.

The kind of maps that we used before, in which it was guaranteed that all labels can be placed, have practically no separate components. For example, the map from Figure 3.4 contains three separate components, of sizes 998, 1, and 1; only the two points in the top right have no rivals.

In the remainder of this section we present the results of the comparisons. We will start by comparing our GA with the simulated-annealing algorithm of Christensen et al., followed by a comparison with the GA of Verner et al. We end the section by a comparison in the running times of all algorithms considered.

Simulated Annealing

The process of heating a metal until it melts and then slowly cooling it until it solidifies is called *annealing*. The atoms in the metal start to move randomly because of the heating, and if the cooling is done sufficiently slow, the atoms crystallize in a highly ordered structure. If the cooling is done too fast, the crystallization is not done homogeneously, and the metal is less structured (which means it is less strong).

An analogy of annealing can be used to devise an algorithm, which is thus called *simulated annealing*.[60] Like with a genetic algorithm, we need a cost function and a way to encode a solution. Unlike a genetic algorithm, which works

with a population of solutions, simulated annealing only uses one solution. We start with a randomly generated solution. We repeatedly alter a randomly chosen part of the solution, and see if the solution has become any better. If it has, the change is kept and another iteration is started. If it is worse, a choice should be made. Either the change is discarded, on the grounds of it degrading the solution, or it is kept, on the grounds of it giving a possible way to escape a local optimum. This choice is made according to a certain probability. This probability is dependent on a variable which models the temperature of the process. At the start of the run, the temperature is high: the system is in a random state. The probability of keeping a change for the worse is high. During the run of the algorithm, the temperature is lowered sufficiently slowly and the probability of keeping a bad change converges to zero. In the end, only good changes will be kept. It is now the hope that the solution has avoided local optima and has found the global optimum. The speed at which the temperature is lowered is called the *annealing schedule*.

Simulated annealing for the basic map-labeling problem can be done by simply repositioning a label at each iteration and observing the change in the cost function—the number of intersecting labels. If the solution has become worse, the change is kept with probability $Pr = e^{-\frac{\Delta E}{T}}$ where ΔE is the additional number of intersecting labels and T is the temperature. For high temperatures, we have: $\lim_{T \to \infty} Pr = e^0 = 1.0$. For low temperatures, we reach: $\lim_{T \downarrow 0} Pr = e^{-\infty} = 0.0$.

The simulated-annealing algorithm we implemented is the one described by Christensen et al.[14] It works as follows. The initial value of T is chosen such that $Pr = \frac{2}{3}$ when there is one additional intersecting label ($\Delta E = 1$). After $20n_{feat}$ alterations, the temperature is decreased by 10%. It is also decreased immediately if the last $5n_{feat}$ alterations were accepted, and occurred at the same temperature. The algorithm continues until the temperature has been decreased 50 times. Additionally, it stops immediately if the last $20n_{feat}$ alterations at the same temperature were all rejected. The final solution is given as output of the algorithm.

We first compare simulated annealing (or SA) against our GA for the basic map-labeling problem using the four-position placement model. Figure 3.20 shows the quality of the labelings both algorithms found for maps of different sizes. Each data point in the graphs is the average of five runs, each conducted on a different map of the same size. Since the GA derives much of its power from the use of the geometrically local optimizer, we also included a variant of simulated annealing that employed it. Instead of randomly choosing a new location for the label of a given point, slot filling is applied to the point. As a result, no changes for the worse can occur, which is necessary for simulated annealing to optimize well. The figure shows that the GA and normal SA find maps of similar quality, but SA using the geometrically local optimizer has poor performance.

We also compared both algorithms for two other variants of the map-labeling

problem. In Figure 3.21, the results are shown when both algorithms use the eight-position placement model. In Figure 3.22, comparisons are shown of the performance of the algorithms on the problem which included integrated name selection. This was done by extending the placement model with an extra position, indicating the label has been removed. Previously, two intersecting labels did not contribute anything to the overall cost (or fitness). Now, one of the two can be deleted, freeing the other. Initialization of the initial solutions for both algorithms was done as before, by choosing a random position for every label on the map. The "deleted" position was not used. For simulated annealing, no other adjustments were necessary. For the GA, we alter the geometrically local optimizer. It is applied when the label intersects another label, or when the label is deleted. In the slot filling procedure, when the label can not be moved to a free slot, it is deleted.

In both the case of the eight-position model, and the case of integrated name selection, the genetic algorithm slightly outperforms simulated annealing. This gives an indication that the genetic algorithm is more robust in the sense that it copes better with changes in the problem definition. However, it should be noted that there is a substantial difference in the running times of the algorithms (see also Figure 3.24 on page 72). For the densest maps, simulated annealing was able to find the solution quicker than the genetic algorithm. An annealing schedule is a trade-off between the amount of computation and the quality of the solutions. Therefore, one could change the cooling schedule of the simulated-annealing algorithm, and lower the temperature more slower. The algorithm may find better solutions with a more time-consuming cooling schedule. However, Christensen et al. say that they "found the particular choice of annealing schedule to have a relatively minor affect on the performance of the algorithm" (Christensen et al.[14]). Therefore, we did not experiment with cooling schedules.

The GA of Verner et al.

The GA of Verner et al. was described in Section 3.2 (see page 46). We implemented two variants of their algorithm. The first implementation faithfully follows the directions in their description of the algorithm. Unfortunately, we were unable to reproduce the reported results. We found that we could improve the performance of our implementation if we changed the algorithm slightly. Recall that a labeling for a point is considered "good" if the label of the point is not intersected, and additionally the labels of the four closest neighbors are not intersected. The second implementation considered a labeling for a point to be good if the label of the point is not intersected. Labels of neighbors can intersect.

In Figure 3.23, we show the results of our GA, the results reported by Verner et al., and our two implementations of their GA. Again, each data point in the graphs

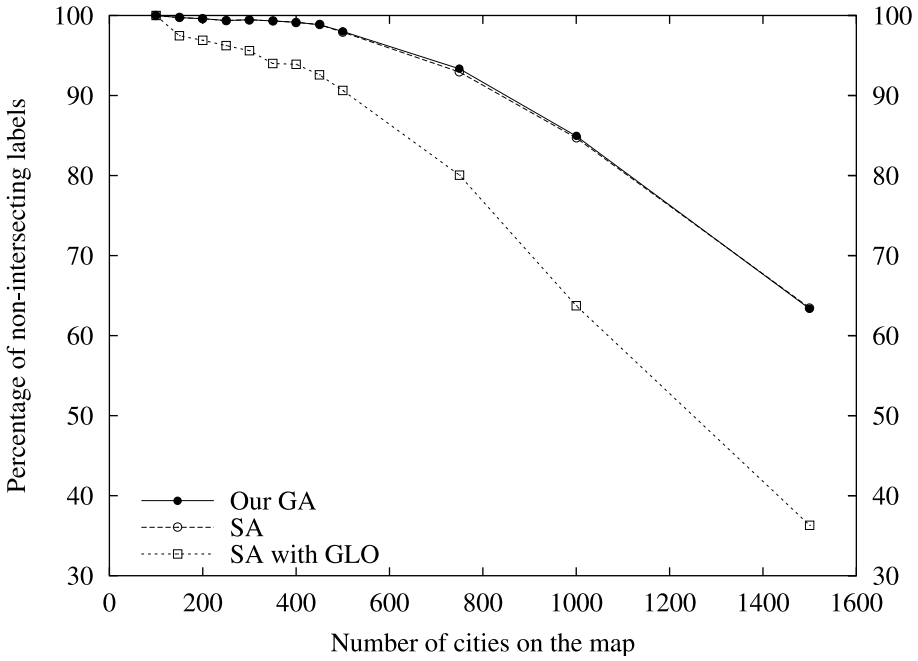

Figure 3.20: Comparison of SA with our GA for the four-position model. Also shown is a run of SA with the GLO.

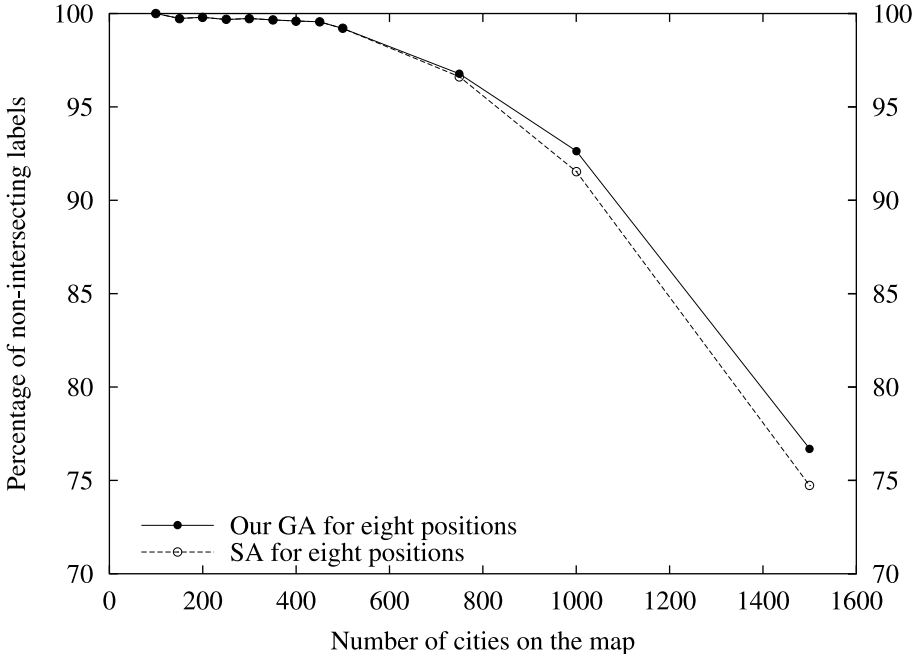

Figure 3.21: Comparison of SA with our GA for the eight-position model.

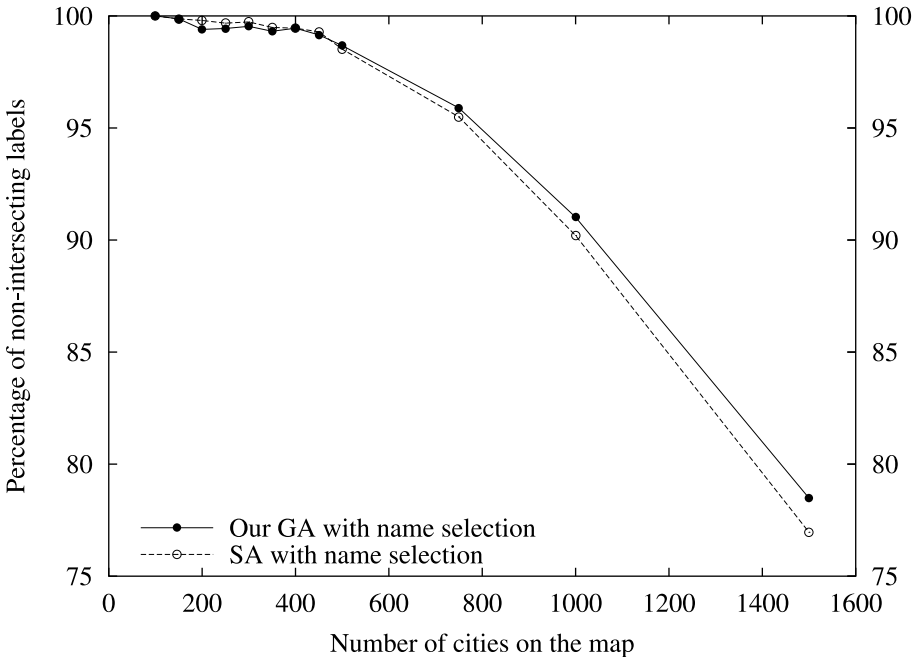

Figure 3.22: Comparison of SA with our GA for name selection.

is the average of five runs, each conducted on a different map of the same size. Note that all algorithms used the eight-position placement model, as suggested by Verner et al. We observe that our second implementation matches the reported results most closely.

The results of our GA are better than the results reported by Verner et al. However, we have to be cautious about drawing conclusions because the algorithms were not run on exactly the same maps. The maps were randomly generated, so while the maps used by Verner et al. are very similar to our maps, they are not exactly alike. Note that the results given for the simulated-annealing algorithm (in Figure 3.20 on page 68) also deviate slightly from the results reported by Christensen et al. (shown in Figure 3.18 on page 62); the results in Figure 3.20 are slightly better. However, even if it were necessary to apply the same correction to the results of Verner et al., our GA would still be competitive.

Running times

When comparing different algorithms, one is primarily interested in the quality of the solution an algorithm finds. However, this can only be meaningful when seen in relation to the amount of time it took to find those solutions. We can provide a rough indication of the way computational effort is related to the quality found.

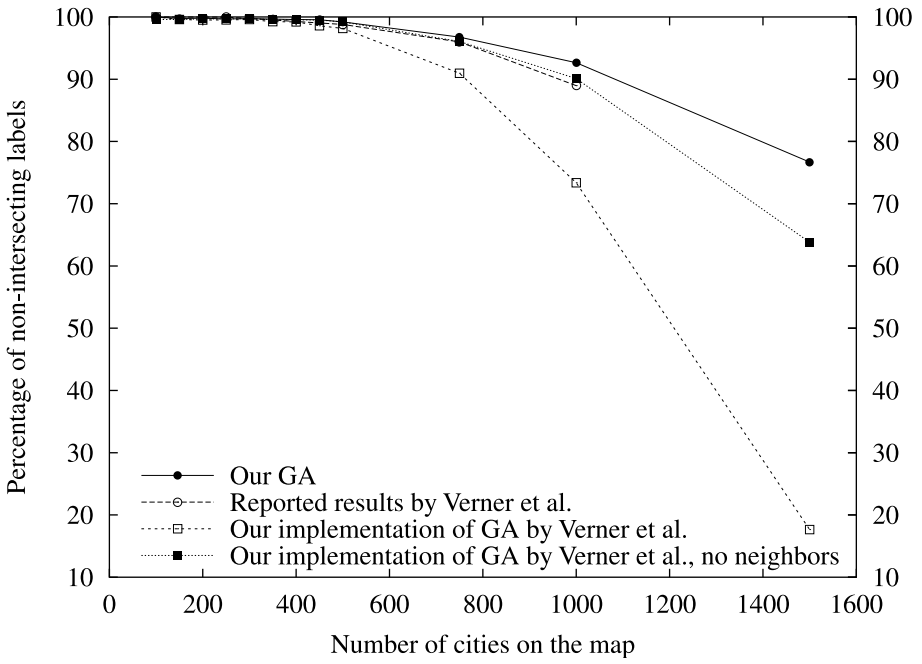

Figure 3.23: Comparison of our GA with our implementation of the GA of Verner et al., their reported results, and our implementation of their GA without neighbors.

Figure 3.24 shows the amount of label-intersection tests performed to find the solution for a given map size, for all algorithms discussed. Note the logarithmic scale for the amount of label-intersection tests. Measuring label-intersection tests allows us to compare the genetic algorithms (a population-based approach, requiring global evaluations of a map) with simulated annealing (an algorithm using a single search point, and capable of doing local evaluations). The figure doesn't show results for maps with 1500 points, since we were not able to reliably count the number of label-intersection tests, due to overflow of the counter in some experiments. In addition, recall that we used very large population sizes for the GAs, which is of course a disadvantage for the GA. Better running times for the GA can be obtained by using smaller populations—see Chapter 4.

As the maps get more dense, the problem becomes more complex. As a result, we don't expect a map with double the number of features to require twice the amount of computation, due to the added complexity. For maps with *constant* density, we show in Chapter 4 that the GA has quadratic scale-up behavior: a map twice as large takes four times as much computation time.

Comparing the running times of simulated annealing with those of our genetic algorithm, we see that SA takes less time on the densest maps. We may conclude, therefore, that it is the algorithm of choice when one wants to solve the basic map-labeling problem on very dense maps. In reality, the map-labeling problem is more complex and multi-faceted. A deciding factor in the choice of algorithm for a map-labeling problem is how well the algorithm is extendible with new cartographic rules. We will discuss this in the next section.

3.4 Discussion

As described in the Chapter 1, there are two reasons why the map-labeling problem is hard:

- The basic, combinatorial problem has a large, complex search space.
- The problem can include many cartographic rules of a local nature. A good labeling strikes a balance between these rules.

In this chapter, we have concentrated on the combinatorial part of the problem, and have developed a genetic algorithm for the basic map-labeling problem. We have shown that it finds good solutions and is competitive with other map-labeling algorithms. We didn't consider any cartographic rules, but we designed the GA specifically with their incorporation at a later stage in mind. In Chapter 5, we will show how the GA can be extended with cartographic rules.

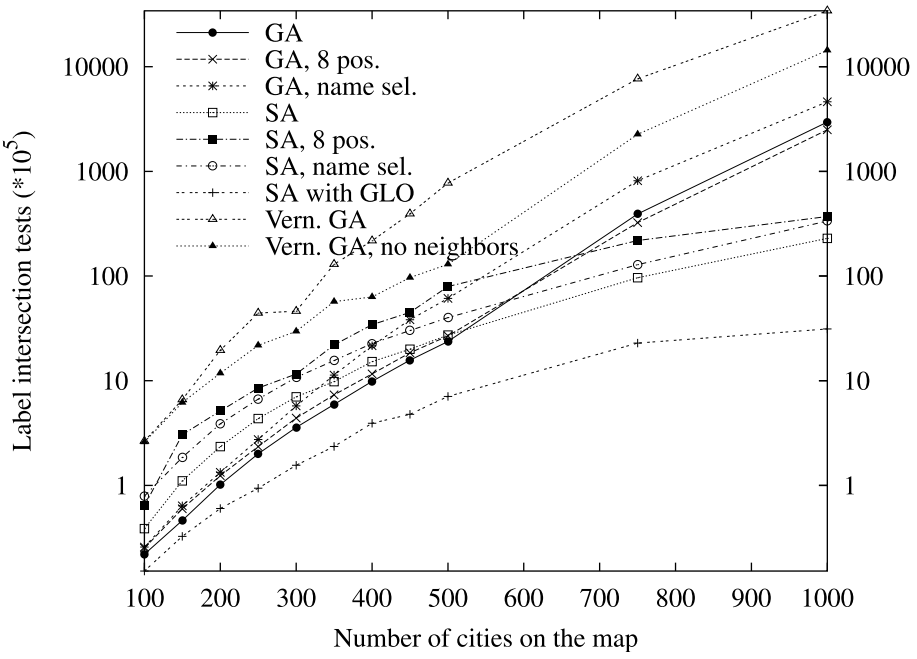

Figure 3.24: Comparison of running times. Note the logarithmic scale for the amount of label-intersection tests. The following abbreviations are used in the key. "GA" refers to our GA for map labeling. "GA, 8 pos." is the same GA using the eight-position placement model. "GA, name sel." is the same GA with integrated name selection. Similarly, "SA" refers to the simulated-annealing algorithm, "SA, 8 pos." to the variant using the eight-position placement model, and "SA, name sel." to the variant with name selection. "Vern. GA" refers to our implementation of the GA by Verner et al. "Vern. GA, no neighbors" refers to our implementation of their GA which does not use neighbors (see the discussion on page 67).

In order to be a viable solution for the map-labeling problem, an algorithm should be *robust*. That is, it should be possible to integrate the algorithm in a geographical information system (GIS), where it can be used transparently by the user of the GIS, and easily extended by the maintainer of the algorithm. This means, for example, that we have to be careful in introducing parameters that are difficult to set, like Pr_c, Pr_m, and the population size. We have handled these parameters in a number of ways: Pr_c can safely be set to 1.0 thanks to the elitist recombination scheme, and Pr_m was set to 0.0 because its role has been taken over by the geometrically local optimizer. The population size remains as a parameter which has to be set, but there is an intuitive relation between this parameter and the running time of the GA. Using a small population size gives poor solutions, in little time. Using a large population gives better solutions, which takes more time. However, if desired, one can adopt one of the adaptive population-sizing schemes mentioned in Section 2.1.1 (see page 21). In the next chapter we will explore the topic of choosing a good population size further.

It is also important to avoid a tuning phase, for all the reasons mentioned in Section 3.2.3 (on page 49): the cartographic rules can degrade the combinatorial rule, evaluating a map is hard, tuning takes time, and it makes the algorithm inflexible. We managed to keep our fitness function simple by using the geometrically local optimizer, in which we will incorporate the cartographic rules.

Summarizing, in order to be robust, the following properties should hold:

- It should be possible to extend the definition with additional constraints (cartographic rules) without dramatic changes to the algorithm or much loss of quality.
- The number of parameters that are difficult to set, like Pr_c, Pr_m, and weighting factors should be kept at a minimum.
- No tuning should be necessary.

How robust are other algorithms for map labeling? It is hard to give a general answer to that question, but we would like to draw attention to the necessity of making the algorithm extendible with additional cartographic rules. We briefly discussed in Chapter 1 the simulated-annealing algorithm of Edmondson et al., which added rules by using multiple functions in the cost function. We summarized the problems with that approach above. We have proposed a new technique for incorporating additional rules—the geometrically local optimizer—that provides an elegant way to combine combinatorial optimization with enforcing cartographic rules. It seems difficult to incorporate the same technique in other algorithms. The GA uses a population to maintain a set of different candidates for any locally bounded region (say, a rival group). It would be interesting to see

whether algorithms which don't use a population could be extended with the geometrically local optimizer. Recall, however, that the straightforward inclusion of the GLO in the simulated-annealing algorithm did not work very well (see Figure 3.20 on page 68).

The general technique we described in this chapter is applicable to more problems that have a geometrically determined structure. We will discuss the techniques more formally in Chapter 5. There, we will also show how cartographic rules can be incorporated in the map-labeling GA, and how a GA for two different GIS-problems (line simplification and a certain generalization task) can be designed.

3.5 Conclusion

We described a GA that provides high-quality solutions for the basic map-labeling problem. We compared the GA against other algorithms that reported high-quality solutions for the same problem. It was found that the GA performed as well as, or better than the best known algorithms. In terms of speed the simulated-annealing algorithm outperformed the GA on the densest maps. The basic map-labeling problem only includes the combinatorial aspect of the full map-labeling problem. However, the GA was designed with the inclusion of additional cartographic rules at a later stage in mind. We will explore these possibilities in Chapter 5. In Chapter 4, we will show that our GA has quadratic time complexity on maps with bounded density.

CHAPTER 4

Scale-up behavior of the GA

In Chapter 3, we described a GA that finds good solutions for the basic map-labeling problem. In this chapter, we will investigate the relation between input size l and the amount of computational effort W required to find a solution of sufficient quality. This is called the *scale-up behavior* of the algorithm. The amount of computational effort the GA spends is the product of the number of fitness evaluations E and the time needed to perform a single fitness evaluation e_{fit}: $W = E \cdot e_{fit}$. In the case of the map-labeling GA on maps of fixed density, the latter is easy to compute: in the fitness function each label on the map is checked for an intersection in constant time. Therefore, the total time needed for a single fitness evaluation is $e_{fit} = O(l)$. We will show that for E the relation is also linear; that is: $E = O(l)$. As a result, the total scale-up for the map-labeling GA is quadratic: $W = O(l^2)$.

The number of fitness evaluations E is also the product of two factors: the population size and the number of generations it takes to converge. These two factors also influence the quality of the final solution. For example, population sizing is a trade-off between solution quality and computational resources: one does not want the population to be too small and get bad solutions, but also does not want it to be too high and make the algorithm inefficient. The selection pressure determines the speed of convergence, but pushing too hard may be detrimental to the solution quality. Therefore, the two main questions to be answered are the following. Firstly, what is the *critical* population size needed to obtain a certain level of quality for the final solution? Secondly, using a GA with a critical population

size, how many generations are needed before the population is converged?

In general, it is quite hard to answer these questions. Population sizing is usually done by trial and error. For real-world problems, often little is known about the scale-up behavior. For abstract problems, however, interesting results have been obtained. For instance, a number of studies[71, 66, 95, 6] show that for GAs with a rank-based selection scheme, a separable, uniformly scaled, additively decomposable fitness function,* and a suitably-sized population, the number of generations until convergence is $O(\sqrt{l})$, where l is the size of the problem instance. Furthermore, the gambler's-ruin model[40] predicts that the critical population size for GAs under similar assumptions is $O(\sqrt{l})$ as well. So far, these models have only been investigated using artificial problems.

Our GA for the map-labeling problem was designed with the theoretical insights of these models in mind, in order to make the GA efficient and scalable. In this chapter, we will investigate whether the assumptions that underly the models are not violated too much for the predictions to hold for our GA. In Section 4.1, we examine the model for predicting the number of generations until convergence and the gambler's-ruin model, to study their underlying assumptions. We proceed (in Section 4.2) by checking whether these assumptions hold for our GA. It turns out that, partly because of our use of the geometrically local optimizer, many of these conditions are indeed satisfied. This leads us to expect that the models are applicable, which we then verify experimentally in Section 4.3. Indeed, our experiments show that both the optimal population size and the number of generations until convergence scale roughly as $O(\sqrt{l})$, leading to a linear scale-up for the total number of fitness evaluations. We continue in Section 4.4 with a discussion and conclude in Section 4.5.

4.1 The models

For the number of fitness evaluations E, the following holds:

$$E = n^* \cdot t^*,$$

where n^* is the critical population size needed to obtain a solution of a certain quality, and t^* is the number of generations until convergence when the GA uses a population that is sized large enough ($n \geq n^*$). Both factors (n^* and t^*) therefore determine the scale-up behavior of the number of fitness evaluations spent by the GA.

The remainder of this section examines the models that resulted from the research from others.[66, 95, 71, 6, 40, 35] We extract the underlying assumptions and

*See Section 2.1.4 on page 23 for explanations and definitions of "separable", "uniformly scaled", and "additively decomposable function".

will check in the next section whether they hold for the GA that solves the map-labeling problem. We start in Subsection 4.1.1 with the convergence model, assuming the population size is adequately sized. Subsection 4.1.2 will cover the gambler's-ruin model, which relates the critical population size to a certain level of quality and the input size.

4.1.1 Determination of t^*

There have been several studies[71, 66, 95, 6] on the convergence characteristics of GAs that solve the bit-counting problem, which is to find a bitstring of length l with the maximal number of 1's. The bit-counting problem has the following fitness function:

$$f_{fit}(\mathbf{x}) = u(\mathbf{x}),$$

where

$$u(\mathbf{x}) = \text{the number of 1's in } \mathbf{x}.$$

The bit-counting problem is a very useful problem to study because its properties (for example the distribution of fitness values in a randomly-generated population) can be calculated exactly. Furthermore, it has building blocks of only one gene, which means that no disruption can occur. Using uniform crossover, almost perfect mixing can be obtained. Mixing is called *perfect* when no correlations between genes—introduced by selection—remain after crossover. The bit-counting problem serves as a idealized model of how GAs based on selection and recombination function. We postpone these considerations until later, and start with deriving the number of generations to obtain convergence to the optimal string, assuming a suitably-sized population. The following discussion is based on the papers by Mühlenbein and Schlierkamp-Voosen,[71] Thierens and Goldberg,[95] and Bäck.[6]

The GA that will solve the bit-counting problem uses a selection scheme with constant selection pressure, uniform crossover, and no mutation. Selection schemes for which the analysis holds are rank-based, such as tournament selection and the elitist recombination scheme, but not fitness-proportionate selection. Uniform crossover is used because no disruption can occur, and it mixes the building blocks well. Crossover is always applied: $Pr_c = 1.0$.

Recall from Section 2.1.5 that the selection intensity I is defined as follows (see Figure 2.1 on page 24):

$$I = \frac{\mu_{sel} - \mu(t)}{\sigma(t)}, \tag{4.1}$$

where μ_{sel} is the mean fitness of the selected individuals, and $\mu(t)$ and $\sigma(t)$ are the mean and the standard deviation of the population at generation t, respectively. Crossover does not change the proportion of 1's in the population, so the mean of the fitness of the new population $\mu(t+1)$ is equal to μ_{sel}. Equation 4.1 can then be rewritten as

$$\mu(t+1) - \mu(t) = I \cdot \sigma(t) . \tag{4.2}$$

The selection intensity I depends on the selection scheme.[95] For example, for tournament selection with a tournament size of two, $I = \frac{1}{\sqrt{\pi}}$.

Under the assumption of perfect mixing, the population fitness is binomially distributed, which can be approximated well with a normal distribution. The mean and variance of this distribution are as follows:

$$\mu_t = l \cdot prop(t) , \tag{4.3}$$

$$\sigma^2(t) = l \cdot prop(t)(1 - prop(t)) , \tag{4.4}$$

where $prop(t)$ is the proportion of genes at generation t that have the optimal value (namely 1), and l is the length of the chromosomes. Equations 4.3 and 4.4 can be plugged into Equation 4.2 to yield:

$$prop(t+1) - prop(t) = \frac{I}{\sqrt{l}} \sqrt{prop(t)(1 - prop(t))} . \tag{4.5}$$

Equation 4.5 can be approximated with a differential equation:

$$\frac{dprop(t)}{dt} = \frac{I}{\sqrt{l}} \sqrt{prop(t)(1 - prop(t))} . \tag{4.6}$$

Solving Equation 4.6, we obtain:

$$prop(t) = \frac{1}{2}(1 + \sin(\frac{I}{\sqrt{l}} t + \arcsin(2prop(0) - 1)) .$$

The population is totally converged at generation t^* when $prop(t^*) = 1$. Hence,

$$\frac{1}{2}\pi = \frac{I}{\sqrt{l}} t^* + c , \tag{4.7}$$

where $c = \arcsin(2prop(0) - 1)$. Extracting t^* from Equation 4.7 gives:

$$t^* = \frac{(\frac{1}{2}\pi - c)\sqrt{l}}{I} = O(\sqrt{l}) .$$

For a randomly initialized population, $prop(0) = \frac{1}{2}$ and $\arcsin(2prop(0)-1) = 0$, giving $t^* = \frac{\pi\sqrt{l}}{2I}$.

Miller and Goldberg[66] extended this research by considering noisy fitness functions. Furthermore, they also applied the model to more complex problem domains. They derived equations for domains in which the mean and the standard deviation of the fitness distribution can be expressed as functions of the proportion of converged building blocks. The more complex domain of so-called *concatenated trap functions* was approximated using the prediction for an extension of the bit-counting problem.

The concatenated trap-function problem is defined as follows. Chromosomes use a binary alphabet and are $l = k \cdot m$ long, where m is the number of trap functions and k is a constant. For example, if $k = 4$, a trap function has the following form:

$$f_{trap}(\mathbf{x}_{i+1\ldots i+4}) = \begin{cases} 4 \text{ if } u(\mathbf{x}_{i+1\ldots i+4}) = 4 \\ 3 \text{ if } u(\mathbf{x}_{i+1\ldots i+4}) = 0 \\ 2 \text{ if } u(\mathbf{x}_{i+1\ldots i+4}) = 1 \\ 1 \text{ if } u(\mathbf{x}_{i+1\ldots i+4}) = 2 \\ 0 \text{ if } u(\mathbf{x}_{i+1\ldots i+4}) = 3 \end{cases} .$$

In the general case, a *trap function* is defined as

$$f_{trap}(\mathbf{x}_{i+1\ldots i+k}) = \begin{cases} k & \text{if } u(\mathbf{x}_{i+1\ldots i+k}) = k \\ k-1-u(\mathbf{x}_{i+1\ldots i+k}) & \text{otherwise} \end{cases} .$$

Trap functions with $k > 3$ have also been called *deceptive functions*,[17,31] because information from lower-order partitions (defined over less than k genes) leads away from the optimal schema. In order to find the optimal schema of the partition defined on the genes that are input to the trap function, schemata in the partition should not be disrupted during crossover. In other words, there exists strong linkage between those genes.

The fitness function for the concatenated trap-function problem is a concatenation of m trap functions:

$$f_{fit}(\mathbf{x}) = \sum_{i=0}^{m-1} f_{trap}(\mathbf{x}_{i\cdot k+1\ldots i\cdot k+k}) .$$

Each trap function is defined on k genes and introduces linkage between those genes. The optimal solution can be found by combining the best schema of each partition defined over linked genes. For example, if $k = 4$ and $m = 2$, these partitions are simply FFFF#### and ####FFFF. The optimal solution 11111111 can

be found by searching the best schema in each partition (namely, the schemata 1111#### and ####11111) and combining them.

The concatenated trap-function problem is representative for a whole class of problems that have *bounded difficulty*. That is, a problem can be solved by combining the best schemata of partitions of bounded size. In the more general case, the fitness function is assumed to be a separable, uniformly scaled, additively decomposable function; the fitness function can thus be expressed as follows:

$$f_{fit}(\mathbf{x}) = \sum_{i=0}^{m-1} f_i(x_{i,1}, x_{i,2} \ldots x_{i,k}),$$

where partial functions f_i are defined on at most k genes, with $k \ll l$. Additionally, the functions $f_i(\cdot)$ all depend on different genes and have the same range. The bit-counting problem is the most simple instance in this class of functions. The fitness functions discussed in this chapter are all additively decomposable. The requirement of separability can be relaxed by modeling the interactions between different partitions as noise. Miller and Goldberg showed that adding small levels of noise to the fitness function added a constant to the number of generations until convergence. Therefore, as long as the linkage between genes from different partitions is weak, the model gives a good approximation. The case where a fitness function is "almost" separable is called *semi-separable*. An additively decomposable function is defined as semi-separable if each gene is input to only a small, bounded number of partial fitness functions.

Miller and Goldberg found that their prediction of the convergence behavior for the concatenated trap function closely matched experimental results. This indicates that the prediction $t^* = O(\sqrt{l})$ holds for all problems that satisfy the following assumptions:

- The fitness function is semi-separable, uniformly scaled, and additively decomposable.
- The selection scheme is rank-based.
- Mixing is perfect: no correlations remain between genes of different partitions after crossover.
- There is no disruption of building blocks.

The accuracy of the prediction depends on how well the GA adheres to these assumptions. When the assumptions are not seriously violated, we can expect the prediction still to be quite good.

Note that for the case where the fitness function is exponentially scaled (instead of uniformly), similar studies[63,97] show that the number of generations is linear with respect to the input size: $t^* = O(l)$.

It is also assumed that the population size is large enough and contains sufficiently many building blocks. We will look at this requirement next.

4.1.2 Determination of n^*

In this section, our goal is to derive an equation for the critical population size n^* needed to solve a problem of input length l, given some required level of quality for the final solution.

The issue of determining n^* was investigated by Goldberg et al.,[35] who provided a model of the GA based on statistical decision making. Assuming that the GA would find the best solution if the search progressed in the right direction after the first generation, they obtained a population-sizing equation. Drawbacks of this approach were that it did not model the way a GA can recover from decision errors (explained later) and did not include a building block supply model. Harik et al.[40] extended the model by integrating the so-called gambler's-ruin model and a building-block supply model into the previous model. The following discussion follows the papers by Goldberg et al.[35] and Harik et al.[40]

We again assume that the fitness function is separable, uniformly scaled, and additively decomposable. The assumption of an additively decomposable fitness function allows us to appeal to the Central Limit Theorem and assume that the fitness of the population is normally distributed. Another consequence of the use of an ADF is that we can look at the growth of the optimal schema in a single partition. The GA should converge to a population where all individuals match the same, optimal schema for that partition. During the run, all partitions are searched in parallel. The influence of the contributions to the overall fitness from other partitions can not be ignored. Instead, it is modeled as *collateral noise*. An example will make this more clear.

Suppose we have two schemata $\mathbf{s}_{bb} = 1111\#\ldots\#$, and $\mathbf{s}_c = 0000\#\ldots\#$. The schema $\mathbf{s}_{bb}$ is the schema with the largest fitness contribution from that partition. From now on, we will call this schema the *building block* of the partition under consideration. The inferior schema $\mathbf{s}_c$ is called the competitor. On average, we expect that a comparison between a chromosome $\mathbf{x}_{bb}$ matching $\mathbf{s}_{bb}$ and a chromosome $\mathbf{x}_c$ matching $\mathbf{s}_c$ is decided in the favor of $\mathbf{x}_{bb}$. As a result, the schema $\mathbf{s}_{bb}$ is propagated. However, due to the contributions of the other partitions (the noise), it is possible that a *decision error* is made: the chromosome containing the competitor is chosen. In Figure 4.1, the fitness distributions are shown of the two schemata. The figure shows that the average fitness of the chromosomes matching $\mathbf{s}_{bb}$, denoted by $\overline{f}_{bb}$, is larger than the average fitness of the chromosomes match-

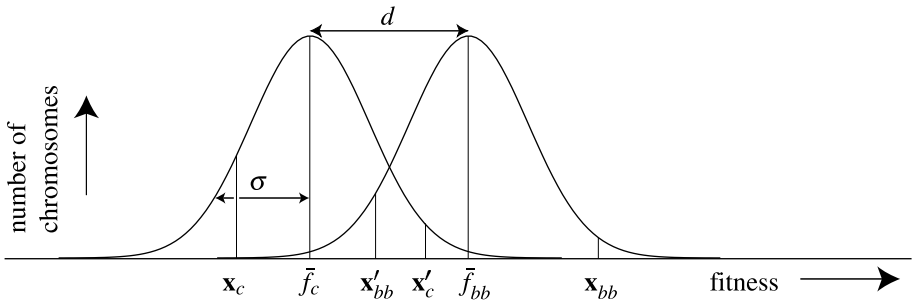

Figure 4.1: The two distributions corresponding with the schemata $\mathbf{s}_c$ and $\mathbf{s}_{bb}$ of a certain partition, and their means $\overline{f}_{bb}$ and $\overline{f}_c$. Solutions $\mathbf{x}_{bb}$, $\mathbf{x}'_{bb}$ match $\mathbf{s}_{bb}$ and solutions $\mathbf{x}_c$, $\mathbf{x}'_c$ match $\mathbf{s}_c$.

ing $\mathbf{s}_c$, denoted by $\overline{f}_c$. The difference between these two averages, $\overline{f}_{bb} - \overline{f}_c$, is called the *signal*. However, it is possible to have two chromosomes $\mathbf{x}'_{bb}$, $\mathbf{x}'_c$ such that $f_{fit}(\mathbf{x}'_{bb}) < f_{fit}(\mathbf{x}'_c)$, due to the fitness contributions from the other partitions. In that case, a chromosome containing the competitor will be chosen and a decision error is made.

Now consider a fixed partition. We will start by deriving an equation for the probability that the correct decision is made for that partition, for a single competition. We will conservatively assume that a competition takes place between a building block $\mathbf{s}_{bb}$, and an other schema $\mathbf{s}_c$ in the partition that has the next-highest fitness contribution. We denote the signal between these schemata with $d = \overline{f}_{bb} - \overline{f}_c$. We will then look at a population-sizing model that views the search of the GA as a series of competitions.

The probability of making the correct decision

First, we show that the standard deviations of both distributions depicted in Figure 4.1 (corresponding with $\mathbf{s}_{bb}$ and $\mathbf{s}_c$) are equal. Since the fitness function is additively decomposable, it can be expressed in the following way:

$$f_{fit}(\mathbf{x}) = \sum_{i=0}^{m-1} f_i(\mathbf{x}_i)\,.$$

Both schemata ($\mathbf{s}_{bb}$ and $\mathbf{s}_c$) are elements of the same partition $\mathbf{p}$, which corresponds to one of the functions $f_i(\mathbf{x}_i)$ and is defined over the genes $\mathbf{x}_i$. The noise that both schemata face comes from the contributions from the $m-1$ other functions. The fitness function is a separable and uniformly scaled ADF, therefore the distributions of fitness contributions of all the functions $f_i(\mathbf{x}_i)$ have the same standard deviation, denoted as σ_{part}. The variance of the fitness function is the sum of the variances of the partial functions. However, the genes of the schema

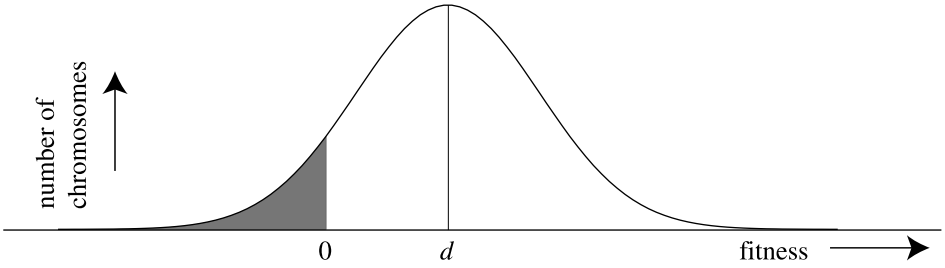

Figure 4.2: The distribution of the difference between two random samples from the distributions from Figure 4.1. The shaded area is the probability of making a decision error.

$\mathbf{s}_{bb}$ (or $\mathbf{s}_c$) are fixed, so the variance of the distribution of fitness values of chromosomes matching $\mathbf{s}_{bb}$ (or $\mathbf{s}_c$), denoted by σ^2, is the sum of the variances of the $m-1$ remaining functions. Hence,

$$\sigma^2 = (m-1)\sigma^2_{part}\,.$$

We can calculate the probability that $f_{fit}(\mathbf{x}''_{bb}) < f_{fit}(\mathbf{x}''_c)$ for randomly chosen $\mathbf{x}''_{bb}$, $\mathbf{x}''_c$ by using the distribution of *differences*, that is, $f_{fit}(\mathbf{x}''_{bb}) - f_{fit}(\mathbf{x}''_c)$ (shown in Figure 4.2). The mean of this distribution is the difference in means of the two schema distributions, and its variance is the sum of their variances. The probability of a decision error Pr_{err} is the probability that the difference in fitness between two randomly chosen chromosomes is less than zero (the shaded area in Figure 4.2).

In Figure 4.3, the cumulative distribution function of the distribution of differences is shown together with Pr_{err}. The probability Pr_{ok} of making the *correct* decision, which equals $1 - Pr_{err}$, is also shown in the figure. Because of the symmetry of the distribution, Pr_{ok} is equal to the probability that the difference is less than $2d$. The value of Pr_{ok} can now easily be found by normalizing to a normal distribution and using the cumulative distribution function of the standard normal distribution. We thus obtain the following result:

$$Pr_{ok} = \Phi\left(\frac{2d-d}{\sqrt{\sigma^2+\sigma^2}}\right) = \Phi\left(\frac{d}{\sqrt{2(m-1)}\sigma_{part}}\right),$$

where $\Phi(\cdot)$ is the cumulative distribution function of the standard normal distribution with zero mean and standard deviation of one.

The gambler's-ruin model views the search of a GA in a single partition as a series of competitions which progresses until either all individuals match $\mathbf{s}_{bb}$, or they all match $\mathbf{s}_c$. The outcome is dependent on the population size and the initial number of building blocks in the population. We will look at the latter point next, before returning to the gambler's-ruin model.

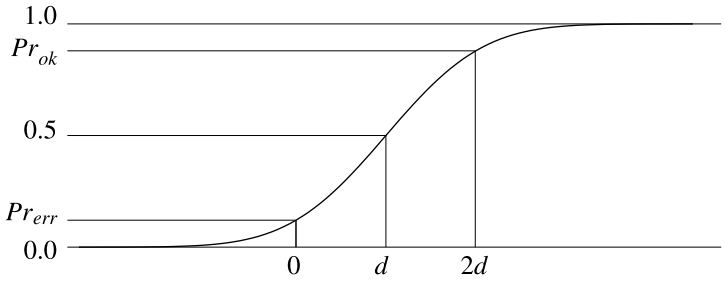

Figure 4.3: The cumulative distribution function of the distribution of differences, shown in Figure 4.2.

The building-block supply model

The initial number of building blocks is denoted x_0. Next, we obtain a lower bound on the expected value of x_0. A simple building-block supply model assumes each element of the partition is equally likely to be (randomly) created in initialization. Therefore, conservatively assuming that only one element of the partition is a building block, the expected number of building blocks in the initial population of size n is:

$$x_0 = \frac{n}{|A|^k},$$

where k is the number of genes of the building block, and $|A|$ denotes the cardinality of the alphabet.

The gambler's-ruin model

The gambler's-ruin model derives its name from the following analogy. A gambler plays a game in a casino (against the house) in which she can bet a certain amount of money. She starts with an initial amount of cash, and keeps on going until either the house is broke, or she has lost all her money. There is a certain probability Pr_{win} she wins a game, earning a fixed amount of money; with probability $1 - Pr_{win}$ she loses, costing her the same amount. The outcome of a series of bets can only be the ruin of either the gambler or the house. The expected outcome depends on the financial reserve of the house, the starting capital of the gambler, and the probability she wins a bet.

For our purposes, a gambler will represent the number of chromosomes in the population that match the optimal schema of a certain partition. The gambler's-ruin model is a one-dimensional random walk between absorbing barriers, corresponding with the loss of the building block (no building blocks left; called the *depletion barrier*) and the existence of the building block in all individuals (n

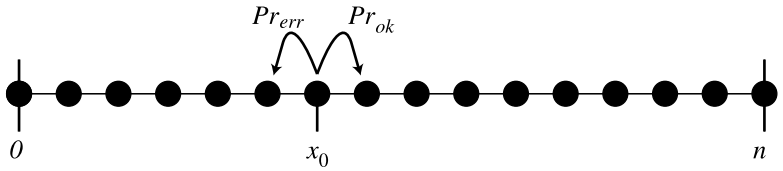

Figure 4.4: The gambler's-ruin model.

building blocks in the population; called the *saturation barrier*). The walk starts at x_0, the number of building blocks in the initial population. Each competition advances the walk to either the saturation barrier (the chromosome with the building block wins the competition, which increases the number of building blocks), or the depletion barrier (a decision error, which decreases the number of building blocks). Figure 4.4 depicts the situation after initialization, before the first competition.

Before discussing how to calculate the probability that the saturation barrier is hit by the gambler, we pause to reflect on the selection scheme of the GA. After all, that is how it is decided who will win the competition. The gambler's-ruin model only considers the effect of selection, under the assumption that mixing is perfect. Mixing of building blocks is called perfect when no correlations remain between genes of different partitions after crossover. The situation in which the concept of competitions is represented most naturally is in an incremental GA, where selection is performed by picking four different individuals and holding two tournaments. This is demonstrated in Figure 4.5. During the selection phase, two competitions take place. Each competition is a tournament between two randomly chosen individuals from the population. One chromosome in each competition contains a building block, the other does not. In the competition at the bottom, a decision error is made: the chromosome with the building block loses the tournament. It is assumed that crossover mixes perfectly and does neither disrupt nor create building blocks. After crossover, the two children replace the losers from the competitions. As a result, if a chromosome containing a building block loses a competition against a chromosome containing a competitor, the number of building blocks decreases by one. If it wins the competition, the number of building blocks increases by one. In the figure, there was one loss and one gain, so the net result is zero. The gambler's-ruin model matches this selection scheme. However, the experimental results presented by Harik et al. show that the more conventional generational replacement scheme also agrees well with the model. As a result, we can assume that any selection scheme with constant selection pressure suffices. This implies a rank-based selection scheme, such as tournament selection or the elitist recombination scheme.

We now turn to the question of determining the probability $Pr(n)$ of the gam-

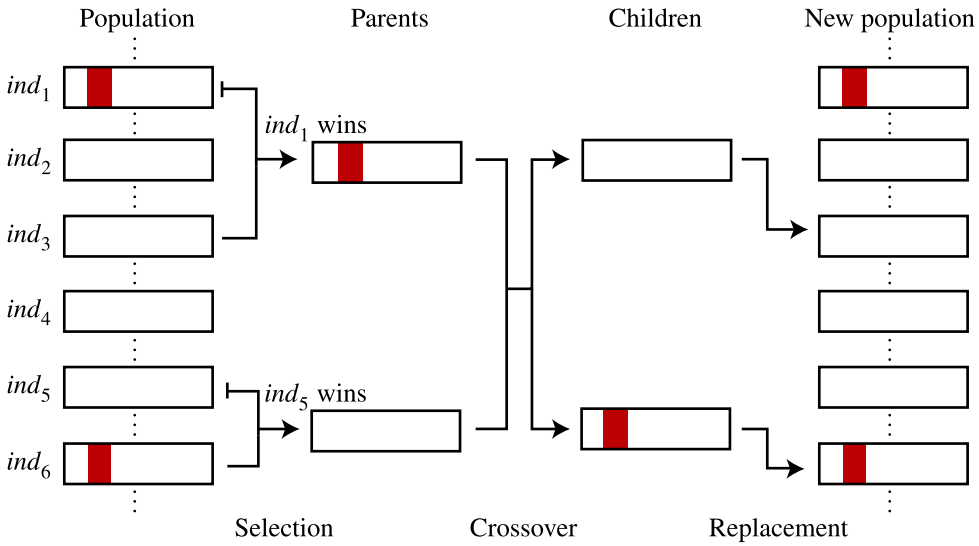

Figure 4.5: Selection in an incremental scheme. Building blocks are shaded. One iteration proceeds by subsequently doing selection, crossover, and replacement.

bler hitting the saturation barrier using a population of size n. Fortunately, this is a known result from random-walk literature:[25]

$$Pr(n) = \frac{1 - (\frac{1-Pr_{ok}}{Pr_{ok}})^{x_0}}{1 - (\frac{1-Pr_{ok}}{Pr_{ok}})^{n}}. \tag{4.8}$$

Here Pr_{ok} is the probability of making the right decision, as calculated before. Note that for our purposes, the depletion barrier need not be absorbing—that is, building block formation is allowed—since $Pr(n)$ can then serve as a lower bound. Assuming no disruption of building blocks occurs, the saturation barrier is absorbing. The probability $Pr(n)$ for a single partition can be used to calculate the outcome of all the partitions searched in parallel. Hence, the following holds for n_{conv}, the number of partitions that converge to the building block:

$$E[n_{conv}] = m \cdot Pr(n), \tag{4.9}$$

where m is the total number of partitions and $E[\cdot]$ denotes the expected value.

Given a measure of desired quality $\alpha = \frac{n_{conv}}{m}$ that denotes the expected fraction of found building blocks, we can find the critical population size to obtain that result. Extracting n from Equation 4.9, assuming a binary alphabet, and approximating Pr_{ok} gives the following approximation of a population-sizing equation (Harik et al.[40]):

$$n \approx -2^{k-1} \ln(1-\alpha) \frac{\sigma_{part} \sqrt{\pi(m-1)}}{d}. \tag{4.10}$$

Note that $\lim_{\alpha \uparrow 1} \ln(1-\alpha) = -\infty$. In other words, finding the optimal solution with absolute certainty requires an infinite population size. This is only to be expected, since a GA is a stochastic algorithm. Harik et al. performed experiments to test their model on various domains, including the concatenated trap-function problem with overlapping partitions (they share genes). They found that the model gave a good estimate of the relation between the quality of solutions (expressed in α) and the population size. The good results on the domain with overlapping partitions show that the assumption of separability of the fitness function can be relaxed to semi-separability.

To find the relation between input size l and the optimal population size, for a certain level of quality α (for example 0.97), we observe that k, σ_{part}, and d are constants and $l = \Theta(m)$. Hence, given some α, we have

$$n^* = O(\sqrt{l}),$$

where n^* is the *critical* population size needed to find a solution with the required level of quality.

This holds for every GA that adheres to the assumptions of the gambler's-ruin model. These assumptions are now restated for convenience:

- The fitness function is semi-separable, uniformly scaled, and additively decomposable.
- The order of partitions, k, is a fixed constant, with $k \ll l$.
- All building blocks are present in the initial population.
- The selection scheme is rank-based.
- Mixing is perfect: no correlations remain between genes of different partitions after crossover.
- There is no disruption of building blocks.

Note that these assumptions subsume the assumptions from the convergence model. In the next section, we will check each assumption to see whether it holds for the map-labeling GA.

4.2 Adherence of assumptions

In order to apply the theoretical models for t^* and n^*, we need to check their assumptions and see if they hold for the GA for map labeling described in Chapter 3.

In the case of map labeling there are certain higher-order relations—defining partitions over many genes—which have to be considered in order to find the optimal solution. Since map labeling is an NP-complete problem, this is only to be expected. We hypothesize that by restricting ourselves to low-order relations—the rival group relations—we are still able to find close to optimal solutions. This hypothesis will be experimentally verified in the next section.

The map-labeling GA described in Chapter 3 is designed with the same theoretical insights in mind that underlie both models. The fitness function is kept additively decomposable. The rival-group relationship defines the partitions in which the optimal schemata are searched. Crossover is made to be linkage respecting. Disruption is minimized by using the geometrically local optimizer, which makes partitions semi-separable. The fitness function can be expressed as a semi-separable ADF:

$$f_{fit}(\mathbf{x}) = \sum_{i=1}^{n_{feat}} free(\mathbf{x}_i)\,, \tag{4.11}$$

where $\mathbf{x}_i$ denotes the genes corresponding to the rival group of the i'th feature. Therefore, we expect the assumptions to hold for our GA.

We will now check each assumption that underlies the models discussed in the previous section.

The fitness function is additively decomposable: Equation 4.11 shows the fitness function can be expressed as an ADF. Note that this is a result of the fact that we can avoid placing penalty functions in the fitness function. Penalty functions can be avoided by placing additional constraints in the geometrical local optimizer instead of the fitness function. Of course, the structure of the problem allows an additively decomposable fitness function.

The order of partitions, k, is a fixed constant, with $k \ll l$: The partitions in the map labeling problem (rival groups) are not of fixed order, but the largest rival group can be taken as a conservative estimate. Moreover, the size of rival groups does not vary too much (on the dense maps used in the experiments, cities have on average about six or seven rivals, with a maximum of 14 rivals).

The fitness function is uniformly scaled: Each partial function can contribute either zero or one to the overall fitness. Therefore, the fitness function is uniformly scaled.

The fitness function is semi-separable: Each gene occurs in a bounded number of rival groups, since the number of rivals is bounded. Therefore, each gene

is input to a bounded number of partial functions, and the fitness function is semi-separable.

All building blocks are present in the initial population: Since building-block formation is possible, and indeed very likely to happen, this requirement can be relaxed.

The selection scheme is rank-based: This requirement is met by using either tournament selection or the elitist recombination scheme. We will experimentally test the predictions for both selection schemes.

Mixing is perfect: Rival crossover can be seen as a kind of uniform crossover on the level of the building blocks. As a result, we can be reasonably confident that mixing is performed adequately.

No disruption of building blocks takes place: In practice, some disruption takes place but is minimized due to the use of the geometrically local optimizer with the effect that it has a marginal influence on the behavior of the algorithm. Since building blocks can be disrupted, the saturation barrier is not absorbing. However, a gambler that reached the saturation barrier will with high probability stay in its proximity.

We conclude that we can be reasonably confident that none of the underlying assumptions is seriously violated. Therefore, we expect to see the scale-up behavior predicted by the models. The next section is devoted to experimentally putting this expectation to the test.

4.3 Results

Experimental data was gathered by running the GA on randomly generated maps. These maps are similar to the maps described in Section 3.1 on page 38. Recall that those maps were square, and embedded on a torus (to remove boundary effects). They were generated by repeatedly selecting uniformly at random a location for a point and its label on the torus, making sure the label did not overlap other labels. Then the labels were removed and the GA was used to find a placement for the labels again. This way we were certain that it was possible to place all labels without intersecting other labels, and the optimum was always the number of cities on the map. The density of the map—the average number of points in a certain area—remains equal for all maps. Therefore, maps with more points are bigger. The density of the maps can be related to the density of a map used in Chapter 3. Recall that those maps had dimensions of 650 by 650, and label dimensions of 30 by 7; the maps we use here are basically of the same density as

those maps with 1000 points, of which an example was shown in Figure 3.4 on page 39.*

In the remainder of this section, functions will be fitted to data by using the Levenberg-Marquardt algorithm for non-linear least-squares fitting,[69] with the data points weighted by their standard deviation.

We present the experimental verifications in a number of steps. First, we investigate the influence of selection pressure. One of the most critical assumptions of the models is that mixing is perfect. If the selection pressure is too high, this assumption is clearly violated. We then look at how the gambler's-ruin model can be fitted to our experimental data. This allows us to derive the critical population size. The number of generations to converge for a GA using the critical population size can subsequently be found. We show that the functions predicted by the models can be fitted well to the experimental results. Finally, the total, minimal number of function evaluations can be derived and is shown to be linear in the input size.

Selection pressure

We start by investigating the selection pressure. A selection pressure which is too low will introduce genetic drift, but a selection pressure which is too high causes hitchhiking and premature convergence. We performed experiments to find the optimal tournament size for the tournament selection scheme. The GA used a population size of 200, which, as will become apparent from the other experiments, is large enough. The termination criterion used for all runs was convergence of fitness, that is, the average fitness becomes equal to the fitness of the best individual. The GA was run five times with different seeds for the random-number generator on five different maps of 1000 points. The average of those twenty-five runs was used as a data point.

The results are shown in Figure 4.6. As can be seen in the figure, the optimal tournament size seems to be two or three, since with larger tournaments the quality begins to drop too much. We used a tournament size of two in all following experiments.

Use of the gambler's-ruin model

Since we have argued that the assumptions are not significantly violated, we should be able to apply the gambler's-ruin model to describe the behavior of the GA for map labeling. Equation 4.8 gives us the probability $Pr(n)$ a certain gambler hits the saturation barrier. We have $m = n_{feat}$ gamblers running in parallel

*Actually, the density of the maps from this chapter is slightly less (equal to the density of a map from Chapter 3 with 940 points), because the maps were generated from different parameters.

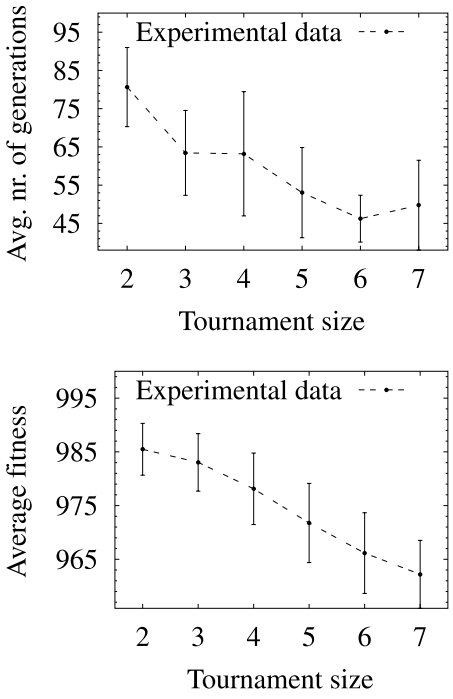

Figure 4.6: Influence of different selection pressures for tournament selection. At the top, the average number of generations is shown. At the bottom, the average fitness is shown.

in the GA (where n_{feat} is the number of features). Each partition corresponds with a rival group. The optimal schema of the partition will place the label of the central point of the rival group in a free position. Given a population size n, the fitness $f_{fit}(\mathbf{x}^*(n))$ of the final solution $\mathbf{x}^*(n)$ is equal to the number of partitions that converge to the optimal schema (given by Equation 4.9: $E[n_{conv}] = m \cdot Pr(n)$). Therefore, the following holds for the expected fitness of the final solution when a population size of n is used:

$$E[f_{fit}(\mathbf{x}^*(n)) = E[n_{conv}] = n_{feat} \cdot Pr(n)\,, \tag{4.12}$$

where $Pr(n)$ is as given in Equation 4.8, and $E[\cdot]$ denotes the expected value.

For maps of size $n_{feat} \in \{200, 500, 1000, 1500, 2000, 4000, 7000, 10000\}$ we ran the GA with population size $n \in \{10, 30, 50, 100, 110, 200\}$. The GA was run three times with different seeds for the random-number generator on three different maps of the same size. The average of those nine runs was used for further computation.

The experimental data for maps of 10000 points is shown in Figure 4.7 (note that the figure is scaled to make 1 the optimum). To this data we fitted Equation 4.12. The closeness of the fit shows that the gambler's-ruin model gives a reasonably close approximation of the relation between population size and the quality of the final solution. All experiments were also done for the elitist recombination scheme, which we used as the selection scheme in Chapter 3. Experimental results for maps with 10000 points are given in Figure 4.8 (again, note that the figure is scaled to make 1 the optimum). Note that the fit shown for elitist recombination is better than the one for tournament selection.

The experimental critical population size

The critical population size for each map of a certain size is found by fitting Equation 4.12 to the experimental data and using the function to find the point where the fitness was 97% of the optimum. Since it is guaranteed that all labels can be placed without intersections, the optimum is n_{feat} labels placed. The critical population size can be calculated as

$$n^* = f^{-1}(0.97 \cdot n_{feat})\,,$$

where $f^{-1}(\cdot)$ denotes the inverse of the fit of Equation 4.12. Since Equation 4.12 is monotone, we do not need to calculate the inverse, but can use a simple binary search to find the result.

The critical population size is calculated in this way for each map size. The results are plotted in Figure 4.9, where a square root function is fitted to verify the prediction of the gambler's-ruin model. This prediction, which states that

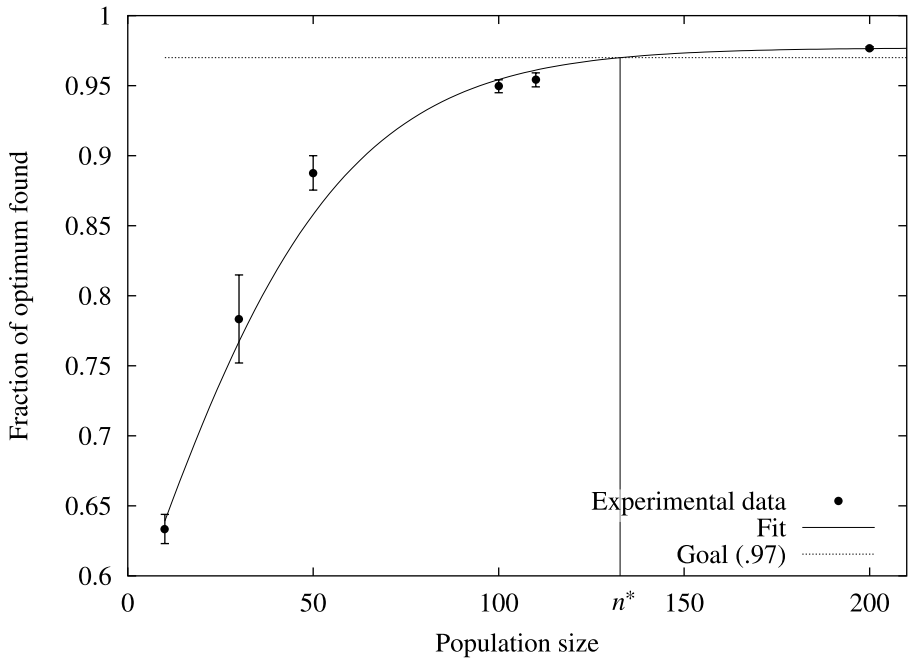

Figure 4.7: Fit of gambler's-ruin prediction to data for maps with 10000 cities. The GA used tournament selection, $Pr_c = 1.0$, $Pr_m = 0.0$. Note that the figure is scaled to make 1 the optimum.

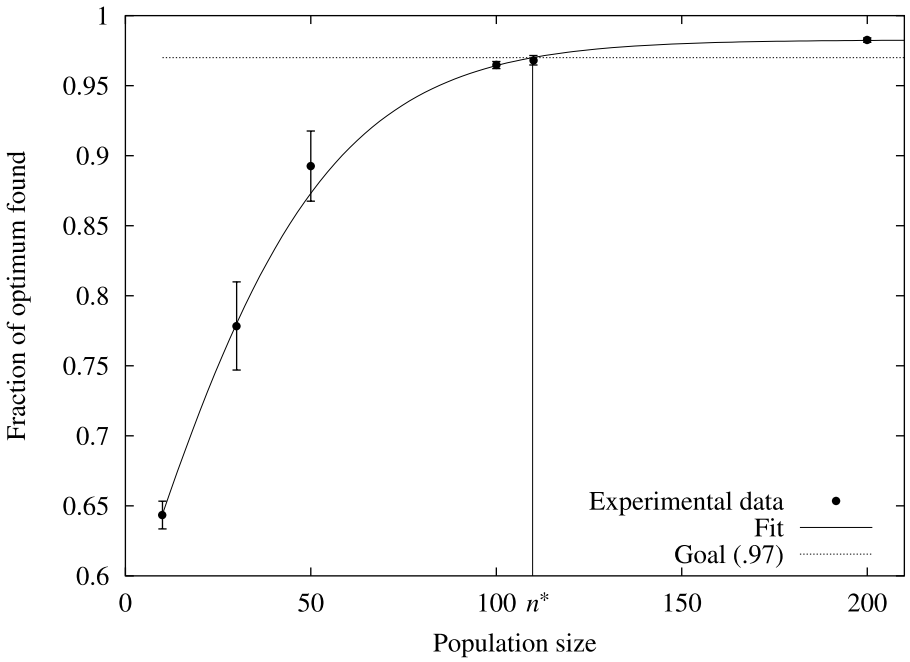

Figure 4.8: Fit of gambler's-ruin prediction to data for maps with 10000 cities. The GA used the elitist recombination scheme, $Pr_c = 1.0$, $Pr_m = 0.0$. Note that the figure is scaled to make 1 the optimum.

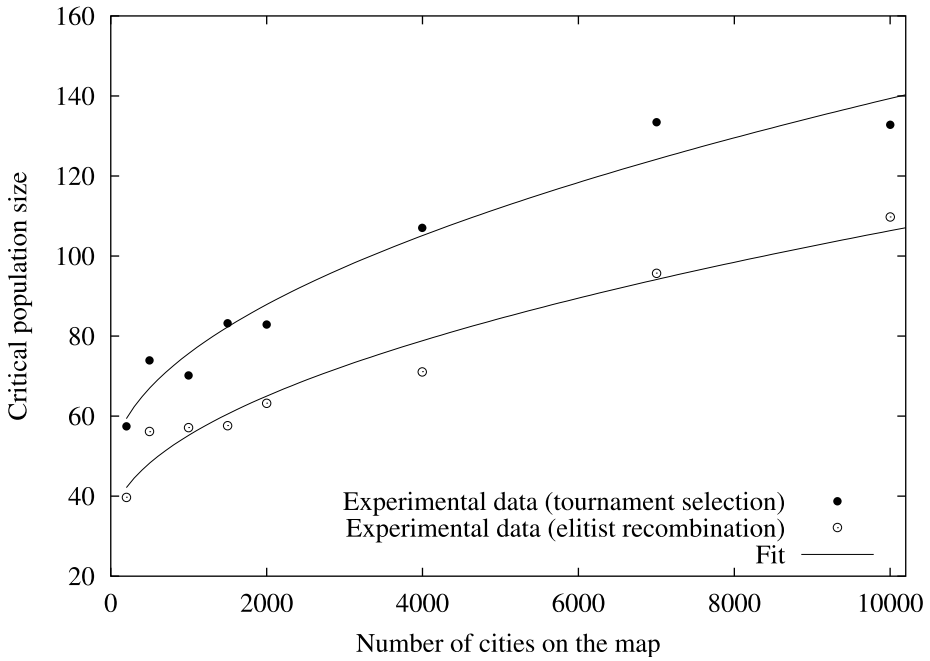

Figure 4.9: Critical population size for a quality of 97% of the global optimum. Shown is experimental data with the fit of the predicted function, for tournament selection and the elitist recombination scheme.

the relation between critical population size n^* and problem length l should be $n^* = O(\sqrt{l})$, is confirmed. Also it is clear that very small population sizes are sufficient.

We also tried the same experiments with elitist recombination instead of tournament selection as the selection scheme. The results are also shown in Figure 4.9 and show the same scale-up behavior. Note that elitist recombination succeeds in finding solutions of the same quality but can use smaller populations than tournament selection.

The number of generations

The number of generations needed to converge, when the population is equal to the critical population size, is obtained in a similar fashion. The critical population size n^* has already been calculated. To perform an interpolation, we need a function that fits the experimental data reasonably well. We use the function $f(x) = O(-1/x)$. After fitting the function to the data, the critical number of generations t^* is given as $t^* = f(n^*)$. This is done for each map size, and the results are shown for both selection schemes in Figure 4.10. The number of generations

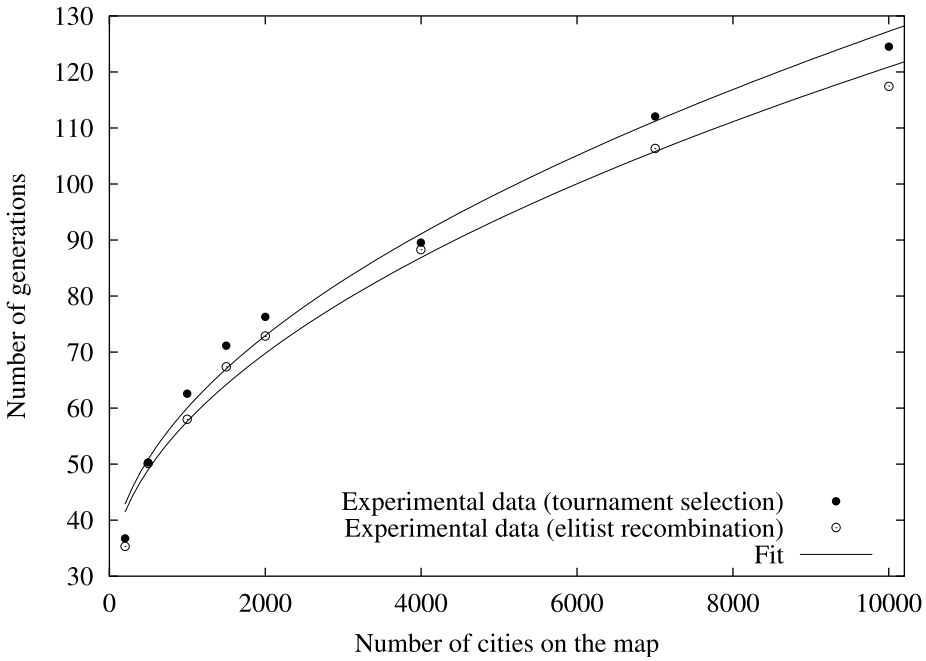

Figure 4.10: Run time in number of generations of GA when using critical population size. Shown is experimental data with the fit of the predicted function, for tournament selection and the elitist recombination selection scheme.

for the elitist recombination scheme, an incremental replacement scheme, is calculated by dividing the number of recombinations by half of the population size. The experimental results are shown with a fit to a square root. The prediction of $t^* = O(\sqrt{l})$ is confirmed.

Total amount of computational effort

Since the number of fitness evaluations is $E = t^* \cdot n^*$, it follows that $E = O(l)$ (the number of evaluations scales up linearly with the problem size). In Figure 4.11 the required number of evaluations for a given map size—the optimal population size times the number of generations until convergence—is plotted, and a linear function is fitted to it.

Figure 4.11 shows that the GA using tournament selection, compared with the GA using elitist recombination, requires more computational effort to obtain the same level of quality. Tournament selection also gives less reliable results, because the spread of the final solutions is larger than with elitist recombination. We now give two possible explanations for this phenomenon.

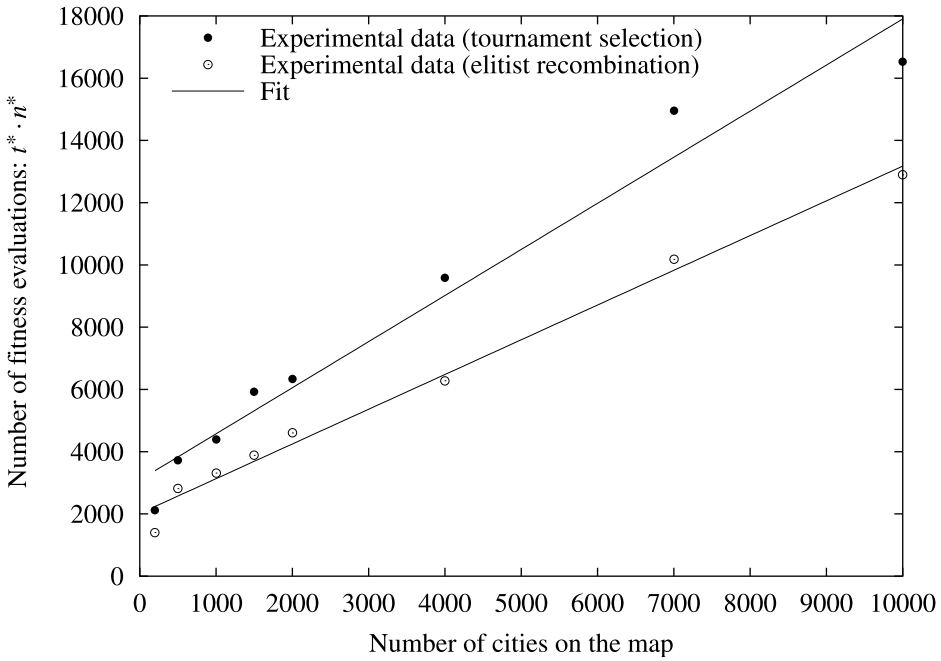

Figure 4.11: The scale-up behavior of the number of fitness evaluations is linear.

The GA using tournament selection uses a generational replacement scheme, whereas the GA using elitist recombination uses an incremental replacement scheme. The gambler's-ruin model most naturally matches an incremental scheme, which may explain that elitist recombination gives a better match.

The GA using tournament selection didn't use elitism of any kind, whereas elitist recombination introduces elitism on the family level. Elitism has the effect of protecting against the loss of good individuals by chance. Therefore, the GA using elitist recombination produces a more reliable result.

4.4 Discussion

In this chapter, we have shown that two important models from literature can be applied to the GA for the map-labeling problem: the assumptions that underlie both the convergence model and the gambler's-ruin model were not seriously violated. We verified the predictions of the models experimentally. We were mainly interested in the relation between the size of the input and the amount of computation needed to obtain a certain level of quality. It would also be interesting to study the amount of computation needed when the density of the maps increases.

To this end, Equation 4.10 can be restated as follows:

$$n = O(2^k \cdot \sqrt{m}) . \tag{4.13}$$

We now observe that k, the size of a partition in the GA for the map-labeling problem, is presumably closely correlated to the size of the rival groups. Equation 4.13 therefore implies an exponential scale-up of the number of fitness evaluations when the density of the map increases. A further indication that this expectation may be correct is given by the experiments from Chapter 3, where the density increased when more points were placed on the map. Figure 3.24 on page 72—note that this figure uses a logarithmic scale for the number of label-intersection tests—suggests an exponential scale-up.

We used artificial maps in this chapter, because it made comparisons easier. It was guaranteed that the maps could be labeled without any remaining intersections, so the fraction of the optimum found could easily be calculated. Maps of real cartographic data are more diverse. They often differ in density at different parts of the map and are normally not as dense as our artificial maps. We can conservatively assume that the largest rival group dictates the population size. However, since the rival groups are quasi-independent, that would mean that for large portions of the map an over-sized population is used. We could lower the population size, with the risk of sacrificing the quality of the relatively rare portions of the map that are most dense. Again we are looking for a trade-off in efficiency and the quality of the solution.

In Chapter 3, it was shown how to incorporate integrated name selection into the GA. Map-labeling problems for which such a GA would be useful are maps with many features, of which only a fraction can be placed. Since the GA that uses integrated name selection does not seem to violate any of the assumptions any more than the normal GA for map labeling, we expect that the models used in this chapter can also be applied in this case.

4.5 Conclusion

In this chapter, we investigated the relation between input size and computational effort for the map-labeling GA described in Chapter 3. The input consisted of uniformly dense, artificially-generated maps, for which it was guaranteed that a labeling existed in which all labels could be placed. The input size was varied by increasing the dimensions of the map and adding more points. To gain insight into the relation between input size and computational effort we used two models from literature. The population-sizing model predicts that the critical population

size n^* needed to find solutions with a specified quality is related to the input size l as $n^* = O(\sqrt{l})$. The convergence model predicts that for a GA using the critical population size, the average number of generations t^* is related to the input size l as $t^* = O(\sqrt{l})$.

The models were examined to investigate the assumptions that underlie them. We argued that these assumptions hold for the case of the map-labeling GA, in part as a result of the use of the geometrically local optimizer. We proceeded by experimentally verifying the predictions. Both predictions were affirmed.

The number of fitness-function evaluations E is the product of the number of generations and the population size. As a corollary of the predictions from the models we can conclude that the relation between the number of fitness evaluations and input size is linear: $E = O(l)$. The total computational effort W is the product of the number of fitness evaluations and the time to perform a single fitness evaluation e_{fit}: $W = E \cdot e_{fit}$, with $e_{fit} = O(l)$. Therefore, the scale-up—the relation between the amount of computational effort and input size—is quadratic: $W = O(l^2)$.

CHAPTER 5

A design framework for GIS problems

In Chapter 3, we designed a GA to solve the problem of point-feature map labeling. We applied several principles from theory to make the GA efficient. This allowed us to study the scale-up behavior of the GA in Chapter 4. In this chapter, we will formalize the techniques that were used, and show how they fit in a general framework for solving problems with a geometrical structure. We will live up to our promise to show how to extend the map-labeling GA to handle cartographic rules, and give results on real cartographic data. We will also apply this framework to two other problems that may arise in automated cartography.

When we want to design usable GAs that solve algorithmic problems arising in automated cartography, what are the characteristics that have to be taken into account? On the other hand, what properties can help us to construct an efficient algorithm?

Firstly, GIS problems often involve many aspects. For example, the combinatorial aspect of the map-labeling problem makes it hard, but additional aspects that satisfy aesthetical or presentation demands, may need to be considered too. Therefore, it is important that the additional constraints can be easily included in the algorithm. GAs 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 (see the discussions in Section 1.1 on page 9 and in Subsection 3.2.3 on page 49). Our method uses geometrically local optimizers to offer a way of including additional constraints by

enforcing them on a local scale.

Secondly, to be practical, the GA 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 of limited use. We also discussed this in Section 3.4, where we argued that an algorithm for a GIS problem has to be robust—that is, it should be possible to integrate the algorithm in a GIS, where it can be used transparently by the user, and easily extended by the maintainer of the algorithm. In order to achieve this, we'll adopt the following approach. We separate the combinatorial aspect from all other aspects and handle the latter in a geometrically local optimizer. This keeps the fitness function simple and additively decomposable, and avoids the problem of tuning the weighting factors in the cost function. We believe that this allows for a pragmatic approach to solve hard GIS problems.

The use of a geometrically local optimizer makes it easy to extend the GA with additional constraints. However, the GA should not only be flexible, but also efficient. Fortunately, the design of efficient genetic algorithms for GIS problems is eased by the geometrical nature of the problems. As a result, the structure of the problem (more precisely, the linkage of the encoding) is often quite clear. The GA is able to exploit this information by using a linkage-respecting crossover and the use of the geometrically local optimizer.

In this chapter, we will outline a framework for solving a class of hard GIS problems. Our goal is to present a flexible method that can be applied to problems where the linkage is reasonably clear. The GA is able to find good solutions by using a linkage-respecting crossover, and is flexible as a result of the customizability of the geometrically local optimizer. Therefore, it can be considered a useful tool to get good solutions for hard problems with reasonable demands on design time. Note that it is not our intention to present a "magic bullet" that is guaranteed to work best for all GIS problems, but rather to show how a good trade-off between design time and solution quality can be reached.

The GA for the map-labeling problem that we described in Chapter 3 demonstrates the general technique. It will serve to exemplify the general framework to solve GIS problems, which we present in Section 5.1. We then apply the framework in three case studies, which were chosen to best show the various advantages of the framework. The first case study, described in Section 5.2, extends the map-labeling problem with cartographic rules, and studies the application of the GA on real-world cartographic data. It shows how the framework is applied and demonstrates its flexibility.

The second case study is discussed in Section 5.3, and deals with line simplification. It exemplifies how to avoid infeasible solutions. An extra criterion (avoiding "spikes") was added to the basic problem, as an additional example of the flexibility of our approach. The GA outperforms the well-known algorithm

of Douglas and Peucker,[21] but also the algorithm of Imai and Iri,[45] which both needed post-processing to deal with the extra criterion.

In Section 5.4, we treat the last case study, which concerns a generalization task where a minimal subset of points has to represent a large set of points. It illustrates (once again) why it is better to use a geometrically local optimizer instead of constructing a complex cost function. We compare our GA to two greedy algorithms, and show that it finds better solutions.

We continue with a discussion in Section 5.5 and conclude in Section 5.6.

5.1 A general algorithm for GIS problems

The GA for the map-labeling problem from Chapter 3 illustrated all the techniques we will formalize in a framework in this section. Recall that the encoding consisted of a chromosome with a gene for every point feature, and each gene stored one allele. The four possible alleles corresponded with the four label positions (see Figure 3.3 on page 37). Initialization was done by assigning a randomly chosen allele to every gene.

The key components of the GA were the following:

- The fitness function expressed only the combinatorial aspect of the problem. (For the map labeling problem, it counted only the number of free labels.)
- All other aspects—the "soft" constraints—were handled with the geometrically local optimizer. (For the map-labeling problem, these were the cartographic rules.)
- Insights about the linkage of the encoding (which expresses the structure of the problem) were derived from the geometrical properties of the problem. (In the GA for map labeling, it was assumed genes were linked if their corresponding points were rivals.)
- Crossover was constructed to be linkage-respecting. It performed a kind of uniform crossover on the level of building blocks. (The GA for map labeling performed crossover on the level of rival groups.)
- Disruption was minimized by using the geometrically local optimizer, which was applied to each gene that was linked to a gene which inherited its allele from a different parent. The geometrically local optimizer acted as a source of building blocks, allowing us to set $Pr_m = 0.0$. (For the map-labeling problem, slot filling tried to resolve label intersections after crossover.)

- The elitist recombination scheme was used, which allowed for a GA in which $Pr_c = 1.0$.

These ideas can also be used for other hard problems encountered in a GIS that have the same characteristics as map labeling (consisting of many aspects, and with linkage that is geometrically determined). We will now formalize these ideas in a general framework.

The general algorithm is an incremental GA and is given in Algorithm 3. In this algorithm, n denotes the population size, and l denotes the number of genes in a chromosome. To start the search, the first population is generated and initialized, to make sure it contains feasible solutions and covers the search space well. The main loop follows, which is iterated until TERMINATE(*Pop*) returns TRUE, after which the best individual of the population is reported. The main loop applies the elitist recombination scheme: parents are randomly chosen, children are created, and from this family of four the two best replace the parents. In the case of ties, children precede the parents. Children are made as follows. First, a crossover mask M is generated, which specifies which genes have to be copied from which parent to which child. Linked genes are placed together in the mask to make the crossover linkage-respecting. This mask is a set which is initially empty. To this set, genes are added until the number of genes in the set is more than half the total number of genes on the chromosome. Genes are added by randomly selecting a gene (which may already be in the set) and placing it, together with linked genes, in the set. Next, the set P is constructed, which holds all the genes on a chromosome which are not in M. These sets are used to perform complementary crossover, which generates the children. The notation $child(x_i) \leftarrow parent(x_i)$ is used to denote that the allele of gene x_i from *parent* is copied to gene x_i of *child*. After crossover, the set B is constructed, which contains all the genes which may be part of a disrupted building block. The GLO is applied to these genes in both children; the procedure REPAIR(·) ensures the solution becomes feasible again, and LOCALSEARCH(·) tries to make the solution better.

Compared to Algorithm 2 on page 26, the function CROSSOVER(·) has disappeared; it is explicitly given in the algorithm. Crossover uses the function LINKED(·) to construct a linkage-respecting crossover mask. Additionally, instead of MUTATE(·), the geometrically local optimizer is used. Note that the functionality of the geometrically local optimizer has been divided into two functions: REPAIR(·) and LOCALSEARCH(·).

In the algorithm, the following subfunctions are used:

INITIALIZE(*ind*): Initialize the individual *ind*, making sure it is a feasible solution which satisfies all problem constraints.

```
1:  generate Pop with n chromosomes                          } Construct initial
2:  for ind ∈ Pop do                                           population
3:      INITIALIZE(ind)                                      }
4:  repeat
5:      randomly choose two different individuals             } ER Scheme —
          p1, p2 from Pop                                      step 1
6:      M ← ∅                                                }
7:      while |M| ≤ (1/2)l do                                } Build crossover
8:          choose at random a gene x_i                        mask
9:          M ← M ∪ {x_i}                                    }
10:         for all genes x_j such that LINKED(p1, x_i, x_j) do
11:             M ← M ∪ {x_j}
12:     P ← all genes of a chromosome that are not in M      }
13:     for x_i ∈ M do                                       } Perform
14:         c1(x_i) ← p1(x_i) ; c2(x_i) ← p2(x_i)            } crossover
15:     for x_i ∈ P do                                       }
16:         c1(x_i) ← p2(x_i) ; c2(x_i) ← p1(x_i)            }
17:     B_M ←all genes in M that are linked to a gene in P   }
18:     B_P ←all genes in P that are linked to a gene in M   }
19:     B ← B_M ∪ B_P                                        } Geometrically
20:     for child ∈ {c1, c2} do                              } local optimizer
21:         for x_i ∈ B do                                   }
22:             REPAIR(child, x_i)                           }
23:             LOCALSEARCH(child, x_i)                      }
24:     replace p1, p2 with the best two from {p1, p2, c1, c2} } ER Scheme —
                                                               step 2
25: until TERMINATE(Pop)
26: report best individual in Pop
```

Algorithm 3: GA for GIS problems.

LINKED(ind, x_i, x_j): Check if gene x_i is linked to gene x_j in individual *ind*, and return TRUE or FALSE. This function expresses the assessment of the designer about the linkage of the encoding.

REPAIR(ind, x_i): Resolve any constraints for gene x_i that make the solution *ind* infeasible.

LOCALSEARCH(ind, x_i): Perform local search on gene x_i in order to make the solution *ind* better, or at least not worse.

TERMINATE(*Pop*): Return TRUE when the algorithm should be stopped, and FALSE otherwise.

Note that the function LINKED($\cdot$) expresses the *assessment* of the designer about the linkage of the encoding, which may or may not be correct. An educated guess about the linkage that is close to the true linkage is vital to the success of the algorithm. The true linkage can usually not be determined. Recall from Section 2.2 that any fitness function can be expressed as the sum of 2^l terms, which each contain a Walsh coefficient. The linkage between any combination of genes corresponds with a single Walsh coefficient. The strength of the linkage is quantified by the magnitude of the coefficient. If there is no linkage, the coefficient is zero. Calculating all coefficients is too time consuming. Even procedures that approximate the most significant Walsh coefficients[93,57] have a considerable computational cost. Fortunately, in GIS problems, as we will see, it is often easy to make a reasonable guess about the linkage. The next sections cover three case studies, which will exemplify how the framework can be used to solve hard GIS problems.

5.2 Point-feature map labeling

The first case study revisits the map-labeling problem, and adds three cartographic rules to it. It gives us the opportunity to show how the framework is used. In addition, it shows one of the main strengths of the framework: the ease at which additional constraints can be handled by a GA that originally solved a simpler problem. In this section, we will mainly be concerned with modifying the geometrically local optimizer to handle the additional cartographic rules. In doing this, we can rely on common-sense intuitions about what constitutes a good labeling in a local sense. Global optimization is handled by the normal mechanism of selection and recombination, which requires no changes. No tuning of weighting factors is needed either. As such, the point-feature map-labeling problem clearly demonstrates the flexibility of the framework.

The map-labeling problem was stated in its basic form in Chapter 3: given a set $P = \{p_1, p_2, \ldots, p_{n_{feat}}\}$ of points which can place their labels in one of four positions, find a labeling such that the number of non-intersecting labels is maximized. The GA for point-feature map labeling can be reformulated in terms of the framework described in the previous section. The encoding $\mathbf{x} = x_1 x_2 \ldots x_{n_{feat}}$ is a chromosome of n_{feat} genes long, using the alphabet $A = \{1,2,3,4\}$, corresponding with label positions. The fitness function is simply the number of non-intersecting labels: $f_{fit}(\mathbf{x}) = free(\mathbf{x})$. There exists linkage between two genes if their corresponding points are rivals.

The components of the framework are filled in as follows:

INITIALIZE(*ind*): Randomly choose an allele for each gene in *ind*.

LINKED(ind, x_i, x_j): Genes x_i and x_j are linked if the points p_i and p_j are rivals.

REPAIR(ind, x_i): Since all possible solutions are feasible, no action is required by this function.

LOCALSEARCH(ind, x_i): If point p_i has an intersecting label, slot filling is performed on gene x_i.

The remainder of this section will incorporate the following cartographic rules into the GA:

Order of preference for label positions. When possible, place a label in the following order of desirability: top-right, top-left, bottom-left, and bottom-right.

Different priorities for cities. Depending on the size of the city, it is placed in one of the following classes of increasing importance: SMALL, MEDIUM, or MEGA. In addition, the capital is classified as a mega city. The importance of the city is shown by the size of the label.

Integrated name selection. When there is not enough room to place all labels, select and delete appropriate labels. A label should not be deleted to make room for the label of a less important city.

Additionally, we will test the algorithm on real-world geographical data of cities in the USA. The classification of the cities depends on thresholds ε_{medium} and ε_{mega}. If the number of inhabitants of a city is less than ε_{medium}, the city is classified as SMALL. If it is between ε_{medium} and ε_{mega}, the city is classified as MEDIUM. If it is more than ε_{mega}, the city is classified as MEGA. The choice of the thresholds $\varepsilon_{medium}, \varepsilon_{mega}$ is discussed later.

Recall that the slot-filling procedure from Chapter 3 works as follows. Slot filling views all possible candidate label positions as *slots*, which can be in two states: EMPTY and FULL. EMPTY signifies that no label of the rivals of the point is intersecting the candidate position corresponding with the slot. FULL signifies the opposite, and shows that the label of the point would intersect another label when placed there. After determining the status of all slots, a free slot is picked at random and the label of the point is placed in the corresponding label position. If no free slot is available, the label remains in its initial position.

The classification in different priorities is reflected in the states used in the slot-filling procedure. Instead of having the states EMPTY and FULL, we now have the states EMPTY, SMALL, MEDIUM, and MEGA. The order is important, because the label of a large city should take precedence over the label of a small city. To make the following discussion less cumbersome, we will use the concepts of class (of a city) and status (of a slot) interchangeably, although there is no class EMPTY.

The GA handles all cartographic rules stated above in the geometrically local optimizer (in the function LOCALSEARCH(·)), although an alteration to INITIALIZE(·) will be needed later.

Consider the labeling corresponding to a child resulting from crossover. The geometrically local optimizer (GLO) is applied to cities that have a rival which was transfered from the other parent.* Placing the label in a preferred position can be done with only a slight alteration: instead of randomly choosing an EMPTY slot, choose among the EMPTY slots in order of preference. For example, if the label can be placed in the top-left and bottom-right position without intersecting other labels, it is placed in the top-left position because it is more preferred. In Chapter 3, we already briefly mentioned how integrated name selection is performed: if all slots had status FULL, the label was deleted. We will have to change this to take the different priorities of cities into account. A label should not be deleted if a less important label could be deleted to make place for it. To achieve this, we try to place a label in a number of steps.

We start by assigning each slot the class of the most important city whose label intersects the corresponding position (see Figure 5.1). If no label intersects a position, the slot is assigned the status EMPTY. Now, suppose we are trying to place the label of a SMALL city. First, we try to place the label in an EMPTY slot. If no EMPTY slot exists, the label is deleted. Next, suppose the feature is a MEDIUM city. Again, we first try to place the label in an EMPTY slot. If this is not possible, we try to place the label in a SMALL slot. Note that the label

*It would be more correct to say that the GLO is applied to each gene in a child which was linked to another gene whose contents was copied from the other parent, where linkage is determined by the rival relationship of the corresponding features.

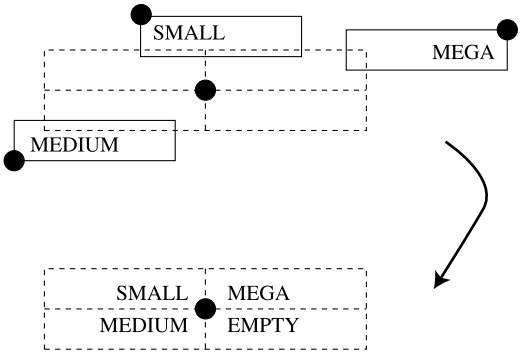

Figure 5.1: Determination of the status of each slot.

will be intersected by the SMALL label, but we assume the SMALL label will be moved or deleted at some later point. If no SMALL slot exists (in other words, all positions are intersected by labels of rivals that are MEDIUM or MEGA), the label is deleted. The same process occurs for a MEGA city, except that, in addition, it is tried to place the label in a MEDIUM slot before deleting it. The algorithm for LOCALSEARCH$(\cdot)$ is summarized in Algorithm 4.

1: For each of the four possible slots for x_i, determine its status. If the slot is not intersected by any other label, the status of a slot is EMPTY. Otherwise, the status of the slot is the class of the most important city whose label intersects the slot.
2: Determine the lowest status in any of the slots. If this status is higher than, or equal to the status of *ind*, then delete the label of *ind*; otherwise place its label in one of the slots with the lowest status.

Algorithm 4: LOCALSEARCH(ind, x_i)

The only remaining question is which of the slots with the lowest status is chosen in the second step of Algorithm 4. An initial attempt is to just place the label in the first slot that has the required status. Slots are checked for their status in the order of preferred positions. There are two problems with this approach:

1. The solution can become worse, because the label may be placed such that it intersects a formerly free label. For example, in Figure 5.2, a MEDIUM label is placed in a SMALL slot and causes the total number of non-intersecting labels to decrease. Fortunately, this is easy to remedy: of all slots with the lowest status we only choose from those that yield the highest overall fitness.

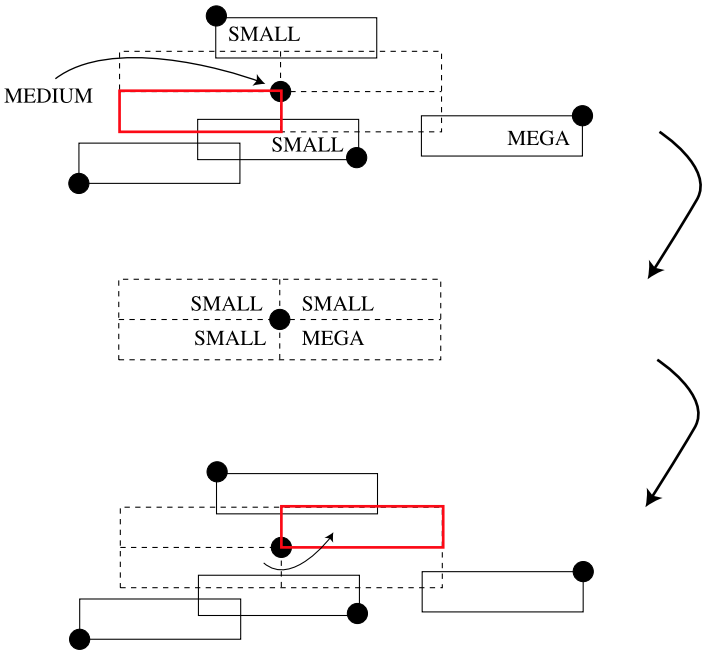

Figure 5.2: First problem with the first attempt to extend slot filling: the fitness can go down when a label is moved to another slot.

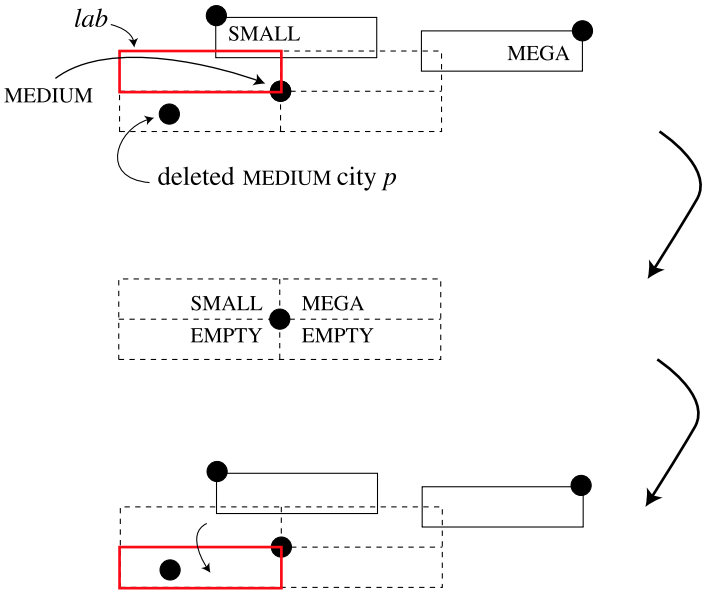

Figure 5.3: Second problem with the first attempt to extend slot filling: a label can force another label to unnecessarily stay deleted.

2. Placing a label may cause a deleted label never to be placed back, even when there exists a better configuration in which it can be placed without intersections. For example, in Figure 5.3, label *lab* was moved to a slot which was EMPTY, but contains a point p with a deleted label. As long as the label stays in that position, the deleted label of p can never be placed back, because in all positions it will intersect *lab*. Since the label *lab* could be placed somewhere else, allowing the deleted label to be placed back, this is unwanted behavior.

The second problem is more difficult to solve than the first one. Note that selection favors solutions in which the labels of important cities are deleted. The label of a MEGA city is likely to be much larger than the label of a SMALL city (since it is written in a larger font), so more labels can be placed by deleting the MEGA label and placing multiple SMALL labels. The GLO has to make sure that the local configurations of labels that are subjected to selection respect the ordering in priorities of the cities. That is, if the geometrically local optimizer ensures that no labels of important cities are deleted in favor of labels of less important cities, the final solution will be good. For MEGA cities, the problem is easily solved: we just never delete labels of MEGA cities. In the case of MEDIUM cities, however, we can not do this, since their labels may have to be deleted to make room for MEGA labels. Therefore, we make sure that from suitable slots, a choice is made from the slots with the minimal number of deleted points with status equal or larger to the point that the GLO is applied to. This way, when possible, it is avoided to place a label in a position that completely covers a point with a deleted label. For example, in Figure 5.3, there are two EMPTY slots the label can be placed in. The minimal amount of deleted points that have higher or equal class than the point with label *lab* for these two slots is zero, so the bottom-right position is chosen, thereby avoiding the position which would cover the point with the deleted MEDIUM label. The algorithm for choosing the proper slot is summarized in Algorithm 5.

1: For each of the slots with the lowest status, count the number of deleted points in the slot whose status is higher than or equal to the status of x_i.
2: Of the slots for which this number is minimized, take the one that yields the highest fitness. If there is more than one with highest fitness, take the slot with the highest preference.

Algorithm 5: CHOOSESLOT(ind, x_i)

Note that there is another small change compared to the GLO described in Chapter 3. In the GLO described there, we checked for an intersection or a deleted

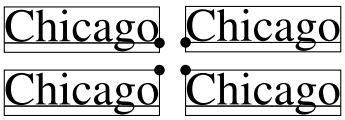

Figure 5.4: The placement model for point-feature labels is adjusted to align the baseline with the feature.

label before applying the slot-filling procedure. Here, we always apply the GLO to ensure the cartographic rules are enforced. Similarly, we have to use a different initializer to ensure the GLO is always applied to each point at least once. Therefore, after assigning each gene a random allele, the GLO is applied to all genes in random order.

Using the framework, cartographic rules can be added to the problem definition by adapting the geometrically local optimizer. The GLO allows for a search of high-quality solutions, without the need to explicitly quantify quality. By choosing the order in which the rules are applied in the GLO, the designer can express her expertise about the (local) quality of solutions. For example, given a local situation with a capital and two small cities, we know that a labeling which places the label of the capital without intersections is best. Contrast this with the method of a cost function as a combination of subfunctions. Such a function would need a subfunction counting the number of small-city labels, a subfunction counting the number of labels of capitals, and weighting factors to combine the two subfunctions such that the cost function yields goods results.

5.2.1 Results

Figure 5.5 illustrates the output of the described GA (the marked regions will be discussed later). The input consisted of 2380 cities in the USA, with their name, coordinates and number of inhabitants. The cities were classified according to ε_{medium} and ε_{mega}, which have to be set by the user of the algorithm. We used $\varepsilon_{medium} = 100000$ and $\varepsilon_{mega} = 800000$. An exception was made for Washington, DC (with approximately 640000 inhabitants), the capital of the USA, which was also considered a MEGA city. Using this classification, there are 2224 SMALL cities, 147 MEDIUM cities, and 9 MEGA cities. In the figure, SMALL cities have font size 10, MEDIUM cities have font size 14, and MEGA cities have font size 20 (absolute sizes are different in the figure due to scaling). Note that the placement model for the point features is changed slightly to align the baseline of the name with the point feature (see Figure 5.4).

It is useful to compare Figure 5.5 with Figure 5.6, the result of the GA with the GLO from Chapter 3. It shows that few important labels are placed when the GLO does not respect the division in classes. As stated earlier, this is to be

Figure 5.5: A labeling for the cities of the USA, using the new GLO. (The circles are used in the discussion on page 117.)

Figure 5.6: A labeling for the cities of the USA, using the simple GLO that was described in Chapter 3.

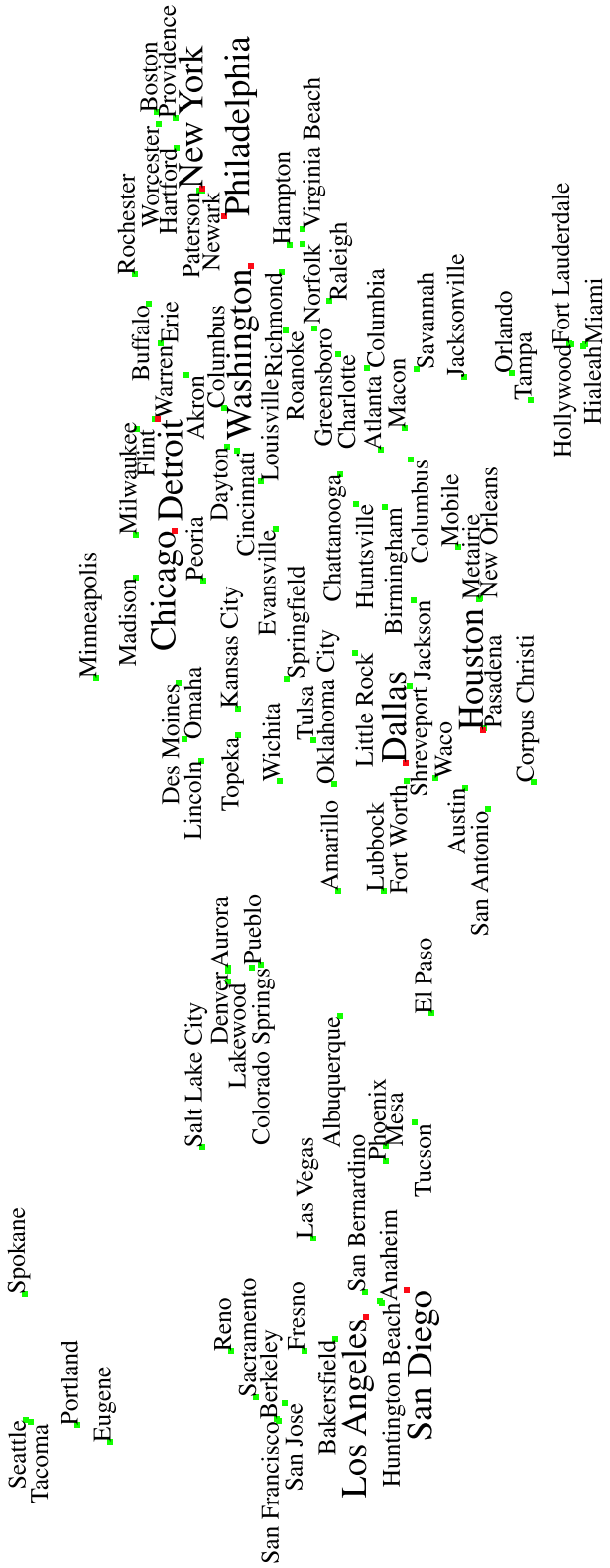

Figure 5.7: A labeling for the cities of the USA, using data with no SMALL cities.

expected, because labels of important cities are larger. Another useful comparison is with Figure 5.7, which is the result of the GA when all SMALL cities were removed from the input. If the geometrically local optimizer correctly respects the priorities of the cities, both figures should display the same number of MEDIUM cities placed. Unfortunately, this is not entirely the case; more MEDIUM labels are deleted than is necessary. From these observations we draw the conclusion that our approach is largely successful in combining the cartographic rules with the combinatorial aspects of the problem. However, a more sophisticated GLO is needed to avoid deleting too many important labels.

Next, we will compare the GA with a local-search algorithm (LSA). It is possible that the GLO introduces such a strong bias in the search of the GA that the combinatorial aspect of the problem is not adequately addressed anymore. In that case, the GA should perform no better than the LSA, which only performs local search. The GA uses a population size of 200 and the termination criterion is as follows. If the last 500 recombinations do not improve the average fitness of the population by one, the run is ended. The local-search algorithm repeatedly tries to improve a single solution. In each iteration, it applies the geometrically local optimizer to every point on the map in random order. If the LSA has run for 50 iterations without making any improvement, it performs a *restart*—that is, it generates a random solution (using the same initializer as the GA) and starts again. Its final solution is the best solution it found during the entire run. The total length of the run is the maximal amount of time the GA spends under the same conditions.

After each run, post processing was done to resolve remaining intersections, although this was never necessary for the runs with integrated name selection. The post-processing step repeatedly chose a label that intersected other labels and deleted it, until all labels were either deleted or without intersections. The label which was deleted was chosen in the following way. If there are SMALL cities that have an intersecting label, the label of the city that is SMALL and has the most intersections is chosen. If there are no SMALL cities, but there are MEDIUM cities with intersecting labels, the label of the city that is MEDIUM and has the most intersections is chosen. Finally, if there are only MEGA cities left with intersecting labels, the label of the MEGA city that has the most intersections is chosen for deletion.

Table 5.1 shows the number of cities for the three maps, divided by class and whether they were deleted. The notation $avg_{\pm sd}$ denotes the average *avg* of the runs, which standard deviation of *sd*. The table shows the results for the GA and for the LSA, both with and without integrated name selection. For the GA, each entry in the table is the average of ten runs. The table shows that the GA using the new GLO is successful in finding good solutions which respect the additional cartographic rules. However, the difference between the input set with and without SMALL cities shows that there is still room for improvement of the GLO. It is also

clear that the use of integrated name selection gives better results than the post processing heuristic. Furthermore, the results of the GA are better than the results of the LSA. This illustrates that the map-labeling problem can not be solved using a powerful GLO alone. The combinatorial difficulties need to be addressed as well. The application of the framework to the map-labeling problem demonstrates how both combinatorial and soft constraints of the map-labeling problem can be handled by a problem solver.

In Figure 5.5, some artifacts can be observed (the enumeration refers to the circles on the map):

1. The labels are tightly packed, because each label is the bounding box of the city name. In the figure, this is clearly demonstrated by the cities Dishman and Moscow. Their labels do not intersect, but only barely. To avoid placing the names of cities too near each other, one can make labels slightly larger by introducing a buffer of space around the text. It becomes easier to associate the name of a city with the feature, at the expense of a decrease of the total number of labels placed.

2. In the figure, the name of the city is not properly aligned with the point feature. This happens because the top-right corner of the bounding box is placed at the coordinates of the point feature. A similar problem occurs when the symbol for the point feature (a little square) overlaps the label. A more detailed label model (for example, the join of the bounding boxes of the individual letters), or a more careful positioning of the label can resolve these problems.

3. At first glance, it is not immediately clear which city in the figure bears the name “Gilette”. This kind of ambiguity should be avoided. This problem can also be resolved by making the labels slightly larger.

Note that these artifacts are the result of the specification of the input of the algorithm, not the design of the GA itself. The GA tries to find a solution that maximizes the number of non-intersection labels. The input specifies what the positions of the labels can be. In the case of point features, a simple, four-position placement model was used. In Chapter 6, line features will be added to the input of the GA, and it has to be specified beforehand what the candidate label positions are.

5.3 Line simplification

The next case study concerns *line simplification*, which is the problem of removing some of the points of a polyline while maintaining approximately the

		SMALL		MEDIUM		MEGA	
with name selection		placed	deleted	placed	deleted	placed	deleted
GA	Full map—old slot filling	$409.5_{\pm 2.3}$	$1814.5_{\pm 2.3}$	$8.3_{\pm 1.1}$	$138.7_{\pm 1.1}$	$0.8_{\pm 0.4}$	$8.2_{\pm 0.4}$
	Full map—new slot filling	$255.7_{\pm 3.7}$	$1968.3_{\pm 3.7}$	$76.1_{\pm 1.1}$	$70.9_{\pm 1.1}$	$9.0_{\pm 0.0}$	$0.0_{\pm 0.0}$
	Map without SMALL	0.0	0.0	$94.0_{\pm 0.0}$	$53.0_{\pm 0.0}$	$9.0_{\pm 0.0}$	$0.0_{\pm 0.0}$
LSA	Full map—old slot filling	310	1914	14	133	0	9
	Full map—new slot filling	191	2033	79	68	9	0
	Map without SMALL	0	0	90	57	9	0

		SMALL		MEDIUM		MEGA	
without name selection		placed	deleted	placed	deleted	placed	deleted
GA	Full map—old slot filling	$124.0_{\pm 4.8}$	$2100.0_{\pm 4.8}$	$43.4_{\pm 4.8}$	$103.6_{\pm 4.8}$	$7.0_{\pm 0.6}$	$2.0_{\pm 0.6}$
	Full map—new slot filling	$116.4_{\pm 2.6}$	$2107.6_{\pm 2.6}$	$61.5_{\pm 4.7}$	$85.5_{\pm 4.7}$	$9.0_{\pm 0.0}$	$0.0_{\pm 0.0}$
	Map without SMALL	0.0	0.0	$83.4_{\pm 1.4}$	$63.6_{\pm 1.4}$	$9.0_{\pm 0.0}$	$0.0_{\pm 0.0}$
LSA	Full map—old slot filling	93	2131	48	99	6	3
	Full map—new slot filling	77	2147	62	85	9	0
	Map without SMALL	0	0	69	78	9	0

Table 5.1: Comparison between GA and LSA under differing conditions.

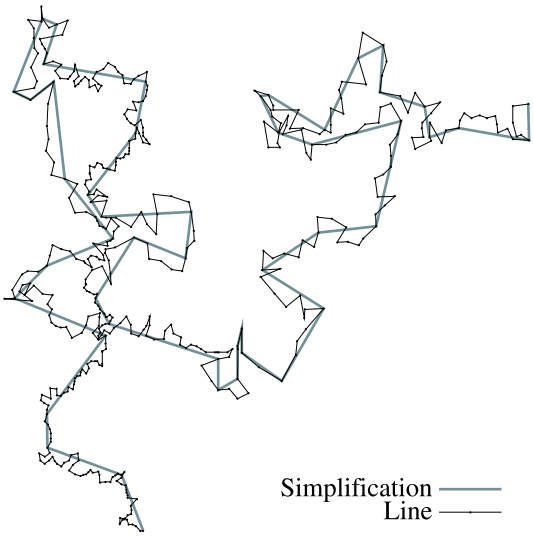

Figure 5.8: Simplifying a line with distance from the original line less than ε.

same shape. See Figure 5.8 for an example. Several algorithms exist for line-simplification problems, such as the iterative refinement heuristic of Douglas and Peucker,[21] and the graph algorithm of Imai and Iri.[45] These heuristics usually give satisfactory results. In some cases, however, unwanted artifacts occur which make the simplified polyline look unnatural. One of those artifacts is the existence of "spikes" (small angles) in the simplification. In this section we show that the framework is flexible enough to deal with such additional demands on the shape of the simplification, whereas extending existing algorithms is not so straightforward. In addition, this case study will demonstrate the use of the procedure REPAIR$(\cdot)$ that will keep solutions feasible during the run of the GA.

We consider the following variant: we are given a polyline (that is, a sequence of n_{points} points, with line segments drawn between successive points) and a parameter ε. The polyline is denoted $P = p_1 p_2 \dots p_{n_{points}}$. The line segments do not intersect each other except when they share an endpoint. The goal is to find a polyline using a subset of the original points (this subset must include 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 maximum deviation is defined as the maximum, over all points p_i, of the Euclidean distance of p_i to the line segment on the simplified line that *simplifies* (see below) that point (see Figure 5.9).

A point that participates in the simplified line is called a *participant.* We say

*Note that the ordering of the points in the polyline is significant. That is, a simplification like $p_1 \dots p_i \dots p_j \dots p_{n_{points}}$ where $i > j$ is not allowed.

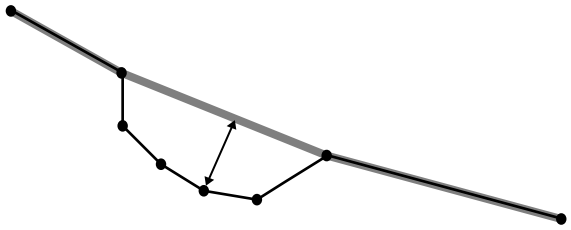

Figure 5.9: The deviation of a point on the original line (thin) from the simplified line (bold).

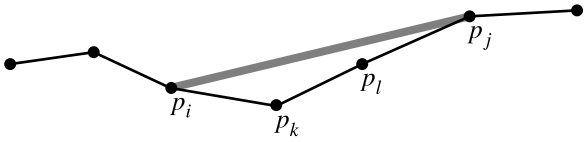

Figure 5.10: The bold segment simplifies points p_i, p_k, p_l, and p_j, but none of the other points.

that a point p_k is *simplified* by segment $\overline{p_i p_j}$ (see Figure 5.10) 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 that we use consists of a gene for every point. For a gene, there are two possible alleles, corresponding to whether the point is a participant or not; the alphabet is binary. The fitness function counts the number of participants. During crossover we want long line segments (possible 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, we require that all angles in the simplification are larger than a certain α (for example 60 degrees). The exception is when the two segments that make the angle also occur in the original line.

Next, we fill in the components of the framework:

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

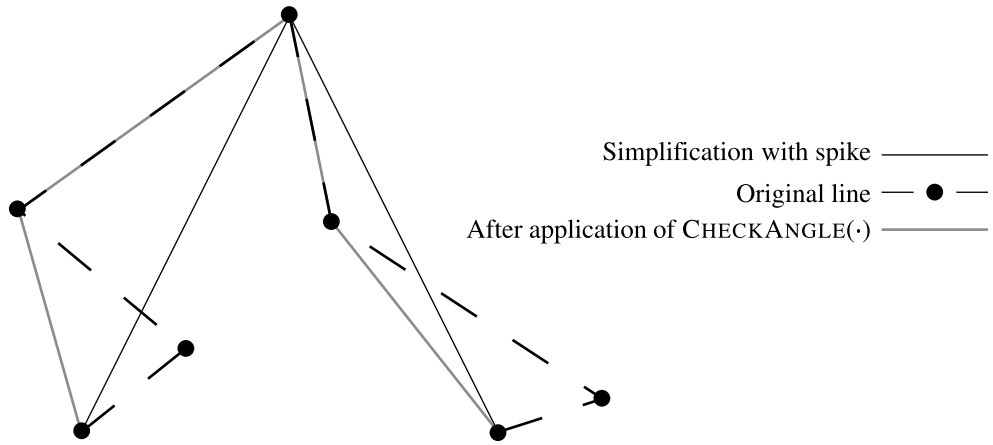

Figure 5.11: The application of CHECKANGLE(·) to remove spikes ($\alpha = 60°$).

LINKED(ind, x_i, x_j): As explained above, genes x_i and x_j are linked if p_i and p_j are simplified by the same segment. Note that this linkage can change whenever the individual changes.

REPAIR(ind, x_i): If there is a conflict, it must be because p_i is at distance greater than or equal to ε from the closest point on the segment that simplifies p_i. Making p_i a participant solves the problem, but a recursive call on all genes linked to x_i is necessary. To avoid spikes (two consecutive segments with a small angle), the procedure CHECKANGLE**(**x_i**)** is called when p_i is a participant.

LOCALSEARCH(ind, x_i): The simplification can be improved if p_i is a participant and can be removed from the simplification without making the solution infeasible. If this is possible, p_i is made a simplified point.

CHECKANGLE(x_i): If the angle between the two segments joining at point p_i is smaller than a given angle, rearrange the simplification (see Figure 5.11). This is done by first turning all points that are simplified by the two segments into participants, 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).

5.3.1 Results

Several line-simplification algorithms are known, such as the iterative refinement method of Douglas and Peucker[21] and the graph algorithm of Imai and Iri.[45] We will briefly describe both algorithms:

Douglas-Peucker algorithm: The algorithm by Douglas and Peucker starts with a simplification consisting of the single segment that connects the start and endpoint of the line. This simplification will probably contain a point that is further away from the segment than the maximally allowed deviation ε. To resolve this, the simplification is recursively refined. From the points that are simplified by the segment, the one with the largest deviation is chosen. The segment is then replaced by two new segments, from the endpoints of the original segment to the chosen point. The new segments are refined until the deviation of all points is less than the maximally allowed deviation.

The Douglas-Peucker algorithm can generate many redundant points that can be removed without making the simplification invalid. In the experiments described further on, we extended the algorithm by applying, in random order, LOCALSEARCH$(\cdot)$ to all points of the simplification that was found by the Douglas-Peucker algorithm.

Imai-Iri algorithm: The algorithm by Imai and Iri casts the simplification problem in a graph model and computes the shortest path to find a valid simplification with a minimal number of points. First, a directed graph is constructed. The node set of the graph contains all the vertices p_i of the polyline. There exists an arc (p_i, p_j), for $i < j$, when the deviation of the points that are simplified by the segment $\overline{p_i p_j}$ is less than ε. After construction of the graph, the shortest path from p_1 to $p_{n_{points}}$ is found by performing breadth-first search, starting in p_1. This path corresponds with a simplification which is guaranteed to have the 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 and the algorithm by Imai and Iri. The latter gives an optimal result when there are no additional constraints. To remove spikes, we applied CHECKANGLE$(\cdot)$ to all participants of the simplifications the algorithms found.

The algorithms were run on twenty randomly-generated lines similar to the one shown in Figure 5.8. The procedure that generated the polyline is recursive and starts with a straight line segment from coordinates $(0,0)$ to $(800,800)$. A random deviation d is 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

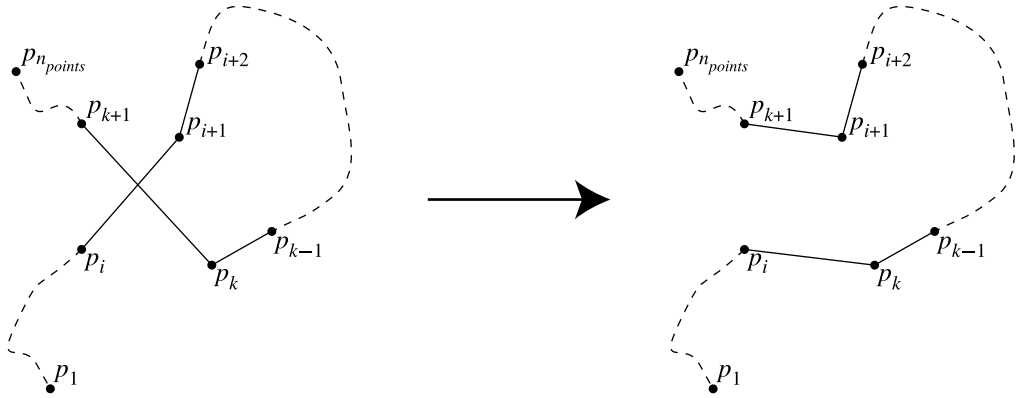

Figure 5.12: Resolving an intersection between segments $\overline{p_i p_{i+1}}$ and $\overline{p_k p_{k+1}}$.

of segments where an intersection occurred. For example, if segments $\overline{p_i p_{i+1}}$ and $\overline{p_k p_{k+1}}$ intersect, the polyline $p_1 \dots p_i p_{i+1} p_{i+2} \dots p_{k-1} p_k p_{k+1} \dots p_{n_{points}}$ is changed to $p_1 \dots p_i p_k p_{k-1} \dots p_{i+2} p_{i+1} p_{k+1} \dots p_{n_{points}}$ (see Figure 5.12).

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 5.13 (where $\alpha = 60^\circ$ and $\varepsilon = 40$) and Figure 5.14 (where $\alpha = 90^\circ$ and $\varepsilon = 20$). The GA generally finds better solutions, thanks to its capability of handling all constraints during the run. Results for various values of α and ε are shown in Table 5.2. The notation $avg_{\pm sd}$ denotes the average *avg* of the number of points in all (twenty) simplifications, which standard deviation of *sd*. The table shows that the GA is outperformed slightly by the algorithm of Imai and Iri when $\alpha = 0$. In that case, spikes are always allowed and the result of the Imai-Iri algorithm is guaranteed to be optimal. When the spike constraint becomes more important (α becomes larger), the GA starts outperforming the Imai-Iri algorithm.

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. For example, it is possible that a simplification that is generated by the algorithms intersects itself. An additional constraint on the output can be that the simplification is free of self-intersections. It is not clear how to alter the algorithm of Imai and Iri to avoid intersections in an efficient manner. For the GA, we can modify the GLO to resolve intersections when they occur, in a similar manner as was done for the spike example. This makes the framework much more flexible and conceptually simpler.

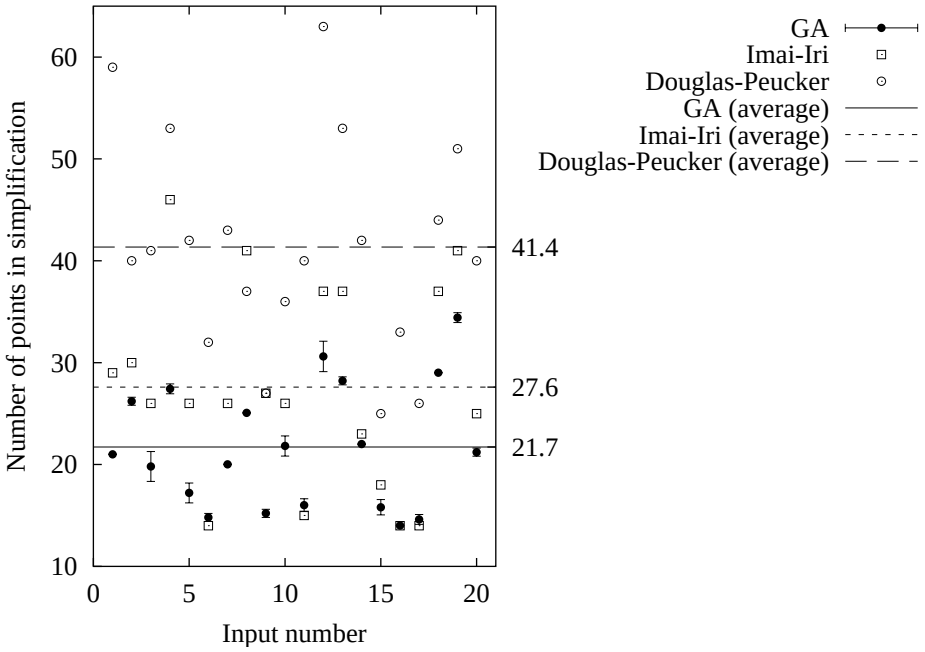

Figure 5.13: Comparison between algorithms for line simplification ($\alpha = 60°$, $\varepsilon = 40$).

$\varepsilon = 20$:

α	0	10	60	90
GA	$46.2_{\pm 11.6}$	$46.0_{\pm 11.8}$	$54.6_{\pm 14.1}$	$76.5_{\pm 20.1}$
I-I	$43.6_{\pm 10.2}$	$44.0_{\pm 10.3}$	$64.5_{\pm 15.3}$	$113.7_{\pm 21.0}$
D-P	$66.9_{\pm 13.3}$	$68.5_{\pm 13.4}$	$88.4_{\pm 18.7}$	$136.1_{\pm 23.9}$

$\varepsilon = 40$:

α	0	10	60	90
GA	$18.9_{\pm 4.6}$	$19.0_{\pm 4.7}$	$21.7_{\pm 5.9}$	$29.2_{\pm 9.9}$
I-I	$18.5_{\pm 4.7}$	$18.7_{\pm 5.1}$	$27.6_{\pm 9.5}$	$76.2_{\pm 25.9}$
D-P	$29.0_{\pm 5.9}$	$29.8_{\pm 6.3}$	$41.4_{\pm 10.2}$	$92.4_{\pm 19.3}$

Table 5.2: Results of the simplification algorithms for different ε and α. ("GA" is the genetic algorithm, "I-I" is the algorithm by Imai and Iri, and "D-P" is the algorithm by Douglas and Peucker.)

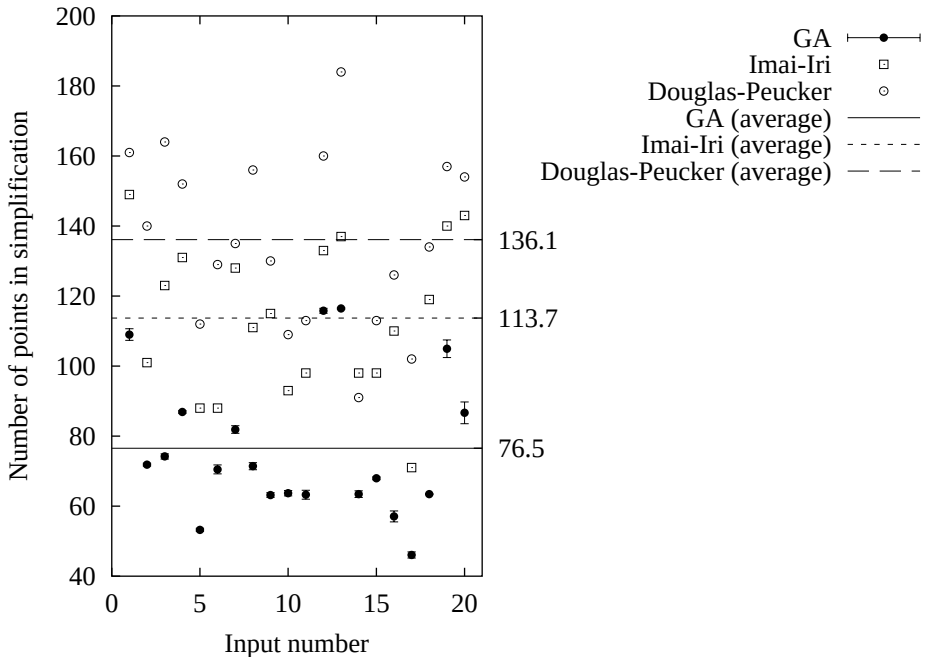

Figure 5.14: Comparison between algorithms for line simplification ($\alpha = 90°$, $\varepsilon = 20$).

5.4 Generalization while preserving structure

The last case study concerns *generalization while preserving structure*, which is the problem of generalizing a given set of objects while maintaining its characteristics. When 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 study will demonstrate that the framework allows us to keep our cost function simple by enforcing feasibility of solutions in the geometrically local optimizer.

The specific problem we study deals with point features: from these points a minimal subset of *representatives* should be chosen such that no two representatives are too close to each other, and every point which is not a representative is close to a representative. The problem is similar to *clustering* problems,[22] in which points are grouped according to some similarity criterion. We formalize our problem as follows. The input consists of a set P of points, and two parameters ε_n and ε_r, with $\varepsilon_n > \varepsilon_r$. Find the smallest subset $R \subset P$ such that:

1. For every two representatives $p, q \in R$ we have $\mathit{distance}(p,q) > \varepsilon_r$.
2. For every point $p \in P$ there is a point $q \in R$ such that $\mathit{distance}(p,q) < \varepsilon_n$.

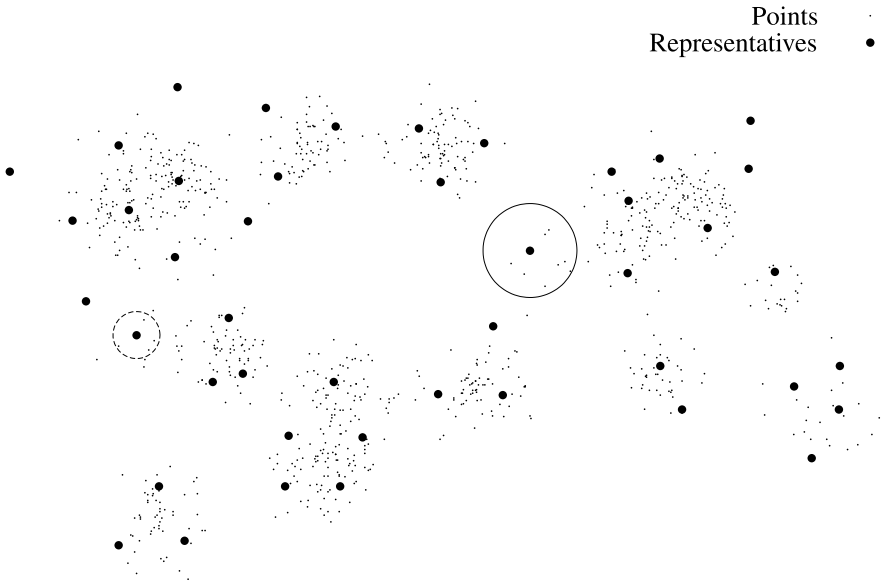

Figure 5.15: A set of points which is represented by a small subset. The solid circle has radius ε_n and contains all the points represented by the bold point. The dashed circle has radius ε_r and should contain only one representative.

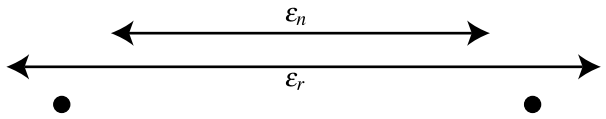

Figure 5.16: If $\varepsilon_n < \varepsilon_r$, a feasible solution may not exist. In this example, neither point can be represented by the other point, but they can not both be representatives.

See Figure 5.15 for an example. Note that we assume $\varepsilon_n > \varepsilon_r$, to ensure a feasible solution exists. In Figure 5.16, a trivial example is given of a problem where $\varepsilon_n < \varepsilon_r$ and no feasible solution exists. The two points are too far apart to represent each other, but too close to both be representatives.

This problem has the interesting property that it 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 that 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:

$$\begin{aligned} f_{fit}(\mathbf{x}) = \quad & w_1 \cdot \mathit{numRepConflicts}(\mathbf{x}) + \\ & w_2 \cdot \mathit{numNormConflicts}(\mathbf{x}) + \\ & w_3 \cdot \mathit{numRep}(\mathbf{x}) \end{aligned} \quad .$$

Here $\mathit{numRepConflicts}(\cdot)$ is the number of representatives that are too close to another representative, $\mathit{numNormConflicts}(\cdot)$ is the number of normal points that are too far from a representative, and $\mathit{numRep}(\cdot)$ gives 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 to balance the effect of the three constraints on the overall fitness. However, the fitness-function has become unnecessary complicated. 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 throughout the run, by using the operator REPAIR$(\cdot)$. As a result, the fitness function can simply be the number of representatives:

$$f_{fit}(\mathbf{x}) = \mathit{numRep}(\mathbf{x}),$$

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 of the encoding is in this problem. 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 5.17.

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, x_i) is applied to each gene.

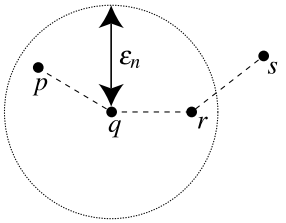

Figure 5.17: Points connected with a dashed line are neighbors. Point p is linked with points q and r, but not with s.

LINKED(ind, x_i, x_j): The genes x_i and x_j are linked if their corresponding points are linked.

REPAIR(ind, x_i): There are two cases:

1. A conflict arises when the point corresponding to the gene x_i is a representative, and it is too close to another representative. This can be resolved by making the point a normal point. For neighbors of the point, new conflicts can arise and therefore REPAIR(ind, x_j) is applied to all genes x_j corresponding with the neighbors of the point.
2. A conflict can also arise when the point corresponding to the gene x_i is a normal point and no representative is nearby. In this case, we can make the point a representative. Since $\varepsilon_n > \varepsilon_r$, no other representatives are close-by, so this does not cause any new conflicts.

LOCALSEARCH(ind, x_i): In this example no local search is performed.

The recursion in REPAIR($\cdot$) is guaranteed to terminate. Figure 5.18 shows the transitions a point can make during multiple applications of REPAIR($\cdot$). A normal point with a conflict is always turned into a representative without a conflict. Likewise, a representative with a conflict is always turned into a normal point without a conflict. It is possible that a point that turned from representative to normal causes a formerly conflict-free, normal point to have a conflict. This happens when the point was represented by the point which was just turned into a normal point. In this case, the transition from normal point without a conflict, to a normal point with a conflict occurs. However, after the point is turned into a representative without a conflict, it need not be changed again. Thus, a point can experience a maximum of three transitions.

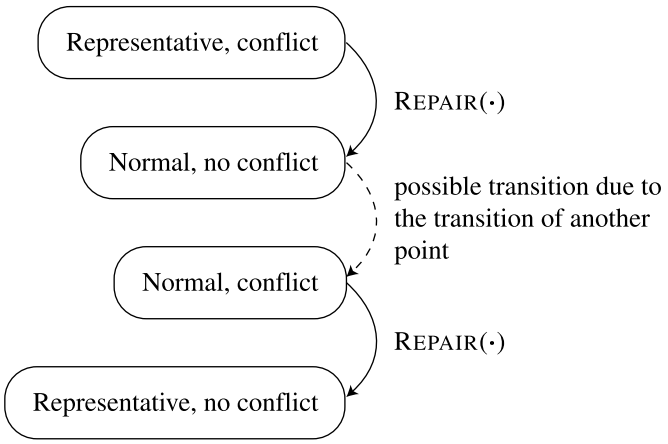

Figure 5.18: State diagram of a point during multiple applications of REPAIR(·).

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.

5.4.1 Results

We generated twenty random maps similar to Figure 5.15. 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. It was then tried to place the following point near the next capital, starting again with the first one after the twentieth. The location of a point, when placed, was calculated using a Gaussian distribution for both the x- and the y-coordinate, centered at the capital. The resulting map gives a more interesting 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 centroid of the points.* The centroid is the point whose coordinates are the averages of the coordinates of all the points. Then, starting with the point closest to the centroid, they are examined in this order. If the point is not represented yet, it is made a representative.

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

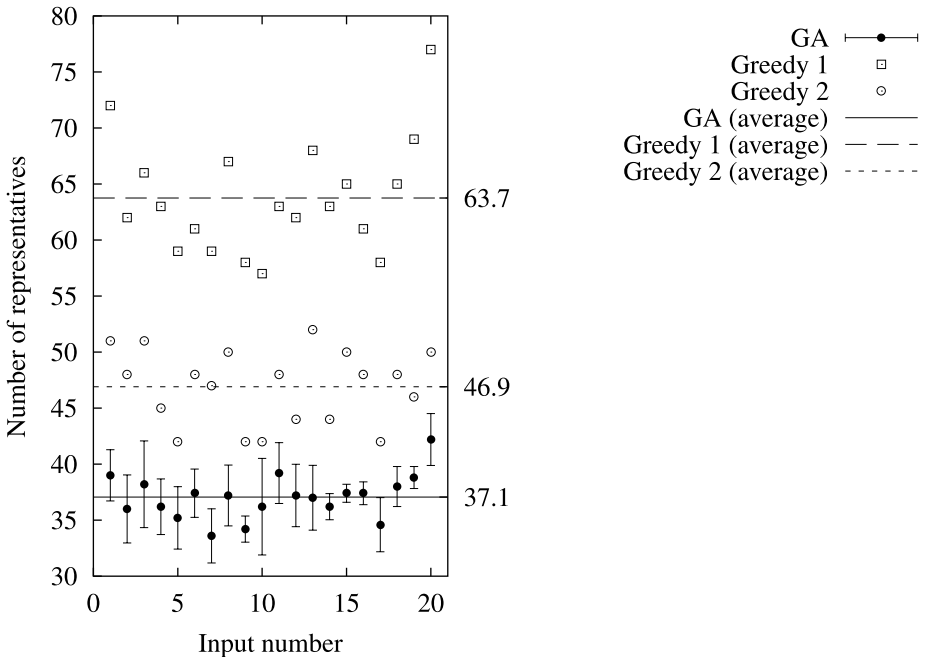

Figure 5.19: Comparison between algorithms for generalization ($\varepsilon_n = 50$, $\varepsilon_r = 25$).

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 5.19 and Figure 5.20, illustrating that the GA performs better than the two other heuristics on every input. 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 was not optimized for speed. For example, it is unlikely that the population size is optimal.

5.5 Discussion

As mentioned in Chapter 1, a problem solver for hard problems in a GIS context has to have the following properties:

1. The problem solver should be capable of giving close to optimal solutions.

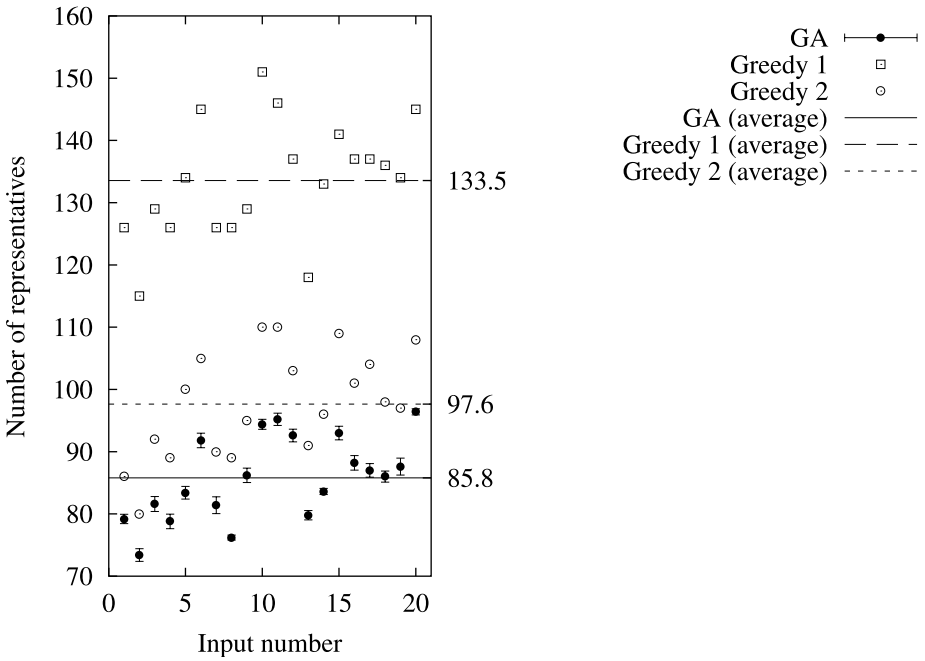

Figure 5.20: Comparison between algorithms for generalization ($\varepsilon_n = 30$, $\varepsilon_r = 20$).

2. It should be possible to extend the problem definition with additional constraints.

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

The first property basically states that when simple, greedy methods fail, a more powerful problem solver is needed. In cases where the problem is combinatorially hard, 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,[60] or tabu search[29] 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, and have proven to be powerful as well as flexible.

The use of geometrically local optimizers in our GA makes it easy to adapt the GA when additional constraints are added to the problem. In a GIS context, 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 solution that is close to the global optimum. No weighting parameters will be needed.

The third property is satisfied by avoiding tunable parameters as much as possible. Since the GA will be used by people who do not have much (if any) experience with GAs, 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, which determines the amount of computational effort that is allocated to the GA. It may be desirable to allow the setting of this parameter. Since the GA gracefully deteriorates in the quality of its solutions, the setting of the population size is a measure for the trade-off between the quality of the final solution and the response time of the algorithm. In other words, if the user wants a solution fast and accepts a lower quality, she can set a low population size. When a high-quality solution is needed, the user sets a high population size and lets the computer run for a night to produce the output. Alternatively, the adaptive population sizing techniques mentioned in Section 2.1.1 on page 21 can be employed to find good solutions without the need for the user to set the population size.

Note that the framework generalizes the techniques—based on theoretical insights—that we used to design the map-labeling GA. In Chapter 4, we were able to investigate the scale-up behavior of that GA. We showed that the assumptions of theoretical models were not seriously violated, which resulted in the prediction of a linear scale-up for the number of fitness evaluations. This prediction was confirmed. We expect that the GAs that are designed by applying the framework have the same characteristics as the map-labeling GA. That is, we expect that they satisfy the assumptions of the models and exhibit the same scale-up behavior. The scale-up behavior is important when the algorithm is integrated in a GIS, because it has to function reliably under all circumstances.

Summarizing, we described a framework in which the structure of the problem needs to be expressed in a function LINKED($\cdot$). When the structure is geometrically determined, as is often the case for GIS problems, this is easy to do. The resulting GAs use a crossover which is linkage-respecting, and a geometrically local optimizer for including additional problem constraints and enforcing feasibility. We think these GAs 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.

Other 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).

5.6 Conclusion

We have proposed a framework for solving hard problems arising in GISs with GAs. Such problems can contain many additional constraints besides the constraint that makes the problem combinatorially hard. In addition, the structure of a problem is often geometrically determined, which means the linkage of the encoding can be assessed with reasonable confidence. In the framework, the linkage is expressed using a separate function, which is used by the GA to make the crossover linkage-respecting. The framework is flexible in that additional problem constraints can be incorporated in the GA by extending the geometrically local optimizer. As a result, an efficient GA—which finds good solutions in a reasonable amount of time—can be designed quickly.

Three case studies exemplified different aspects of the framework. The first case study showed how the map-labeling problem can be extended to a more realistic problem instance with more cartographic rules. The second case study, line simplification, showed that even an optimal algorithm can perform poorly after the addition of a new constraint, whereas the GA is more flexible. It also exemplified how to deal with infeasible solutions. Generalization while preserving structure, the third case study, also demonstrated how to handle infeasible solutions with the geometrically local optimizer. Additionally, it showed how to keep the fitness function simple.

CHAPTER 6

Point and line feature labeling

In Chapter 3, we developed a GA for labeling point features. It was extended with several cartographic rules in Section 5.2. We will now examine the problem of labeling a map that also includes line features. While applications using maps containing only point features exist, for example, labeling a map with measurements, geographical maps usually contain line and area features as well. In order to offer a general approach for solving the map-labeling problem, we should be able to deal with them too. Here, we extend the GA to handle both point and line features. We will return to the question of how to deal with area features in Section 6.4.

Line features are different in a number of ways from point features, and the GA has to be adapted accordingly.

Firstly, finding a placement model for the label is not as easy for line features as it was for point features. For point features there were four fixed positions which are generally considered to be good placements. No such fixed set of positions exists for line features, since each line feature differs in shape. Several guidelines are used to specify a good position for a label (see for example the articles by Imhof,[46] Alinhac,[3] and Youli[116]). In general, a label has to be near the line feature and follow its direction to be associated easily with the feature. On the other hand, if the line feature changes direction a lot (it is "wiggly"), the label should not have a lot of inflection points, since this would make the label less readable. Since there is no a-priori placement model, a procedure has to be

devised that generates candidate positions for a line-feature label that adhere as much as possible to these guidelines.

Secondly, a line feature spans a much larger part of the map than a point feature. For any line feature, the label should be quickly found, starting from any vertex of the line feature. Therefore, it is sometimes necessary to repeat labels for the same line feature. As a result, a way of determining the correct number of labels of a line feature is needed. Furthermore, between labels of the same feature *proximity constraints* have to be enforced: the labels should be neither too close nor too far from each other.

The rest of this chapter is organized as follows. Before we discuss the alterations to the GA, we explain how distances are measured and how the proximity constraints are defined in Section 6.1. Then, in Section 6.2, we describe how the GA is adapted to cope with line features; this involves alterations to the initializer (to place multiple line-feature labels which satisfy the proximity constraints) and the geometrically local optimizer (the slot-filling procedure of Section 5.2 is extended to yield valid labelings for line features). Experimental results are presented in Section 6.3. A discussion of the progress made thus far is given in Section 6.4. The chapter concludes with Section 6.5.

6.1 Measuring distances and proximity constraints

As stated above, we will need to enforce proximity constraints between labels of the same line feature. How these constraints are defined exactly will be explained in this section. Before we can do this, however, we will need to be able to measure the length of the line feature, and distances along it.

Measurements of a line feature. We start by explaining how the length of the line feature is determined. The most straightforward solution, summing the length of all the segments of the polyline, will yield very different results depending on how wiggly the line feature is. In Figure 6.1, two line features are shown. The feature below is more wiggly (and longer) than the one above, whereas they are comparable in length with respect to the label. To circumvent this problem, we calculate the simplification of the polyline and measure along the simplification. We use the Douglas-Peucker algorithm (as described in Subsection 5.3.1 on page 121), using a maximally allowed deviation equal to the height of the label h. We use h since line features on a large scale can have coarser simplifications, thus making the simplification scale-dependent. The length of the line feature is now simply the summation of the lengths of the segments of the simplification.

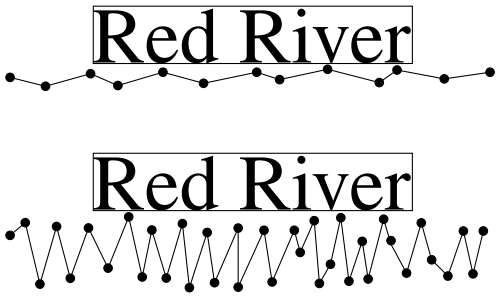

Figure 6.1: The line feature below is more wiggly—and, therefore, longer—than the one above.

The next difficulty is how to measure distances along the line feature: given two labels placed near the line feature, how can we measure the distance between them? Each label is *anchored* to the polyline at a certain vertex of the polyline. How the anchor is determined will be explained in Section 6.2.1 (see page 141). The distance between two labels is the distance between their two anchors, minus the width of the label (see Figure 6.2). What remains is to measure the distance between two anchors. For this, we again use the simplification of the line feature. Note that we use the same simplification as before, in order to keep the measurements constant. As a result, the anchor might not be a vertex of the simplification.

The distance between two anchors on the simplification is the summation of the lengths of the segments of the simplification between the anchors. When the anchor is not a vertex of the simplified polyline, the point on the simplifying segment is used that is the *mapping* of the anchor on the simplifying segment. This is best explained with an example. In Figure 6.3, the dark grey segments belong to the simplification, and we want to measure the distance between anchors p_i and p_j. The distance between p_i and p_j is measured as $length(\overline{p'_i p_{i+1}}) + length(\overline{p_{i+1} p'_j})$, where p'_i and p'_j denote the mapping of points p_i and p_j, respectively, onto the polyline. In the example of Figure 6.3, p_j is already on the simplification, so we have $p'_j = p_j$ To calculate point p'_i, the segments between anchors p_{i-3} (the previous vertex of the simplification) and p_{i+1} are mapped to the segment that simplifies them. This is done in such a way that the relative proportions in length of the four mapped segments ($\overline{p_{i-3} p_{i-2}}$, $\overline{p_{i-2} p_{i-1}}$, $\overline{p_{i-1} p_i}$, and $\overline{p_i p_{i+1}}$) are maintained.

Proximity constraints. The purpose of the proximity constraints is twofold. Firstly, it ensures that the map does not become too crowded (by enforcing a minimal distance between labels). Secondly, it ensures that the user of the map does not have to look too long to find the label of the feature (by enforcing a maximal distance between a label and any point on the line feature). The latter aspect is,

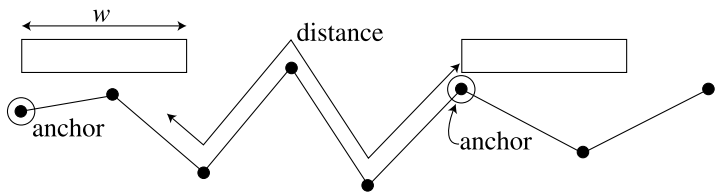

Figure 6.2: Measuring the distance between two labels.

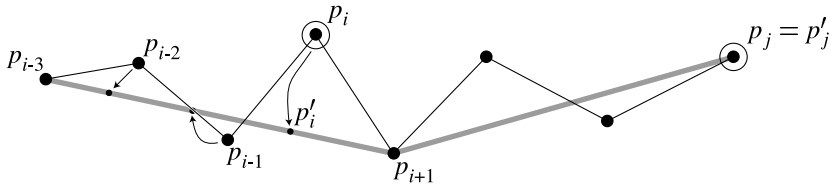

Figure 6.3: Measuring with a vertex that is not part of the simplification (drawn bold and grey).

of course, dependent on the way the user looks for the label. For example, he can examine the whole feature and look for nearby labels, or perhaps he immediately looks for a label in the same color as the feature. Here, we assume the user has seen a part of the line feature and is tracing it until he encounters the label. In this case the length of the (simplified) line feature that was inspected before finding the label is crucial.

There are two proximity constraints. The first one is called the *closeness constraint* and it makes sure the labels are not placed too close to each other.

Definition 5 ***Closeness constraint***

For any two labels lab_1 and lab_2, we have

$$distance(lab_1, lab_2) > \varepsilon_{close} \cdot h,$$

where ε_{close} is a constant set beforehand, and h is the height of the label.

The other proximity constraint is called the *cover constraint* and it ensures that each point on the line feature is near some label, that is, the whole feature is covered.

Definition 6 ***Cover constraint***

For any point p on the line feature, there should be a label lab with

$$distance(p, lab) < \varepsilon_{far} \cdot h,$$

where ε_{far} is a constant set beforehand, and h is the height of the label.

Note that by using *h* the proximity constraints become dependent on scale, ensuring smaller distances on maps of smaller scales. All distances are measured along the simplification of the polyline, as explained above.

6.2 The GA

The GA which we designed for the point-feature map-labeling problem had the following characteristics:

1. Each feature possessed the same number of candidate positions.
2. Each feature could have only one label.
3. All labelings were feasible by definition, because every candidate label position was allowed.

When we add line features to the problem, these properties change:

1. Line features can differ in shape and we want to have the freedom to place the label anyway near the line feature we like. Therefore, we need more candidate positions for long line features than we do for short ones.
2. Labels of a line feature have to be repeated if the line feature is very long, so features can have multiple labels. These labels should not be placed too close to each other, or to far from each other.
3. With the addition of the proximity constraints, not every labeling is feasible anymore. For example, if a label is placed in a certain position and another label is placed in a position which is too close, the solution is not feasible.

Overview of the GA. We maintain our definition of a rival (that is, two features are rivals if their labels can intersect), but it should be noted that different kinds of features can be rivals too. For example, a point feature can be a rival of a line feature.

The encoding we use is an extension of the one that was used for the point-feature labeling GA. A chromosome is a string of genes, where each gene stores the positions of the labels of a feature on the map. For a point feature, the gene just stores a single label position, as before. For a line feature, the gene can hold the positions of multiple labels. This way, the labels of a single feature are kept together, ensuring that crossover does not make the labeling infeasible.

Initialization for point features is the same as in Section 5.2. For line features the proximity constraints have to be enforced, so a different method is needed for initialization.

Crossover is essentially the same as for the point-feature labeling GA: we repeatedly choose a feature and its rivals (a rival group), and transfer the selection from the parents to the children. Since a line feature can span a large piece of

the map, it can also have many rivals. This is reflected in the linkage of the encoding, where relatively large groups of genes can be linked. With the point-feature problem, the solution consisted of building blocks of more or less the same size. Now we can have building blocks which are large (corresponding with line features), combined with those which are small (corresponding with point features).

The fitness function counts the number of conflicts. A conflict is defined as either a label intersection, or a label deletion. Note that this fitness function is conceptually identical to the fitness function from Chapter 3 and Section 5.2.

The selection scheme is not changed: the GA still uses the elitist recombination scheme.

The geometrically local optimizer still uses slot filling, but has to be adjusted to enforce the proximity constraints.

In the next subsections, we discuss the procedure for finding candidate label positions (Subsection 6.2.1), the calculation of the number of line-feature labels (Subsection 6.2.2), initialization (Subsection 6.2.3) and the geometrically local optimizer (Subsection 6.2.4).

6.2.1 Finding candidate label positions

Since there is no a-priori placement model for line feature labels, we need to calculate candidate label positions for each line feature. We choose a discrete placement model, since this extends the GA most naturally from the point-feature case.

Given a polyline, candidate positions for the line-feature label have to be found such that:

- Each candidate position is close to the line feature and follows its direction.
- No candidate position intersects the line feature.
- All candidate positions together cover the line feature, such that a label can be placed more or less anywhere near the line feature.

For simplicity, we consider the line feature in isolation. As a result, it is possible that candidate positions can intersect other features. This can easily be avoided, but requires a more complicated implementation. We will say more about this later (in Section 6.4).

In the map-labeling literature, several approaches have been proposed to generate good candidate positions for line features.[8,113,23] The algorithms by Wolff et al.,[113] and Barrault and Lecordix[8] yield curved labels, which are difficult to implement. Since the combinatorial complexity of the problem is not dependent

on the shape of the labels, straight labels suffice for our research. For the same reason, we are not interested in finding the absolutely best-looking positions possible. Instead, we try to investigate the problems associated with the search for a good map, given a certain search space.

The algorithm by Edmondson et al.[23] also calculates candidate positions for line-feature labels, but it is not guaranteed that these positions cover the whole line feature. In addition, the positions can intersect. Edmondson et al. only placed a single label for each line feature, so their algorithm sufficed for their purposes. Since we want to place multiple labels, their approach can not be adopted. Our method of generating potential label positions (from which the candidates are chosen) is similar to theirs, though.

We propose the following heuristic, which is easy to implement, satisfies the requirements stated above, and worked well in our experiments.

We want to generate candidate label positions for every vertex of the line feature, ensuring the whole line feature will be covered with labels. To do this, we need a way to deduce the local direction of the line feature, so that we can place the candidate position in the same direction. To start, we will generate many potential label positions. Not every label position that was generated will be equally nice. To differentiate between label positions, we therefore need a measure of quality. Based on this measure, we pick the candidate label positions that will be used by the GA. These issues are discussed in the following paragraphs.

Deduction of the local direction and generation of initial baselines. To cover the whole line feature with labels, we will generate a candidate label position for every vertex of the polyline. However, if a segment is very long, no positions will be generated because there is no vertex in the middle of the segment. Therefore, we make sure that the lengths of all segments of the polyline are less or equal to w. We do this by examining the segments of the polyline and breaking up segments longer than w into equal-length subsegments of length at most w. We will call the polyline, after long segments have been broken up, the *extended polyline*.

For each vertex, we generate one or more *initial baseline*s, which will serve as the starting point for the baseline of the label. These initial baselines have to follow the local direction of the line feature.

Given a vertex p_i, we deduce the local direction of the line feature at p_i by taking another vertex p_j (with $j > i$) of the extended polyline which is "close". Two vertices are "close" when the Euclidean distance between them is bounded from below by w and from above by $2w$, where w is the width of the label. The segment $\overline{p_i p_j}$ is the initial baseline, and point p_i is its *anchor*, which we mentioned in Section 6.1. The initial baseline $\overline{p_i p_j}$ is said to *span* the segments $\overline{p_i p_{i+1}}, \ldots, \overline{p_{j-1} p_j}$. See Figure 6.4 for an example of a vertex and its initial baselines. Note that, since

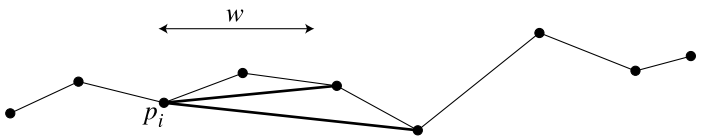

Figure 6.4: The initial baselines (drawn bold) of a certain vertex p_i.

each segment of the extended polyline has length less or equal to w, at least one vertex that is "close" will always be found.

This method succeeds in deriving the local direction of the polyline fairly well. In our experiments, we found that we obtained acceptable results with real-world data of very wiggly rivers like the Mississippi. The initial baselines can't be used directly, since then the labels, when placed, would intersect the polyline. Therefore, the next step will be to move the initial baseline away from the polyline and generate the label position.

Derivation of potential label positions. To avoid that the label position will intersect the polyline, we translate the initial baseline perpendicular to itself until it does not intersect the part of the line feature it spans anymore. We do this for both directions and thus obtain two *shifted baselines* (see Figure 6.5). The label position on the top of the line feature is constructed by using this shifted baseline directly. The position below can not be used directly, since then the label would still intersect the line feature. Therefore, the shifted baseline is translated some more for the distance of the height of the label. Note that we only consider the part of the polyline that is spanned by the initial baseline when we perform the translation. It is still possible that another part of the line feature intersects the potential label position after it was generated. After generating the label position, we check for an intersection. The position will not be considered as a candidate label position if an intersection occurs.*

Now that we have generated many potential label positions, we have to select the best ones to be used by the GA. To do this, we first need a measure of goodness.

Measure of quality. We assign to each label position a measure of quality which describes how well the label is positioned with respect to the line feature. Our primary concern is that each part of the label is close to the line feature, to obtain a good association between the label and the feature. A label that hugs the line feature is better positioned than a label which is some distance from the feature, or at an angle with it. To compute a quality for a label position, we are going

*Note that, at this point, we could check for intersections with other features on the map, too.

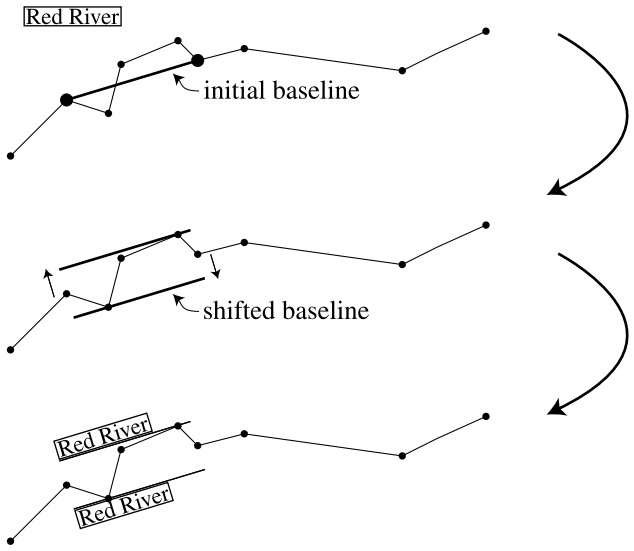

Figure 6.5: Translation of the initial baseline.

to look at the distance between the polyline, and the shifted baseline of the label position—see Figure 6.6. We measure the distance of the baseline b to those segments $s_i s_{i+1} \ldots s_j$ that are spanned by the initial baseline. Long segments should have a larger impact on the quality than small ones, so we weight each segment with the length of the segment and normalize with the total length of the segments. Thus, we define the quality of a label position as follows:

$$\text{quality of label position} = \frac{\sum_{k=i}^{j} \mathit{distance}(s_k, b) \cdot \mathit{length}(s_k)}{\sum_{k=i}^{j} \mathit{length}(s_k)},$$

where $\mathit{length}(s)$ just calculates the Euclidean distance between the endpoints of the segment s, and $\mathit{distance}(s_i, b)$ calculates the Euclidean distance between segments s_i and b).

Using this measure of quality, we are able to distinguish good label positions from bad ones, and we can proceed with the selection of the ones that are going to be used by the GA.

Selection of label positions. We are now ready to provide the candidate label positions of the line feature that will be used in the GA. All potential label positions (two for each initial baseline) are sorted on quality. The best one is returned as a candidate position. The remaining positions are inspected to see if they intersect with the position which was just chosen, in which case they are removed from the list. We remove these positions to avoid that a lot of label positions are

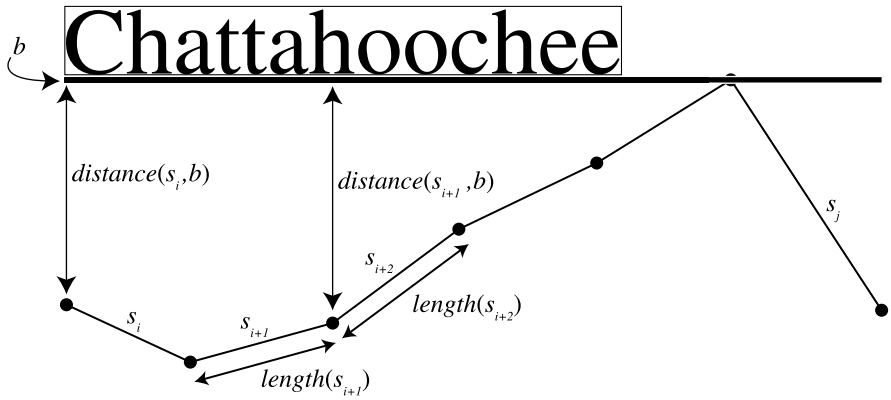

Figure 6.6: The quality of a label position.

chosen that have a similar quality and position, because they are all placed on the same, straight piece of the polyline. We can not just use all label positions, since the performance of our GA drops when a line feature has very many candidate positions. Also, choosing a cut-off value for the quality (to avoid choosing bad positions) can be difficult. By removing intersecting positions, the heuristic is forced to choose the next label on a different part of the polyline.

We keep picking candidates and removing intersecting positions until the list is empty. The chosen candidate positions will cover the whole line feature well since a) each vertex on the extended polyline is likely to generate several potential label positions, b) vertices are never more than a distance of w apart, and c) intersecting positions are removed from the list. This selection method does not guarantee to maximize either the total amount of quality of the candidate positions, or the number of candidate positions, but the results are sufficient for our purpose.

The heuristic is successful in providing candidate positions that are reasonably well positioned. Of course, more sophisticated methods would be able to find positions which look nicer, for example by using curved labels, a better quality measure and an improved selection method. This does not fundamentally change the combinatorial characteristics of the problem, however.

Figure 6.7 shows examples of the application of this heuristic. All candidate positions of (a part of) the Mississippi river are shown, for three different scales of the map. In this chapter, we perform experiments on maps of three scales: the small scale, the medium scale (twice as large as the small scale), and the large scale (three times as large as the small scale). The Mississippi serves as a good example, since it contains parts which are relatively straight, but also parts which change direction a lot. The picture shows that the method is insensitive to quick changes in direction and succeeds in capturing the local direction of the line feature. Furthermore, it produces acceptable results for a wide range of scales.

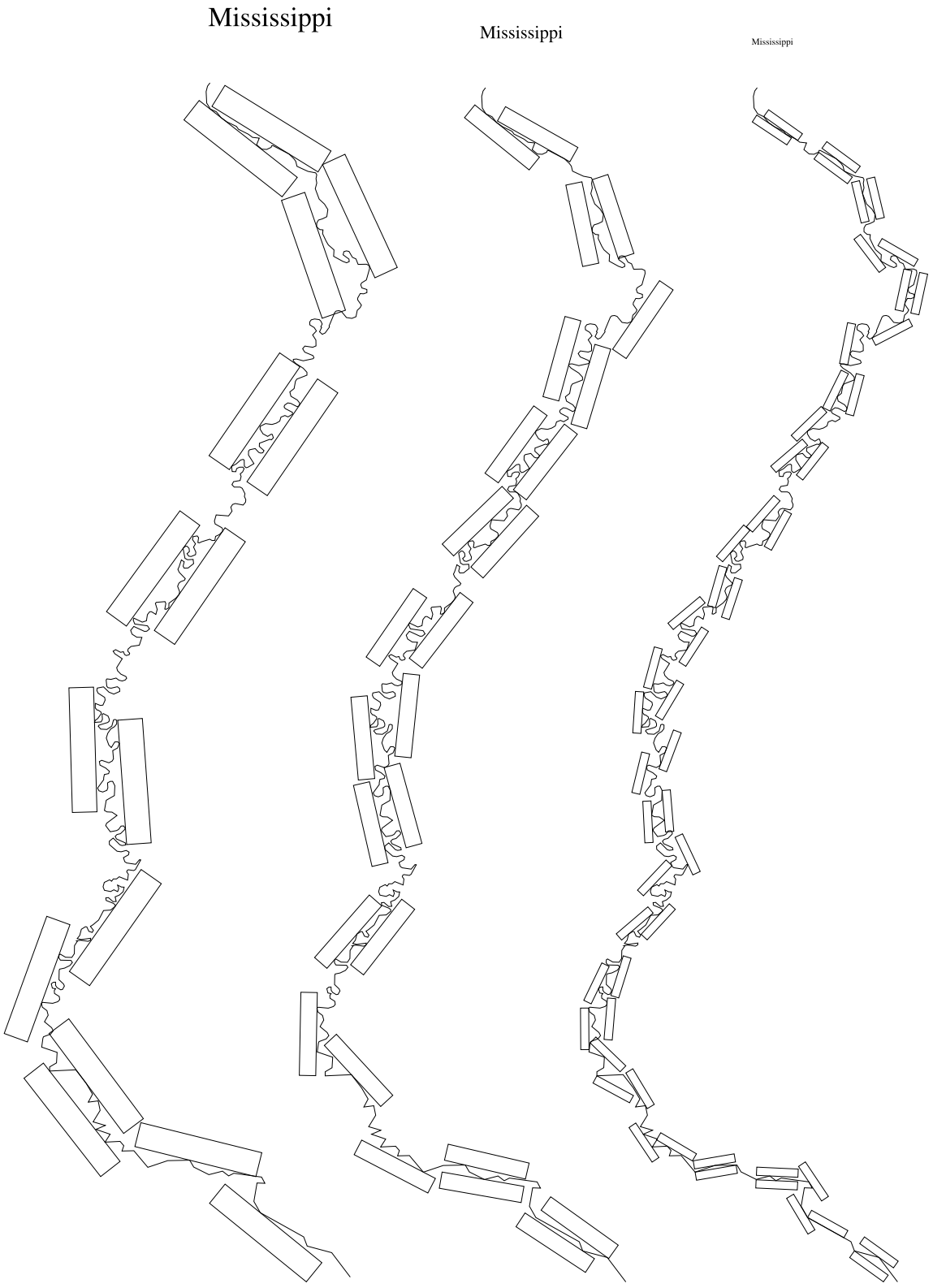

Figure 6.7: Results of the heuristic that calculates candidate label positions for a part of the Mississippi river.

```
1: for segment ∈ line do
2:   if length(segment) > w then
3:     split segment in smaller segments with length ≤ w
4: list ← ∅
5: for point p_i ∈ line do
6:   for point p_j ∈ line with j > i do
7:     if length(p_i p_j) > w and length(p_i p_j) < 2w then
8:       generate initial baseline p_i p_j
9:       generate two shifted baselines
10:      use the shifted baselines to generate two label positions
11:      add the two positions to list
12: while list not empty do
13:   best ←element of list with highest quality
14:   report best as a candidate position
15:   remove all positions from list that intersect best
16:   remove best from list
```

Algorithm 6: Finding label position candidates.

The heuristic is summarized as pseudo code in Algorithm 6. In the algorithm, h and w denote the height and the width of the label, and *line* denotes the line feature.

6.2.2 The number of labels for a line feature

Each line feature must have a certain number of labels placed. If there are too few, the user of the map has to spend too much time searching for the label before he knows the name of the line feature. If there are too many, the map becomes unnecessarily crowded. So the question is: given a certain line feature, how many labels should it have? Clearly, this is dependent on the scale of the map, since a river which takes up only a small piece of the map needs less labels than a river which spans the whole map. For simplicity, we assume there is an ideal, scale-dependent distance d which is just the right trade-off between crowdedness and the time the user is searching. Ideally, labels would be placed evenly spaced, and with a distance of d between them (see Figure 6.8). For our experiments, we use $d = 2\varepsilon_{close} \cdot h$. Recall that the closeness constraint specifies that labels should be more than $\varepsilon_{close} \cdot h$ apart. This value is more-or-less arbitrary (chosen because it produced good results), but that does not really matter. We are interested in providing the techniques needed to label maps with a GA. The optimal value can be set by the user of the GA, if necessary. The ideal distance can not exceed the

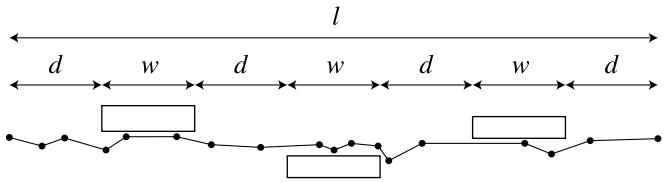

Figure 6.8: Coverage of line feature by labels and space.

two borderline cases that are given by the proximity constraints, though. Thus, we should have: $\varepsilon_{close} \cdot h < d < \varepsilon_{far} \cdot h$, which implies that $\varepsilon_{far} > 2\varepsilon_{close}$.

Next we show how to use the ideal distance d to calculate n_l, the number of labels to be placed. Suppose we have a line feature that has a (simplified) length of l on which we will place n_l labels, evenly spread. The total length of the line feature is consumed by the labels (given by their width) and the space between them. Assuming the space between two labels can be exactly d, the following equation holds:

$$l = n_l \cdot w + (n_l + 1) \cdot d\,.$$

In reality, it will seldom be possible to place the labels such that the space between two labels is exactly d. Therefore, we round n_l up to the smallest integer. Substituting for d and rearranging terms we get:

$$n_l = \left\lceil \frac{l - 2\varepsilon_{close} \cdot h}{w + 2\varepsilon_{close} \cdot h} \right\rceil .$$

Whether it is actually possible to place this many labels depends on the candidate label positions that were calculated and the values of the proximity constraints. However, since the heuristic we use to calculate candidate positions was able to provide a position for every part of the line feature, this is not a problem when the values for ε_{close} and ε_{far} are reasonable.

We would like to stress that the use of a GA does not make it necessary to fix the number of labels beforehand. In the next section, we discuss how n_l labels can be placed without violating the proximity constraints. As will become clear in that section, it is also possible to use a GA which keeps the number of labels for a line feature variable.

6.2.3 Initialization

We now have two essential ingredients to do initialization for line features: the feature has a number of candidate positions and we know the number of desired labels n_l. Initializing a line feature provides it with a feasible labeling, which

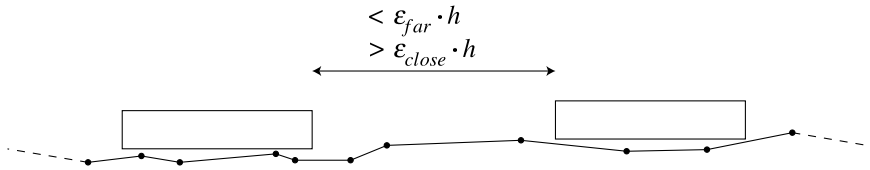

Figure 6.9: Constraints on the distance between two labels.

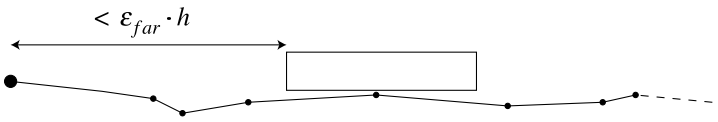

Figure 6.10: Constraints on the distance between the first vertex and the first label.

means we have to place the n_l labels such that the proximity constraints are not violated. Since the candidate positions cover the whole line feature, we can be reasonably certain such a labeling exists if the constants of the proximity constraints are properly set. That is, the values for ε_{close} and ε_{far} should not be too small. If ε_{close} and, hence, d is too small, the required number of labels that have to be placed near the line feature becomes too large. If ε_{far} is too small, it may not be possible to place the labels such that the whole line feature is covered. Of course, the values for ε_{close} and ε_{far} should also satisfy aesthetical demands. A value for ε_{close} which produces good-looking labelings is, in our experience, always large enough. For the maps we tested our algorithm on, a value of ε_{far} that is larger than approximately $2.5\varepsilon_{close}$ was large enough to always find a labeling.

The proximity constraints can be violated in two ways. The first case occurs when the distance between two consecutive labels is too large or too small (see Figure 6.9). The other case occurs when the distance between the first vertex of the line feature and the first label (or the last vertex and the last label) is too large (see Figure 6.10).

The problem is to place n_l labels that avoid these cases. We will first attempt a naive method, examine its shortcomings, and improve on it to find a more satisfactory method.

We denote the candidate positions by $c_1, c_2, \ldots, c_{n_c}$, where n_c is the number of candidates for the line feature. The candidate positions are ordered by their anchors, which is the left-most vertex that defined their initial baseline, as was described in Subsection 6.2.1, on page 141. The anchors are ordered using the ordering of the vertices of the polyline. Two candidates that use the same anchor (one on top of the line feature and the other one below) are ordered arbitrarily.

For now, we ignore the number of labels n_l that have to be placed, and only concern us with producing a labeling that satisfies the proximity constraints. We

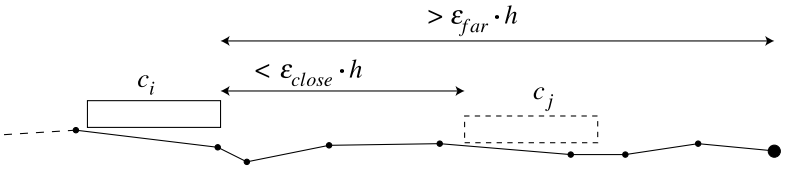

Figure 6.11: An example where the first attempt for initialization does not produce a feasible labeling.

can try to do so by starting on one end of the line feature, and place labels until we reach the end of the line feature. The first label has to be placed at a candidate position that is near enough the first vertex of the polyline. Other than that requirement, the candidate position can be chosen freely. The next label has to be placed such that it is not too close to the label which was just placed, but also not too far. It is placed at a randomly chosen candidate position that adheres to these constraints. We thus proceed until we reach the end of the line feature, and stop when the last label placed is near enough to cover the last vertex of the polyline. We now have a labeling where all labelings are neither too close nor too far from each other, and every vertex of the line feature is close enough to some label. Thus, the labeling satisfies the proximity constraints.

Actually, we can not be really certain that the labeling produced satisfies the proximity constraints, since it is possible that the method got stuck (see Figure 6.11). In the figure, a label was placed at c_i. The distance between the label and the extreme point is still too large, so another label should be placed. The only available position (c_j) can not be used, however, since a label placed there would be too close to the first label. However, for reasonable values of ε_{close}, ε_{far} and w this problem does not arise. A bigger problem is that, since we ignored the required number of labels, there is no guarantee that n_l labels are placed. We can, of course, use a kind of back tracking to systematically try out all combinations, but that would be much too costly. Instead, we will extend this method to deal with these two problems, but first we state the problem more formally. To do this, we use a graph (the so-called *constraint graph*) that expresses when a label can be placed, depending on the candidate positions already used. In the graph, there is a node for each candidate label position. There is an arc between two nodes when labels that are placed at the corresponding positions do not violate the proximity constraints. We also add nodes n_{start}, n_{end} for the extreme points. There is an arc from n_{start} to another node when the corresponding label is close enough to the first vertex of the polyline. Similarly, there is an arc from a node to n_{end} when the corresponding label is near enough to the last vertex of the polyline.

Definition 7 ***Constraint graph***

The constraint graph $G = \langle V, A \rangle$ *of a given polyline is a directed graph, defined as follows:*

- *The node set V consists of all the candidate label positions c_i of the line feature, and two additional nodes n_{start} and n_{end} that correspond to the first and the last vertex of the line feature.*
- *There is an arc $(c_i, c_j) \in A$ when $\varepsilon_{close} \cdot h < distance(c_i, c_j) < \varepsilon_{far} \cdot h$.*
- *There is an arc $(n_{start}, c_i) \in A$ when $distance(p_{start}, c_i) < \varepsilon_{far} \cdot h$, and an arc $(c_i, n_{end}) \in A$ when $distance(c_i, p_{end}) \leq \varepsilon_{far} \cdot h$, where p_{start} and p_{end} are respectively the first and the last vertex of the line feature.*

All distances are measured along the simplification of the line feature.

Figure 6.12 shows an example of a line feature and its candidate label positions (top of figure), together with the constraint graph for that feature (middle of figure).

As an example, we describe the naive method, presented earlier, with this graph. We start at n_{start}. We choose at random one of the outgoing arcs from this node. We follow the arc to a node c_i. From this node, we again choose an arc at random, and proceed to the next node. We keep doing this until we can follow no more arcs. We hope we have now arrived at node n_{end}, meaning the labeling is feasible.

The problem now is to find a path from n_{start} to n_{end} of length $n_l + 1$. This is done by deriving from G a new graph G^* (the so-called *path graph)* that will express all possible paths in G.

Definition 8 ***Path graph***

The path graph $G^* = \langle V, A^* \rangle$ *for a given constraint graph $G = \langle V, A \rangle$ is a directed, weighted multigraph, defined as follows:*

- *The graph uses the same node set V as G.*
- *If there is a path $c_i c_j \ldots n_{end}$ of length ℓ in G, then there is an arc (c_i, c_j) with weight ℓ in G^*.*

In Figure 6.12 (bottom), G^* is shown together with the line feature and G. Note that there can be multiple arcs between two nodes (hence, it is a multigraph), albeit with different weights.

We can now solve our problem by looking for an arc from node n_{start} with weight $n_l + 1$. If one exists, we randomly choose one and find the next node. By definition, this node will have at least one arc with weight n_l. We can again

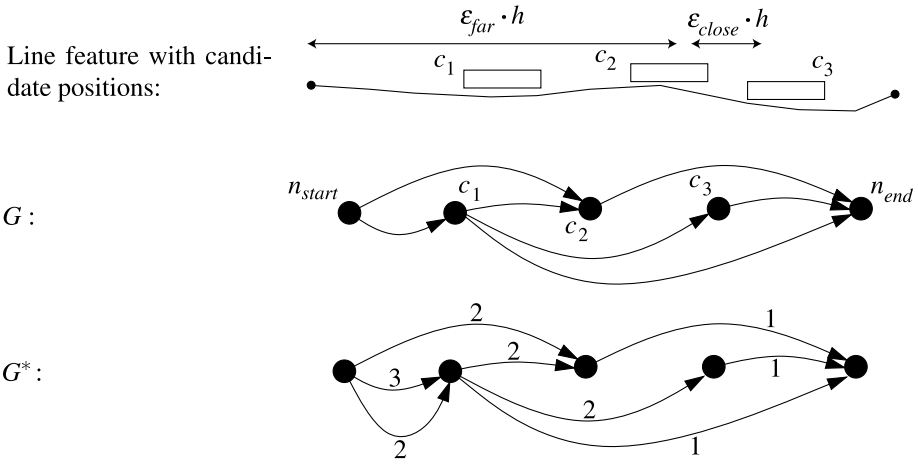

Figure 6.12: A line feature with its candidate positions, the constraint graph G, and the path graph G^*.

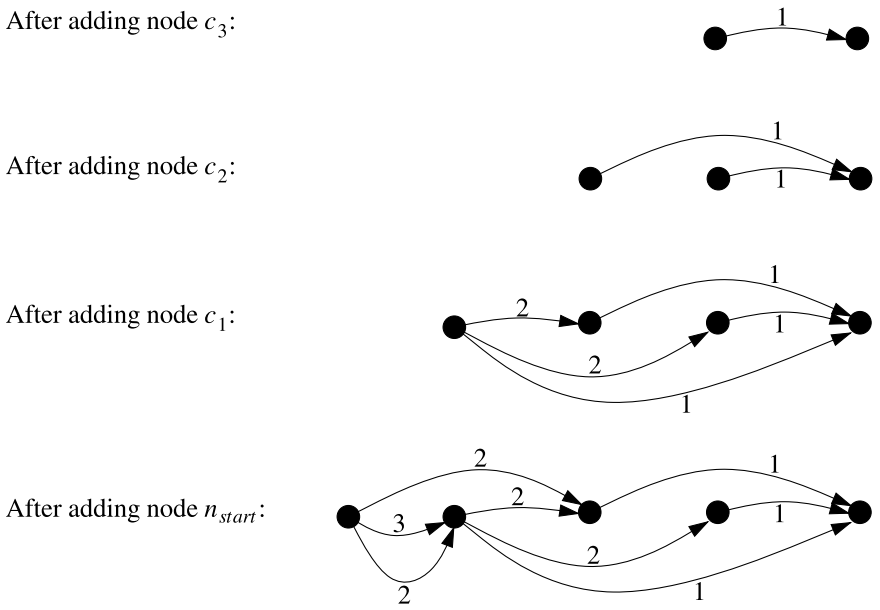

Figure 6.13: Building the path graph G^*.

randomly choose such an arc, and eventually will end up in n_{end}. The nodes from this path (except n_{start} and n_{end}) correspond with candidate positions. If we place labels at them, we produce a randomly generated, feasible labeling of n_l labels, which concludes the initialization. (Placing a variable number of labels—as discussed in Subsection 6.2.2—can be done by choosing an outgoing arc of n_{start} with arbitrary weight, and tracing a path to n_{end} in the same way as described above.)

The path graph G^* can be build in an iterative fashion, starting at n_{end} and ending with n_{start}. We will add one node and its outgoing arc at a time, using G and the part of G^* already built. After we add node c_i to G^*, we examine node c_i in G, and follow each outgoing arc. Suppose such an arc ends in node c_j. If there exists an outgoing arc from c_j in G^* with weight ℓ, we can add an arc from c_i to c_j with weight $\ell+1$ in G^*. A special case occurs when node c_j is actually node n_{end}. In that case, an arc of weight 1 is added from node c_i to node n_{end}. See Figure 6.13 for an illustration of how the graph is built. If no path is found, the algorithm exits with an error. The initialization procedure for line features is summarized as pseudo code in Algorithm 7, which takes as input a list of candidate positions $c_1 \ldots c_{n_c}$, and the required number of labels n_l.

```
1: construct G
2: derive G* from G
3: choose at random an outgoing arc with weight n_l + 1 from n_start in G*
4: if no such arc exists then
5:     exit with an error
6: else
7:     trace path to n_end, choosing arcs with appropriate weight at random
8:     for each node c_i in the path, excepting n_start and n_end do
9:         place a label in position c_i
```

Algorithm 7: Initialization of a line feature.

6.2.4 Geometrically local optimizer

Conceptually, the geometrically local optimizer will work the same as described in Section 5.2. It still uses slot filling and classifies the candidate positions depending on the surrounding features. Then, it will place the label of the feature it is applied to at the most favorable position. However, with the inclusion of line features, we need to address two issues.

Firstly, when a label of a line feature is moved, it is possible that the labeling of the line feature becomes infeasible. Therefore, care has to be taken that candidate

positions that violate the proximity constraints are not chosen.

Secondly, it has to be decided what kind of status line-feature labels have compared to the labels of SMALL, MEDIUM and MEGA cities. Which label is more important? How should conflicts be resolved?

The geometrically local optimizer thus proceeds as follows. It will try to move each label of the feature in random order. First, it classifies the slots (which correspond with candidate positions) with the most important label that intersects it. Positions that should not be used (because they make the labeling infeasible) are classified differently. Then, it tries to place the label of the feature it is applied to at a candidate position such that the labeling of the local region improves. If necessary (and the feature is a point feature), it deletes the label.

Classifying candidate label positions. The first step in the slot-filling procedure gives each slot its state, which is determined by the label of the most important feature that intersects the position. As before, we have the states EMPTY, SMALL, MEDIUM and MEGA. We add a state that signifies a line-feature label is intersecting the position (LINE), and a state that signifies the candidate position should not be used at all (INFEASIBLE). We use the latter state to disregard those positions that would make the labeling infeasible. Next we describe how we can find the slots that need to be assigned the status INFEASIBLE.

We learned in Section 6.2.3 that finding a feasible labeling is equivalent to finding a path of length $n_l + 1$ in the constraint graph G, starting at n_{start} and ending at n_{end}. Since the labeling is feasible before the geometrically local optimizer is applied to it, the labeling corresponds to a path $n_{start} \ldots c_i c_j c_k \ldots n_{end}$, where c_j is the node corresponding to the label which we want to move (which is placed at position c_j). We now have to find all nodes that connect the path $n_{start} \ldots c_i$ to the path $c_k \ldots n_{end}$, yielding a feasible labeling again. We can easily do this by inspecting each outgoing arc from c_i and checking if the node it points to is connected to c_k.

For each candidate position, we can decide whether the labeling still satisfies the proximity constraints if the label is placed there. If it does, we classify the slot corresponding to the position just like we did in Section 5.2—note however, that a slot can now also be assigned the status LINE. Otherwise, we classify the slot as INFEASIBLE, and in the next step of slot filling the candidate position will be disregarded as a possibility to place the label.

Deciding where to place the label. The next step in the slot-filling procedure uses the classification of the candidate positions to decide where to place the label. There are now two extra classes a candidate position can be in: LINE and INFEASIBLE. Handling INFEASIBLE is easy, we simply do not consider that can-

didate position at all. We deal with LINE by giving it the highest priority in the hierarchy of classes, since we consider line-feature labels to be the most important. We do not allow deletion of a line-feature label, because it would make the solution infeasible. Recall from Section 5.2 that MEGA labels were never deleted. A consequence of giving LINE a higher priority is that we need to stop this special treatment. A label of a MEGA city can be deleted when it can not be moved to a position where it does not intersect line-feature labels. This can certainly be a undesired side effect. One solution to prevent this could be to allow line-feature labels to be deleted, but enforce the proximity constraints as if the label is still placed. We have to be careful, however, that this is only done for line features with many labels. Generally speaking, when more complex cartographic rules have to be enforced, the geometrically local optimizer needs to be made more sophisticated.

The slot-filling procedure thus proceeds in the same manner as was described in Section 5.2, with the small exception of how preferences are handled. Since line features do not have preferred positions, no specified order on the positions that are tried is used when the geometrically local optimizer is applied to a line feature.

6.3 Comparisons

In the previous section, we described how the approach for labeling point features with a GA can be extended to handle line features as well. An implementation of the GA was used to explore the results of this approach.

We used real maps for the following experiments, which show portions of the Mississippi delta (see Figure 6.14) and the west coast of North America (see Figure 6.15). The maps were generated by extracting cities, rivers, country outlines and lakes from a data set that was bundled with the ESRI ArcView[24] program. This data was clipped by the bounding boxes that held the two regions of interest. The data is largely unstructured, which is mostly a problem with the river data. Two rivers (the Missouri and the Colorado) were specified by several polylines, which were manually joined. Note that each river is a single polyline, so there is no special significance for branches of a river.* When a river was divided in two parts by a lake, we joined the parts using only a single shore line of the lake—note that this can cause a label to be placed inside the lake. The lake at the start of the Tennessee was disregarded altogether for labeling.

We will compare the results of the GA with the local-search algorithm (LSA) from Section 5.2. It uses the same initializer and geometrically local optimizer

*Two very short branches of the Canadian and the San Joaquin were excluded from labeling.

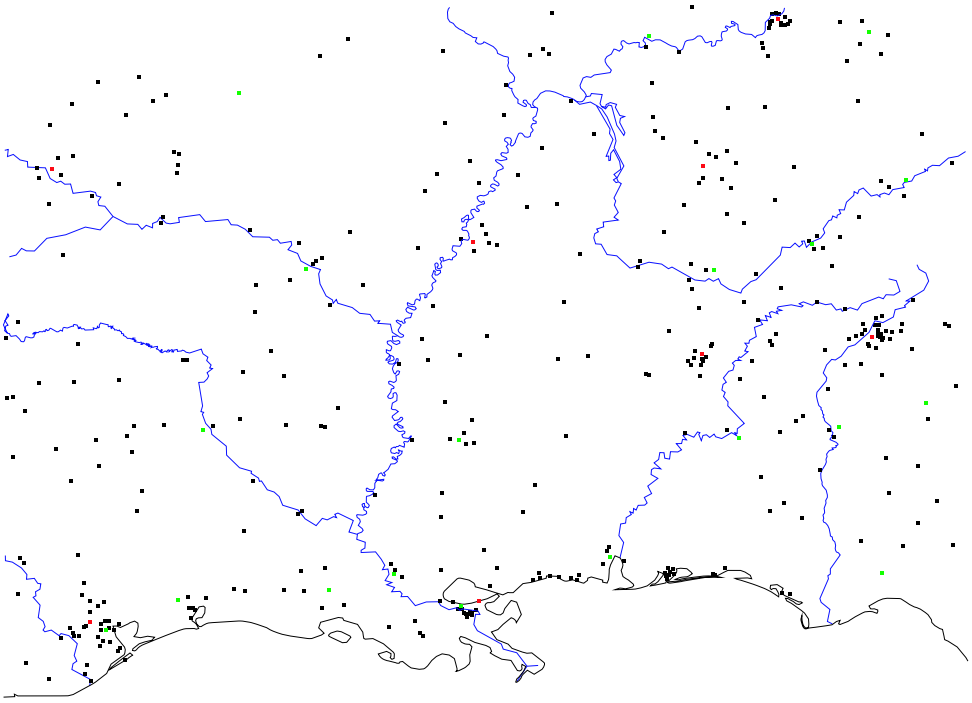

Figure 6.14: The unlabeled map of the Mississippi area.

(GLO) as the GA. After initializing a single solution, it repeatedly applies the GLO to all features on the map. The termination criterion is the same as in Section 5.2: if the last 50 iterations did not improve the number of non-intersecting labels, a new solution is generated. The total length of the run is the (maximal) amount of time the GA spent on the same map. The population size of the GA was 200. A run of the GA was stopped when the last 500 recombinations did not improve the average fitness of the population by one. The results from the GA are the average of 10 similar runs with different seeds for the random-number generator. The values for the constants of the proximity constraints were chosen to produce maps which looked good and had multiple labels. Unless otherwise specified, we used $\varepsilon_{close} = 18$ and $\varepsilon_{far} = 46$.

We are aware of only one other attempt to provide a general algorithm for point and line-feature labeling that combines the combinatorial constraint with cartographic rules. This is the approach based on simulated annealing by Edmondson et al.[23] Since we already did extensive experiments to compare the GA with simulated annealing in Chapter 3, we do not repeat them here. We feel the conclusion from that chapter still holds: both algorithms can produce high-quality solutions, but the approach described in this thesis is more robust, extendible and easier to include domain knowledge with. Other than that, it would be very diffi-

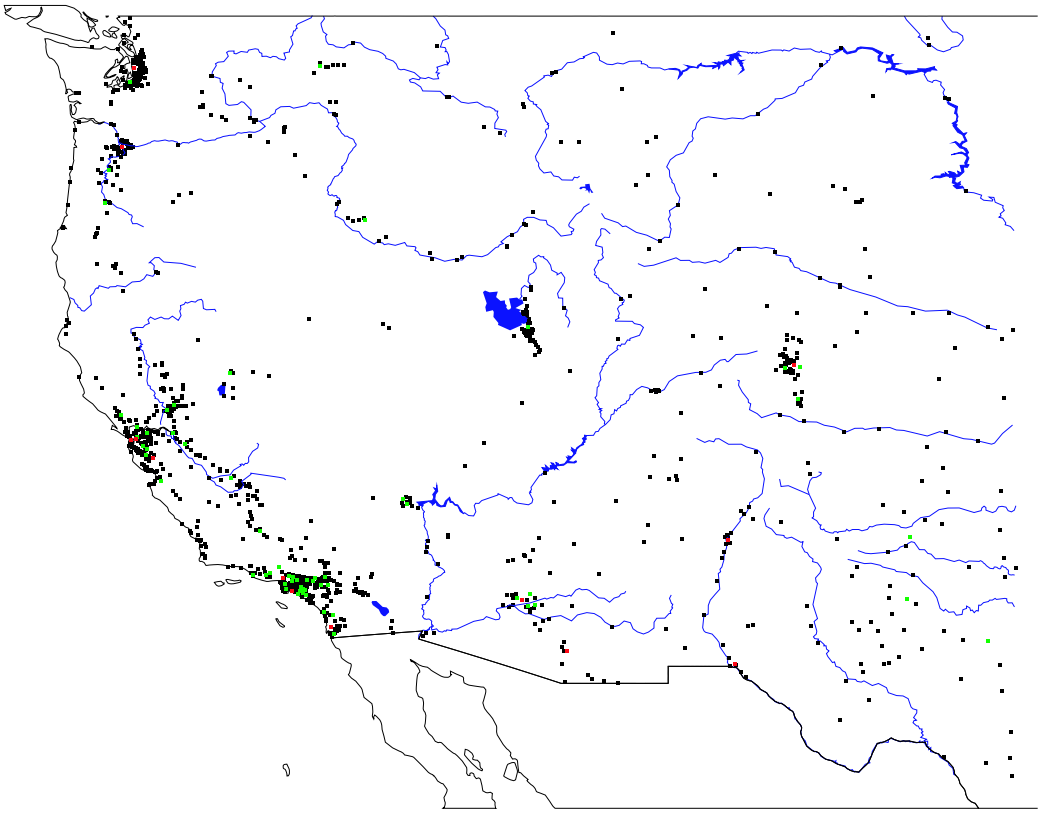

Figure 6.15: The unlabeled map of the west-coast area.

cult to compare both approaches fairly, since it is hard to quantify which result is better when constraints other than minimizing the number of conflicts are added. One of the strengths of the approach using the GA is that the ability to express cartographic rules in the GLO makes it often unnecessary to quantify quality, but this makes comparisons hard. In addition, the approach of Edmondson only places a single label for each line feature, whereas we place multiple labels that adhere to the proximity constraints.

We present the results as follows. Firstly, we will investigate how useful it is to integrate line-feature labeling with point-feature labeling, as opposed to fixing the line-feature labels beforehand and doing just point-feature labeling. Secondly, we will compare the results of the GA with the local-search algorithm.

Usefulness of placing line-feature labels with the GA

Before we accept the GA as outlined in this chapter as a viable approach to solving the map-labeling problem, we should investigate whether it is really necessary to integrate both point- and line-feature labeling in the same GA. After all, maybe it is simpler just to fix the label positions of the line features and then perform point-feature labeling. Do we really gain anything by using a more complex GA that can handle line features? To answer this question, we compared two variants of the GA on the same map. The first variant is the GA described in this chapter. The second variant is the same GA, with the following modification. During initialization, a random labeling for line-feature labels is produced. This labeling is the same for all individuals in the population. In addition, the labeling remains fixed during the run of the algorithm. That is, in the geometrically local optimizer, line-feature labels are never moved.

For both GAs, ten runs were done with different seeds for the random-number generator, and the average is shown in Figure 6.16 for the west-coast map, using a medium scale. The average of ten runs for the Mississippi map, using a medium scale, is shown in Figure 6.17. The standard deviation of the final solutions is shown in the last point of the run. It shows that, depending on the map, the results can improve considerably when the integrated approach is used.

Comparison with the local-search algorithm

Next, we compare the GA against the LSA. In Table 6.1, the results of the GA and the LSA are shown for the Mississippi map and the west-coast map for different scales. The notation $avg_{\pm sd}$ denotes the average *avg* of the runs, which standard deviation of *sd*. The labels of the rivers were never deleted. The table shows that the GA outperforms the LSA. The GLO allows the GA to be extended to incorporate line features, yielding a GA that can search efficiently and respect the

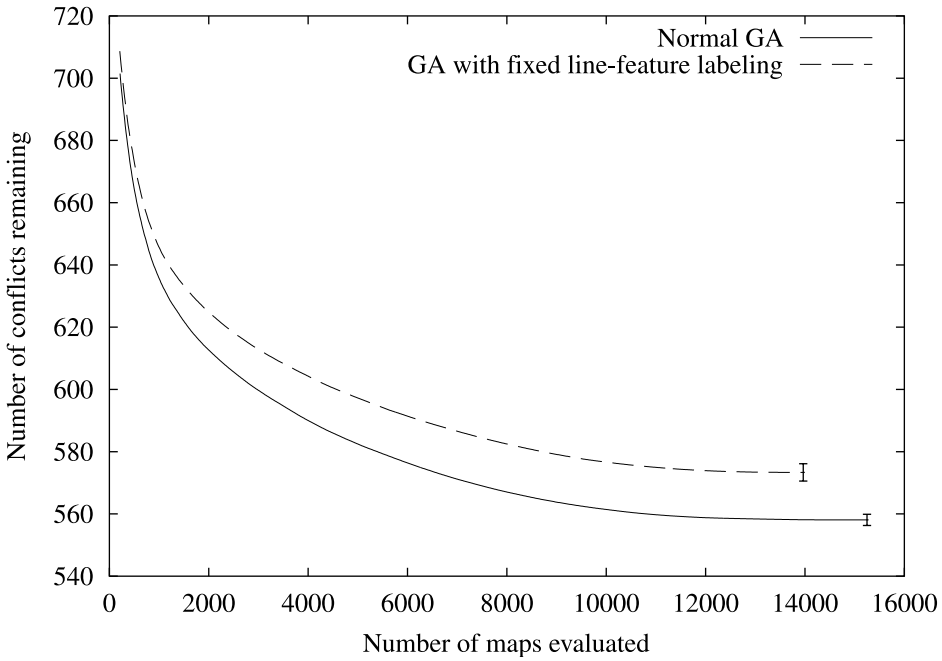

Figure 6.16: The usefulness of integrating both point- and line features in the GA. The map labeled is the west-coast area, using the medium scale.

cartographic rules as well as the combinatorial constraints. The results also show that the GA techniques described in this chapter work well for maps of different scales.

6.4 Discussion

In this chapter, we extended the point-feature GA to handle line features as well. The results showed that the GA is able to produce good solutions. We only needed to extend the initializer and the geometrically local optimizer, which illustrates the flexibility of the GA's design. However, a close inspection of the map produced by the GA reveals there is room for improvement in the geometrically local optimizer.

In Figure 6.18, a labeling of the west coast of the USA is shown, as done by the GA. Indicated in this figure are some areas in which undesirable artifacts are present (the enumeration refers to the circles on the map):

1. No provision is made for dealing with the background of the map. In the figure, the labels of Brigham City, Clinton, Clearfield and Magna are placed inside the lake, making it hard to read. This happens, because, as far as the GA is concerned, there is no lake. The GA, as described, works strictly with

	Mississippi area	SMALL placed	SMALL deleted	MEDIUM placed	MEDIUM deleted	MEGA placed	MEGA deleted	LINE placed	LINE deleted
GA	Small scale	$331.7_{\pm 1.4}$	$31.3_{\pm 1.4}$	$19.0_{\pm 0.0}$	$0.0_{\pm 0.0}$	$8.0_{\pm 0.0}$	$0.0_{\pm 0.0}$	25	0
	Medium scale	$283.9_{\pm 1.2}$	$79.1_{\pm 1.2}$	$18.9_{\pm 0.3}$	$0.1_{\pm 0.3}$	$7.9_{\pm 0.3}$	$0.1_{\pm 0.3}$	10	0
	Large scale	$233.1_{\pm 0.9}$	$129.9_{\pm 0.9}$	$18.0_{\pm 0.0}$	$1.0_{\pm 0.0}$	$8.0_{\pm 0.0}$	$0.0_{\pm 0.0}$	9	0
LSA	Small scale	323	40	19	0	8	0	25	0
	Medium scale	269	94	19	0	7	1	10	0
	Large scale	208	155	17	2	8	0	9	0

	west-coast area	SMALL placed	SMALL deleted	MEDIUM placed	MEDIUM deleted	MEGA placed	MEGA deleted	LINE placed	LINE deleted
GA	Small scale	$730.4_{\pm 3.9}$	$351.6_{\pm 3.9}$	$47.3_{\pm 1.6}$	$11.7_{\pm 1.6}$	$12.7_{\pm 0.5}$	$0.3_{\pm 0.5}$	91	0
	Medium scale	$545.5_{\pm 2.8}$	$536.5_{\pm 2.8}$	$38.0_{\pm 1.3}$	$21.0_{\pm 1.3}$	$13.0_{\pm 0.0}$	$0.0_{\pm 0.0}$	44	0
	Large scale	$447.7_{\pm 2.1}$	$634.3_{\pm 2.1}$	$31.5_{\pm 1.7}$	$27.5_{\pm 1.7}$	$12.9_{\pm 0.3}$	$0.1_{\pm 0.3}$	31	0
LSA	Small scale	680	402	50	9	12	1	91	0
	Medium scale	497	585	37	22	13	0	44	0
	Large scale	405	677	32	27	13	0	31	0

Table 6.1: Results for the map of the Mississippi area and the map of the west-coast area, using different scales. The notation $avg_{\pm sd}$ denotes the average number of labels *avg*, with standard deviation of *sd*.

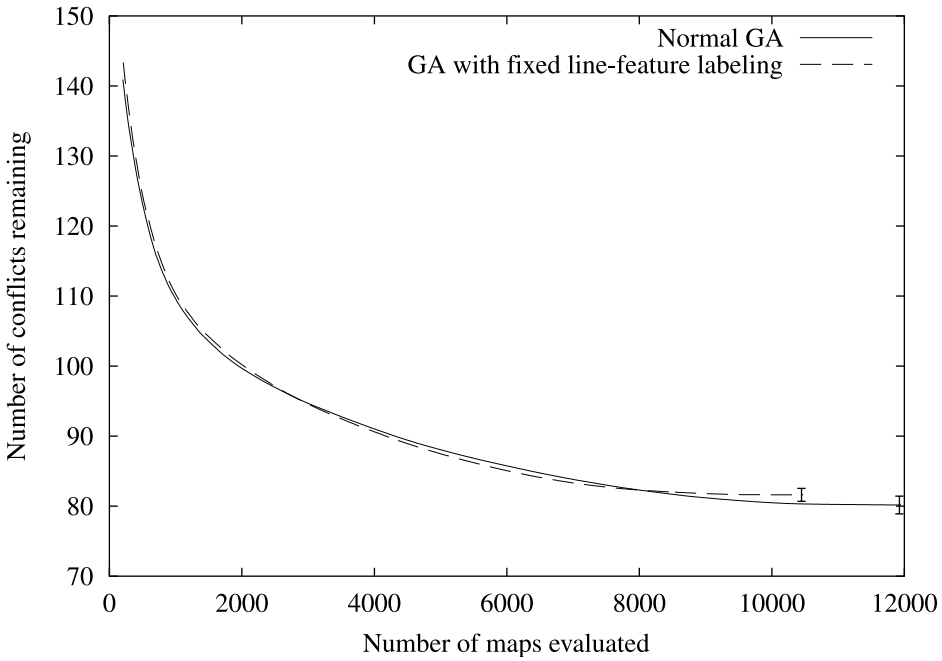

Figure 6.17: The usefulness of integrating both point- and line features in the GA. The map labeled is the Mississippi area, using the medium scale.

label positions, and never considers the geometry of the features. One solution would be to extend the geometrically local optimizer with a measure of how many objects intersect a candidate position. Such a measure can also be used to improve the quality measure which was used to calculate candidate positions for line features, to avoid generating candidates that intersect, for example, another line feature.

2. A special case of the previous artifact is seen where the label of the Missouri river is placed inside the river itself. This is a consequence of the fact that the input data contained a lake, from which the south coast was used to join the parts of the river. The construction of polylines from the geographical data was described at the start of Section 6.3.

3. Certain cartographic rules are not yet incorporated. For example, in cartography it is considered bad when a river divides a city and its label. In the figure, Anderson is placed on the wrong side of the river, even though there is enough room on the other side. Another such rule is: “labels of cities next to a large body of water should be placed entirely in the water”, which is also not used by the GA. These rules can easily be enforced in the geometrically local optimizer, if it can be determined which candidate po-

sitions violate them. To do this, it is needed that the candidate positions are assigned additional attributes (like “places label in water”) which describe their relation to the surrounding geometry.

4. The label of Oakland, which is a MEGA city, is deleted (it is placed on the map lightly shaded). This happens because it intersected in all positions either the label of San Francisco (a MEGA city) or the label of the San Joaquin (classified as a LINE). Thus, it was deleted in the slot-filling procedure. Note that the San Joaquin label could easily be placed out of the way, at the small cost of deleting the label of Bonadelle Ranchos-Madera Ranchos. We tried to address this problem in Section 5.2—a simpler version of the problem is shown in Figure 5.3 on page 110. Solving this would require that the GLO considers the geometrical context of the rivals of the point it is applied to.

5. At junctions in a river, labels should be repeated to make clear how both branches are called. For example, consider the junction in the figure. It is not immediately clear which branch is the Colorado, and what the name of the other branch is. This problem also results from the fact that the geometrically local optimizer only uses label positions. It can be solved by adding a constraint similar to the proximity constraints that ensures a label is placed near a junction. The constraint has to be respected by the initializer and the geometrically local optimizer.

6. When placing labels on a map, it can not be avoided that some labels are placed on portions of a line. In the figure, for example, the label of Aberdeen was placed with a vertical line running through it. This makes it harder to read. The ideal solution would be to paint the part of the line under the label in a color that has a lower contrast with the text. Alternatively, it can be removed completely.

Area features. Support for area-feature labeling is missing, too. It is relatively easy to extend the GA to handle area features. Really all that would be needed is a procedure for calculating suitable area-feature label positions. Note that an area-feature label is usually not a single rectangle, but a set of rectangles corresponding with the bounding boxes of the individual letters of the name. As a result, the procedure that tests for a label intersection needs to be modified slightly.

It is questionable, however, whether it is necessary or useful to build this into the GA. The improvement in the fitness of the final solution which is achieved by integrating the placement of line-feature labels in the GA—as opposed to a preprocessing step—was relatively modest. Since a map will have relatively few

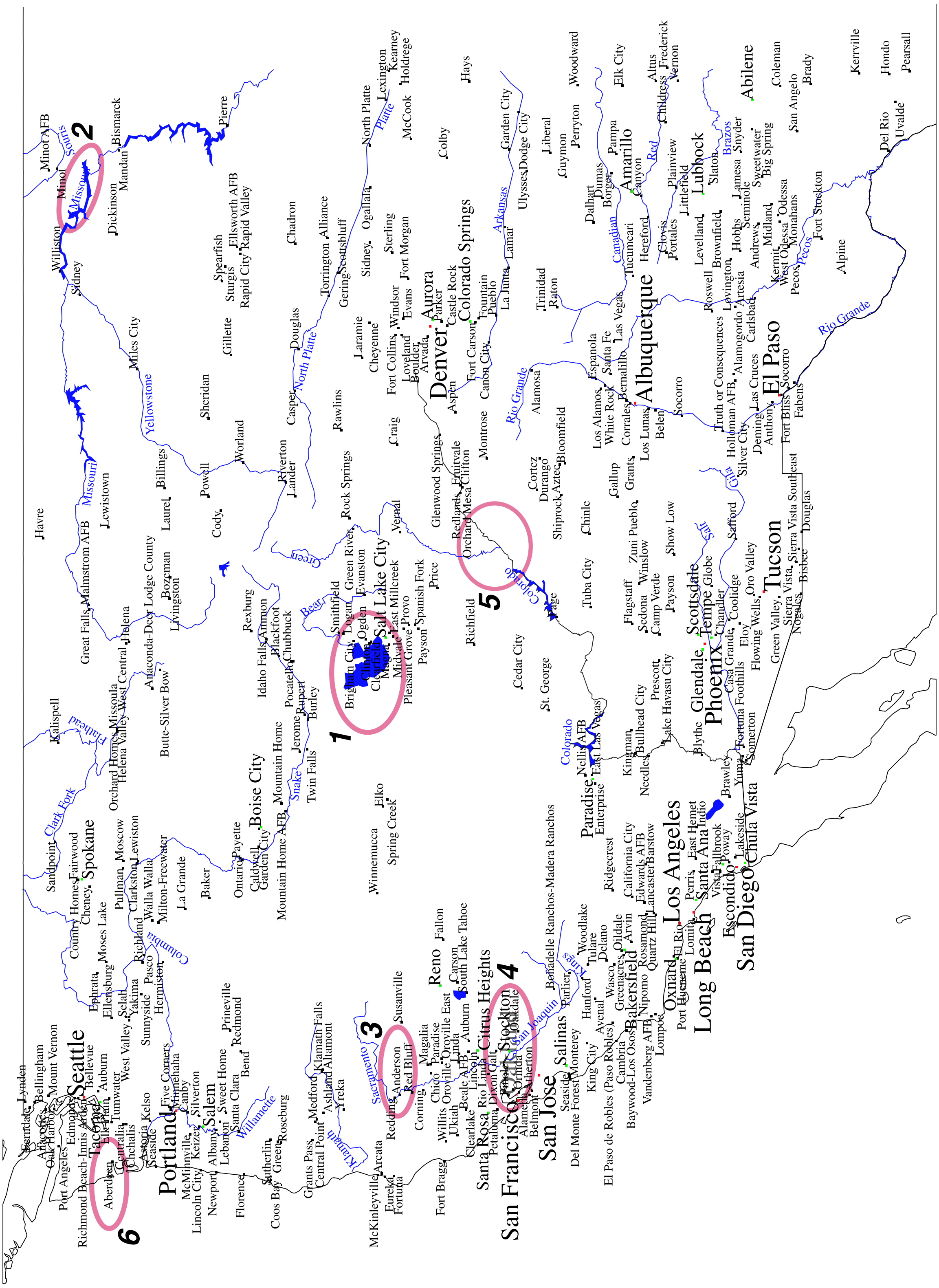

Figure 6.18: The west coast area, as labeled by the GA using $\varepsilon_{close} = 18$, $\varepsilon_{far} = 46$, $\varepsilon_{medium} = 100000$, $\varepsilon_{mega} = 370000$ and a scale that is four times larger than the small scale.

area features, we expect little gain in integrating the placement of area-feature labels in the GA.

Therefore, it is probably best to use pre- and postprocessing steps to take care of the area features. In the preprocessing step, a good labeling for area features alone can be calculated using techniques from literature.[106, 23, 75, 2] In the postprocessing step, the positions of area labels can be altered slightly to resolve remaining conflicts.

6.5 Conclusion

In this chapter, we took a very pragmatic approach to the map-labeling problem. We concentrated on discussing techniques for extending the GA to handle line features. For long line features, labels were repeated at suitable intervals, ensuring that each vertex of the line feature is close to a label. We were less concerned with details than with general solutions. For example, the way we find candidate label positions can be improved by using curved labels, a better quality measure, and so on. However, we think these improvements are relatively straightforward, at least from a conceptual point of view.

We showed the GA succeeds in producing good-looking maps. The usefulness of integrating the placement of line-feature labels into the GA was tested by comparing against the same GA that gave all individuals in the population the same labeling for line features which could not be changed. The GA using the integrated approach showed a modest improvement. Compared to the local-search algorithm, the GA can place significantly more labels. However, the resulting maps contain artifacts which require a more sophisticated geometrically local optimizer to be resolved. The design of the GA as a whole does not need to be changed, however, which demonstrates the flexibility of the approach.

CHAPTER 7

Conclusion

In this thesis, we set out to study the use of GAs to solve the map-labeling problem, and to apply theoretical insights about GAs to a real-world problem. These two goals were reflected in the intended audiences, namely those with a cartographic interest, and those interested in GAs. Next we discuss what this thesis has to offer these two audiences.

For researchers from the field of geographical information systems, we have presented a new method for automated label placement, based on genetic algorithms. We designed a GA that can find good solutions for map-labeling problems that include point and line features. The solutions are good in a combinatorial sense, that is, solutions can be found that have close to the optimal number of labels. In Chapter 3, the GA for point-feature labeling was compared with the current state-of-the-art and was found to be competitive. In Chapter 4, maps of which the optimal number of non-intersecting labels was known were labeled using the GA, and close to optimal solutions could be found using only small populations. The GA can easily be extended to find solutions which are good with respect to cartographic rules. In Section 5.2, we exemplified the addition of cartographic rules by incorporating them in a separate procedure, the geometrically local optimizer. The GA was thus extended to place point-feature labels in preferred positions, respect differences in importance of the features, and to do integrated name selection. Chapter 5 presented a framework for designing GAs for GIS problems. In addition to map-labeling, the application of the framework was demonstrated with the

line-simplification problem and a generalization problem. Furthermore, we added the handling of line features to the GA in Chapter 6. If necessary, line-feature labels were repeated, and placed at appropriate intervals along the line feature. This is one of the few algorithms that allows for multiple labels for line features. Extending the GA to handle area features should be relatively straight-forward, but we argued that we would probably gain little compared to a preprocessing step.

The map-labeling problem is hard in a combinatorial sense—in fact, even basic instances of the problem are NP-hard. In addition, it is also difficult because solutions have to respect many additional cartographic rules. The usual way that map-labeling algorithms—for example, the simulated-annealing algorithm of Edmonson[23]—enforce the cartographic rules is by expressing them in the cost function. There are, however, serious problems with that approach. The cartographic rules can degrade the combinatorial rule, it is difficult to express locally hard rules (for example, "capitals should always be labeled"), it is hard to judge how the weighting factors affect the solutions by looking at the map, the tuning of the weighting factors takes time, and it makes the algorithm inflexible. Therefore, we have opted for another approach. The division in two different classes is reflected in the design of our GA. The combinatorial constraint is expressed in the fitness function, and is handled by the normal mechanics of any GA, namely recombination and selection. The cartographic rules are handled in the geometrically local optimizer. This gives the algorithm considerable flexibility, since we can often avoid the need to quantify the cartographic rules.

In order to be practically useful, it has be possible to integrate the GA into a GIS. Therefore, we took care that the user of the GIS doesn't have to set or tune any arcane parameters. Most GA-specific parameters could be eliminated. The cost function is kept simple, to avoid weighting factors that need to be tuned. Furthermore, the maintainer of the GA is able to add constraints to the problem definition without the need to redesign the whole algorithm. This was made possible with the geometrically local optimizer.

For readers with an interest in genetic algorithms, designing a GA to solve a real-world problem may sometimes seem like a black art. The study of the map-labeling problem shows that this is not necessarily always the case. Theoretical results from literature have helped us to design a GA which solves the problem well. In Chapter 3, we showed that concepts like linkage, building blocks, mixing and disruption could be used to design a GA that outperformed GAs which ignored these issues. The insights from theory naturally lead to the addition of a new operator, the geometrically local optimizer. The clear geometrical structure of geographical problems such as the map-labeling problem allows us to identify likely building blocks of the solution. We can exploit our knowledge about the geometry to construct a geometrically local optimizer, which produces locally

optimal parts of a solution, that is, building blocks. Building blocks in the map-labeling problem are geometrically local regions of the map which have a good labeling.

By designing the GA with these theoretical insights in mind, we were able to perform an analysis of its scale-up behavior in Chapter 4. We used models from literature for the population size (the gambler's-ruin model), and the number of generations until convergence. These theoretical models have thus far only been applied to artificial problems with known properties. We have shown that our GA adheres to the assumptions underlying the models, and experiments on uniformly-dense maps confirmed the predicted scale-up—the number of function evaluations scales up linearly with respect to the input size. As far as we know, this is the first time the models have been applied to a problem of real-world significance. We were able to satisfy the conditions for these models largely because of our use of the geometrically local optimizer. It allowed us to keep the fitness function simple and additively decomposable. In addition, disruption was minimized due to the repairing effect of the geometrically local optimizer.

In Chapter 5, we showed the design of the map-labeling GA can be placed in a more general setting, yielding a framework to build GAs for a class of hard GIS problems. GAs designed in the framework are likely to adhere to the assumptions of the theoretical models, so we expect the same scale-up behavior as for the map-labeling GA.

In conclusion, we have provided a new approach for automated names placement that is based on genetic algorithms. The GA is capable of finding solutions that are good in a combinatorial sense, and adhere to cartographic rules as well. In addition, the GA is based on solid theoretical foundations and was shown to have favorable scale-up behavior. Therefore, we think we have offered a viable approach to solve map-labeling problems.

Further research can be pursued in the following directions:

- The addition of more cartographic rules to the genetic algorithm would solve more complex instances of the map-labeling problem and produce better maps.
- It would also be interesting to test our expectations regarding the scale-up behavior of the line-simplification problem and the generalization problem. In addition, the scale-up predictions of the models for maps of increasing density can be verified.
- The framework can be applied to design GAs for other GIS problems, such as the coloring of countries, political redistricting, and outline simplification of buildings.

APPENDIX A

Notation

Constants

α Expected fraction of gamblers that hit the saturation barrier.

ε Constant used in simplification: distance the simplified line can deviate from the original line.

ε_{close} Constant used in line-feature map labeling: see *closeness constraint*.

ε_{far} Constant used in line-feature map labeling: see *cover constraint*.

ε_{medium} Constant below which a city is SMALL.

ε_{mega} Constant above which a city is MEGA.

ε_n Constant used in generalization: largest distance allowed between a point and its representative.

ε_r Constant used in generalization: smallest distance allowed between two representatives.

d Signal between two competing schemata, for example the building block and its competitor.

I Selection intensity.

k Size of the partition (for example, the size of a trap function in the concatenated trap-function problem).

l Length of a chromosome.

m Number of partitions (for example, the number of trap functions in the concatenated trap-function problem).

n Size of the population.

n^* Critical size of the population.

n_c Number of candidate positions for a label.

n_{feat} Number of features on the map.

n_l Number of labels for a feature.

n_p Number of pairs for the corresponding-bits problem.

n_{points} Number of points in the polyline, like $P_{n_{points}}$.

t^* Generation when a critically-sized population is converged.

x_0 The initial number of building blocks in the population.

Variables:

$\overline{f}$ Average fitness of chromosomes in the population.

p A partition.

prop Proportion of optimal alleles in the population.

s A schema.

t The generation number.

x A solution or chromosome.

Probabilities:

Pr_c Probability of crossover.

Pr_{err} Probability of making an decision error.

Pr_m Probability of mutation.

Pr_{ok} Probability of making the correct decision.

$Pr(n)$ Probability of hitting the saturation barrier.

Functions:

$distance(\cdot)$ Distance between points or labels.

$f_{cost}(\cdot)$ Cost function.

$f_{fit}(\cdot)$ Fitness function.

$f_{trap}(\cdot)$ Trap function.

$free(\cdot)$ Number of free labels.

$length(\cdot)$ Length of a line segment.

$match(\cdot)$ Match between two bits in the corresponding-bits problem.

$pref(\cdot)$ Measure of preferred positions for labels.

$u(\cdot)$ Function of unitation (number of 1's in argument).

Acknowledgments

Writing a thesis is a long and difficult undertaking. Many people have helped and supported me, allowing me to write something I am truly proud of. I want to thank these people.

First and most of all, I want to thank my daily advisors Dirk Thierens and Mark de Berg. They are excellent researchers and have provided me with more advise than I can possibly mention here. They taught me most things I know about properly conducting science.

I thank my promoter Mark Overmars for supervising my PhD-ship, and carefully reading the whole manuscript.

I also want to thank the reading committee, namely Linda van der Gaag, David Goldberg, Joost Kok, Menno-Jan Kraak and Bernard Manderick.

Writing a GA to label maps is one thing, actually visualizing the labeling is another. Geert-Jan Giezeman wrote a tremendously helpful tool to visualize the labelings which the algorithms found. I owe him my thanks.

Linda van der Gaag read an early draft of several chapters and provided suggestions which benefited my thesis. Furthermore, she offered me a future job and a quiet room to work in. At exactly the same time, the university started with building a new library, which spoiled the calm of the room somewhat. Nevertheless, I am very grateful.

Most of the time that I spent as a PhD-student, I shared rooms with others. I want to thank Michiel Hagedoorn, Tycho Strijk, Arjan Kuijper, Marleen de Bruijne and Zeger Knops for interesting and enjoyable times.

I also thank the co-authors of my papers for the stimulating research we did together: Mark de Berg, Marc van Kreveld, Tycho Strijk, Dirk Thierens, and

Alexander Wolff. In addition, I thank Tycho and Alex for maintaining the map-labeling bibliography which was a great help in organizing my bibliography.

I want to give special thanks to my beloved Meike, who supported me always, and calmed me when I was panicking. Even while she was finishing her own studies, she made time to listen to me, talk to me, and bring me cups of coffee when I was writing.

Samenvatting

Traditioneel worden (land)kaarten met de hand door cartografen gemaakt. Deze handmatige aanpak levert kaarten op van hoge kwaliteit. Door de opkomst van computer systemen voor het analyseren en afbeelden van geografische data is er een noodzaak ontstaan voor de automatische generatie van kaarten. Zo'n geografisch informatie systeem (GIS) kan bijvoorbeeld gebruikt worden voor het nemen van een beslissing over de locatie van een toekomstige fabriek die nog gebouwd moet worden. Allerlei randvoorwaarden spelen een rol: de locatie moet bijvoorbeeld dicht bij een snelweg en water zijn, maar niet te dicht bij stedelijke bebouwing of natuurgebieden. Met het GIS kan een kaart worden gemaakt waarop alle geschikte locaties en alle relevante informatie afgebeeld zijn. Omdat deze kaart uniek is voor het specifieke probleem dat bestudeerd wordt, kan er niet gebruik worden gemaakt van kant en klare kaarten die opgeslagen liggen in het systeem; de kaart moet ter plekke gegenereerd worden.

Een belangrijk deelprobleem bij het automatisch maken van kaarten is het plaatsen van de namen op de kaart. Dit vergt dan ook een aanzienlijk deel van de tijd van een cartograaf als de kaart met de hand gemaakt wordt. De elementen van een kaart kunnen opgedeeld worden in *point features* (puntvormige elementen, zoals steden), *line features* (lijnvormige elementen, zoals rivieren) en *area features* (gebiedsvormige elementen, zoals landen). Elke *feature* kan een naam hebben die zodanig geplaatst moet worden dat de *feature* makkelijk identificeerbaar is. Er bestaan veel cartografische regels die bepalen hoe de naam bij de *feature* geplaatst moet worden. De meest belangrijke regel is dat namen elkaar niet mogen overlappen omdat ze anders onleesbaar worden. Andere regels variëren in belangrijkheid. De naam van een *point feature* zoals een stad moet bij voorkeur rechts

boven de stad geplaatst worden. Een andere regel stelt echter dat er een andere positie gekozen moet worden als de stad naast een rivier ligt en de naam en de stad aan verschillende kanten van de rivier zouden komen te liggen. Voor *line features* zoals een rivier geldt dat de namen soms herhaald moeten worden als de rivier erg lang is. De naam van een *area feature* moet dusdanig geplaatst worden dat het ongeveer de vorm van het gebied aangeeft. Zo zijn er talrijke regels die verschillen in belangrijkheid en elkaars soms tegen kunnen spreken. Een goede plaatsing voor de namen heeft geen overlappende namen op de kaart en is een zo goed mogelijk compromis tussen de overige regels.

De “geen overlap”-regel op zich maakt het plaatsen van namen al erg moeilijk. Als elke *feature* zijn naam in enkele verschillende posities kan plaatsen, dan is het totale aantal mogelijke combinaties van alle *features* tezamen erg groot. Het vinden van de combinatie die zoveel mogelijk namen zonder overlap op de kaart zet is dan al erg moeilijk. Als er een redelijk aantal *features* op de kaart staat dan is het zelfs voor de krachtigste computers ondoenlijk om gegarandeerd de beste oplossing te vinden. Daarvoor zouden namelijk alle combinaties overwogen moeten worden (bijvoorbeeld door het aantal namen zonder overlap te tellen) om de beste te vinden. Om toch een goede oplossing te kunnen vinden kan men gebruik maken van een heuristiek. Een heuristiek is een methode die niet garandeert de beste oplossing te vinden, maar die vaak toch redelijk snel een goed resultaat weet te bereiken. De heuristiek overweegt niet alle oplossingen, maar enkel degenen die veelbelovend lijken. In het proefschrift wordt als heuristiek van het genetisch algoritme gebruik gemaakt.

Het genetisch algoritme is een heuristiek om goede oplossingen voor moeilijke problemen te vinden, die gebaseerd is op de Darwiniaanse theorie van evolutie door natuurlijke selectie. In de natuur ontwikkelen dieren (en andere organismen) zich doordat hun eigenschappen (zoals vorm, of gedrag) ten dele genetisch bepaald zijn. Als deze eigenschappen invloed hebben op hun succes in de voortplanting, dan is de kans groot dat nakomelingen deze eigenschappen zullen erven. Aldus zullen de dieren evolueren en beter aangepast raken aan hun omgeving. Dit principe kan ook gebruikt worden om moeilijke problemen in het algemeen op te lossen. Het idee hierbij is om de oplossingen voor een moeilijk probleem te laten “evolueren”, waarbij de meest succesvolle oplossingen diegene zijn die het probleem het beste oplossen. Voor het probleem van het plaatsen van namen op een kaart is een oplossing een specifieke plaatsing van de namen. De mate van succes is het aantal namen dat geplaatst wordt zonder overlap met andere namen. De meest succesvolle kaarten worden geselecteerd om nieuwe kaarten te maken die er op lijken, in de hoop een betere plaatsing te vinden. Dit proces wordt herhaald tot een oplossing die goed genoeg is wordt gevonden.

In feite is het plaatsen van namen een moeilijk probleem vanwege twee verschillende redenen. Ten eerste is het aantal mogelijke manieren om de namen

te plaatsen erg groot en is het moeilijk de beste manier te vinden. Ten tweede bestaan er nog veel andere cartografische regels, die ook op de een of andere manier tot uitdrukking moeten komen. Een voor de hand liggende manier om dat te doen is om elke regel te kwantificeren, zodat het bij kan dragen aan de maat van succes voor een bepaalde plaatsing van de namen. Zo kan bijvoorbeeld elke *point feature* waarvan de naam rechts boven is geplaatst een bonus punt krijgen. De totale maat van succes voor een kaart is dan bijvoorbeeld het aantal namen zonder overlap plus het aantal bonus punten. Een van de problemen met deze methode is dat de meest belangrijke regel (het voorkomen van overlap) nu geschonden kan worden als daar genoeg bonus van andere regels tegenover staat. In het proefschrift is daarom gekozen voor een andere aanpak. De kwaliteitsmaat is simpelweg het aantal namen dat zonder overlap geplaatst kan worden, terwijl alle andere cartografische regels op een lokaal niveau toegepast worden wanneer er van geselecteerde kaarten nieuwe kaarten gemaakt worden.

De methode die beschreven is in dit proefschrift is dusdanig ontwikkeld dat hij makkelijk toegepast kan worden in een GIS. Zo is bijvoorbeeld vermeden dat er veel parameters die specifiek zijn voor het genetisch algoritme ingesteld moeten worden. Ook is het makkelijk om extra cartografische regels toe te passen indien dit wenselijk is, zonder het algoritme ingrijpend te moeten wijzigen of afstellen.

In het eerste hoofdstuk wordt een inleiding gegeven in de problematiek van het plaatsen van namen op kaarten, gevolgd door een hoofdstuk dat een algemene uitleg over de werking van genetische algoritmen geeft.

In hoofdstuk 3 wordt het genetisch algoritme beschreven dat namen kan plaatsen op kaarten met *point features.* Alleen het minimaliseren van het aantal namen dat overlapt wordt beschouwd. Dit hoofdstuk beschrijft de kern van de methode, die zorgvuldig ontworpen is om het praktisch en uitbreidbaar te maken. De volgende hoofdstukken borduren hier op voort, maar in verschillende richtingen.

In hoofdstuk 4 wordt onderzocht hoe efficiënt het genetisch algoritme is. Om praktisch bruikbaar te zijn, moet de methode goed opschalen—dat wil zeggen, bij grotere kaarten moet de rekentijd niet explosief toenemen. Gegeven een gewenste kwaliteit (bijvoorbeeld 97% van alle namen moet zonder overlap geplaatst worden) kan berekend worden hoeveel tijd het algoritme nodig heeft om een oplossing met die kwaliteit te vinden. Er blijkt dat voor de soort kaarten die onderzocht zijn, de hoeveelheid rekentijd kwadratisch is in het aantal *features* op de kaart. Dit betekent dat een twee keer zo grote kaart vier keer zo veel rekentijd vergt. Een methode die gegarandeerd de beste oplossing vindt (en dus geen heuristiek is), heeft rekentijd die exponentieel is in het aantal *features*. Het genetisch algoritme is dus veel efficiënter.

In hoofdstuk 5 worden de technieken van hoofdstuk 3 veralgemeniseerd en wordt een generiek raamwerk voor het oplossen van een klasse van GIS proble-

men gepresenteerd. Het raamwerk wordt geïllustreerd aan de hand van drie *case studies*. Allereerst wordt het plaatsen van namen verder bekeken, en worden enkele cartografische regels toegevoegd. Volgende problemen betreffen het simplificeren van lijnen zonder de vorm te verliezen, en het representeren van groepen objecten door een minimaal aantal representanten.

In hoofdstuk 6 wordt er gekeken hoe het plaatsen van namen op kaarten met *point* alsmede *line features* gedaan kan worden. Een aantal complicaties moet hiervoor opgelost worden. Zo kan het bij *line features* voorkomen dat er meerdere namen geplaatst moeten worden, die niet te dicht op elkaar, maar ook niet te ver van elkaar moeten staan.

Het laatste hoofdstuk vat het hele proefschrift nog eens samen.

Curriculum Vitae

Personalia

Naam:	Steven van Dijk
Geboortedatum:	14 april 1974
Nationaliteit:	Nederlandse

Opleiding

1986-1992:	VWO aan het Christelijk Lyceum te Gouda
1992-1997:	Studie Informatica aan de Universiteit Utrecht

Werkervaring

1997-2001:	Onderzoeker in Opleiding aan het Informatica Instituut van de Universiteit Utrecht (promotiedatum: 26 november 2001)

Bibliography

[1] P. K. Agarwal, M. van Kreveld, and S. Suri. Label placement by maximum independent set in rectangles. In *Proceedings of the Ninth Canadian Conference on Computational Geometry*, pages 233–238, 1997.

[2] J. Ahn and H. Freeman. A program for automatic name placement. In *Proceedings of the Sixth Auto-Carto Conference*, pages 444–453. ACSM/ASPRS, 1983.

[3] G. Alinhac. *Cartographie Théorique et Technique*, chapter IV. Institut Géographique National, Paris, 1962.

[4] H. Asoh and H. Mühlenbein. On the mean convergence time of evolutionary algorithms without selection and mutation. In Y. Davidor, H. P. Schwefel, and R. Männer, editors, *Lecture Notes in Computer Science, Volume 866: Proceedings of the Parallel Problem Solving from Nature III Conference*, pages 98–107. Springer-Verlag, 1994.

[5] T. Bäck. Optimal mutation rates in genetic search. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms and their Applications*, pages 2–8. Morgan-Kaufmann, 1993.

[6] T. Bäck. Generalized convergence models for tournament- and (μ,λ)-selection. In L. J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms and their Applications*, pages 2–8. Morgan-Kaufman, 1995.

[7] T. Bäck, F. Hoffmeister, and H.-P. Schwefel. A survey of evolution strategies. In R. Belew and L. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 2–9. Morgan-Kaufman, 1991.

[8] M. Barrault and F. Lecordix. An automated system for linear feature name placement which complies with cartographic quality criteria. In *Proceedings of the 12th Auto-Carto Conference*, pages 321–330. ACSM/ASPRS, 1995.

[9] T. Blickle and L. Thiele. A mathematical analysis of tournament selection. In L. J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms and their Applications*, pages 9–16. Morgan-Kaufman, 1995.

[10] A. Brindle. *Genetic Algorithms for Function Optimization*. PhD thesis, Edmonton: University of Alberta, Department of Computer Science, 1981.

[11] M. G. Bulmer. *The Mathematical Theory of Quantitative Genetics*. Clarendon Press, Oxford, 1980.

[12] E. Cantú-Paz. *Designing Efficient and Accurate Parallel Genetic Algorithms*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[13] J. Christensen, J. Marks, and S. Shieber. Algorithms for cartographic label placement. In *Proceedings of the American Congress on Surveying and Mapping 1*, pages 75–89, 1993.

[14] 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.

[15] R. G. Cromley. An LP relaxation procedure for annotating point features using interactive graphics. In *Proceedings of the Seventh Auto-Carto Conference*, pages 127–132. ACSM/ASPRS, 1985.

[16] K. A. De Jong. *Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, The University of Michigan, 1975.

[17] K. Deb and D. E. Goldberg. Sufficient conditions for deceptive and easy binary functions. *Annals of Mathematics and Artificial Intelligence*, 10(4), 1994.

[18] Y. Djouadi. Cartage: A cartographic layout system based on genetic algorithms. In *Proceedings of the Fifth European Conference and Exhibition on Geographical Information Systems*, pages 48–56, 1994.

[19] S. Doddi, M. V. Marathe, A. Mirzaian, B. M. Moret, and B. Zhu. Map labeling and its generalizations. In *Proceedings of the Eight ACM-SIAM Symposium on Discrete Algorithms*, pages 148–157, 1997.

[20] J. S. Doerschler and H. Freeman. A rule-based system for dense-map name placement. *Communications of the ACM*, 35:68–79, 1992.

[21] 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.

[22] R. Duda and P. Hart. *Pattern Recognition and Scene Analysis*. Wiley, 1973.

[23] S. Edmondson, J. Christensen, J. Marks, and S. Shieber. A general cartographic labeling algorithm. *Cartographica*, 33(4):13–23, 1997.

[24] ESRI. ArcGIS—ArcView 8.1. `http://www.esri.com/software/arcgis/arcview`, 2001.

[25] W. Feller. *An Introduction to Probability Theory and its Applications (2nd edition)*. Wiley, 1966.

[26] D. B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press, 1995.

[27] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. Wiley, 1966.

[28] M. Formann and F. Wagner. A packing problem with applications to lettering of maps. In *Proceedings of the Seventh Annual ACM Symposium on Computational Geometry*, pages 281–288, 1991.

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

[30] D. E. Goldberg. Genetic algorithms and Walsh functions: Part I, a gentle introduction. *Complex Systems*, 3(2):129–152, 1989.

[31] D. E. Goldberg. Genetic algorithms and Walsh functions: Part II, deception and its analysis. *Complex Systems*, 3(2):153–171, 1989.

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

[33] D. E. Goldberg. Construction of high-order deceptive functions using low-order Walsh coefficients. *Annals of Mathematics and Artificial Intelligence 5*, 1992.

[34] D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In G. Rawlins, editor, *Proceedings of the First Foundations of Genetic Algorithms Conference*, pages 69–93. Morgan-Kaufman, 1991.

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

[36] D. E. Goldberg, K. Deb, and D. Thierens. Toward a better understanding of mixing in genetic algorithms. *Journal of the Society of Instrument and Control*, 32:10–16, 1993.

[37] D. E. Goldberg and P. Segrest. Finite Markov chain analysis of genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms and their Applications*, pages 1–8. Morgan-Kaufman, 1987.

[38] G. Harik. *Learning Linkage to Efficiently Solve Problems of Bounded Difficulty Using Genetic Algorithms*. PhD thesis, University of Michigan, Ann Arbor, 1997.

[39] G. Harik. Linkage learning via probabilistic modeling in the ECGA. Technical report, University of Illinois, 1999.

[40] 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.

[41] G. Harik and D. E. Goldberg. Learning linkage. In R. K. Belew and M. D. Vose, editors, *Proceedings of the Fourth Foundations of Genetic Algorithms Conference*. Morgan-Kaufman, 1997.

[42] 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, *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 258–265. Morgan-Kaufmann, 1999.

[43] S. A. Hirsch. An algorithm for automatic name placement around point data. *The American Cartographer*, 9(1):5–17, 1982.

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

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

[46] E. Imhof. Positioning names on maps. *The American Cartographer*, 2(2):128–144, 1975.

[47] C. Iturriaga. *Map Labeling Problems*. PhD thesis, University of Waterloo, 1999.

[48] C. Iturriaga and A. Lubiw. Elastic labels: The two-axis case. In G. DiBattista, editor, *Lecture Notes in Computer Science, Volume 1353: Proceedings of the Symposium on Graph Drawing*, pages 181–192. Springer-Verlag, 1997.

[49] C. Iturriaga and A. Lubiw. NP-hardness of some map labeling problems. Technical Report CS-97-18, University of Waterloo, Canada, 1997.

[50] C. B. Jones and A. C. Cook. Rule-based name placement with Prolog. In *Proceedings of the Ninth Auto-Carto Conference*, pages 231–240. ACSM/ASPRS, 1989.

[51] K. G. Kakoulis and I. G. Tollis. On the edge label placement problem. In S. North, editor, *Lecture Notes in Computer Science, Volume 1190: Proceedings of the Symposium on Graph Drawing*, pages 241–256. Springer-Verlag, 1996.

[52] K. G. Kakoulis and I. G. Tollis. An algorithm for labeling edges of hierarchical drawings. In G. DiBattista, editor, *Lecture Notes in Computer Science, Volume 1353: Proceedings of the Symposium on Graph Drawing*, pages 169–180. Springer-Verlag, 1997.

[53] K. G. Kakoulis and I. G. Tollis. On the multiple label placement problem. In *Proceedings of the 10th Canadian Conference on Computational Geometry*, pages 66–67, 1998.

[54] K. G. Kakoulis and I. G. Tollis. A unified approach to labeling graphical features. In *Proceedings of the 14th Annual ACM Symposium on Computational Geometry*, pages 347–356, 1998.

[55] H. Kargupta. *SEARCH, Polynomial Complexity, And The Fast Messy Genetic Algorithm*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[56] H. Kargupta. The gene expression messy genetic algorithm. In *Proceedings of the IEEE International Conference on Evolutionary Computation*, 1996.

[57] H. Kargupta and K. Sarkar. Function induction, gene expression, and evolutionary representation construction. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 313–320. Morgan-Kaufmann, 1999.

[58] T. Kato and H. Imai. The NP-completeness of the character placement problem of 2 or 3 degrees of freedom. In *Record of Joint Conference of Electrical and Electronic Engineers in Kyushu*, page 1138, 1988. In Japanese.

[59] R. L. Keeney and H. Raiffa. *Decisions with Multiple Objectives*. Cambridge University Press, 1993.

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

[61] G. E. Langran and T. K. Poiker. Integration of name selection and name placement. In *Proceedings of the Eight Auto-Carto Conference*, pages 50–64. ACSM/ASPRS, 1986.

[62] F. Lobo. *The Parameter-less Genetic Algorithm: Rational and Automated Parameter Selection for Simplified Genetic Algorithm Operation*. PhD thesis, University of Lisbon, 2000.

[63] F. G. Lobo, D. E. Goldberg, and M. Pelikan. Time complexity of genetic algorithms on exponentially scaled problems. In D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H.-G. Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 151–158. Morgan-Kaufmann, 2000.

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

[65] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, 1992.

[66] B. L. Miller and D. E. Goldberg. Genetic algorithms, selection schemes, and the varying effects of noise. *Evolutionary Computation*, 4(2), 1996.

[67] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[68] M. Mitchell, S. Forrest, and J. H. Holland. The royal road for genetic algorithms: Fitness landscapes and GA performance. In *Proceedings of the First European Conference on Artificial Life*. MIT Press, 1992.

[69] J. Moré. The Levenberg-Marquardt algorithm: Implementation and theory. In G. Watson, editor, *Lecture Notes in Mathematics, Volume 630: Numerical Analysis*. Springer-Verlag, l978.

[70] H. Mühlenbein. How genetic algorithms really work: I. mutation and hill-climbing. In R. Männer and B. Manderick, editors, *Proceedings of the Parallel Problem Solving from Nature II Conference*, pages 15–25. Elsevier, 1992.

[71] H. Mühlenbein and D. Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm: Continuous parameter optimization. *Evolutionary Computation*, 1(1):25–49, 1993.

[72] M. Munetomo and D. E. Goldberg. Identifying linkage groups by nonlinearity/non-monotonicity detection. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 433–440. Morgan-Kaufmann, 1999.

[73] M. Pelikan and F. Lobo. Parameter-less genetic algorithm: A worst-case time and space complexity analysis. Technical report, University of Illinois, March 1999.

[74] I. Petzold, L. Plümer, and M. Heber. Label placement for dynamically generated screen maps. In *Proceedings of the 19th International Cartographic Conference*, pages 893–903, 1999.

[75] I. Pinto and H. Freeman. The feedback approach to cartographic areal text placement. In P. Perner, P. Wang, and A. Rosenfeld, editors, *Advances in Structural and Syntactical Pattern Recognition*, pages 341–350. Springer-Verlag, 1996.

[76] M. Preuß. Solving map labeling problems by means of evolution strategies. Master's thesis, Fachbereich Informatik, Universität Dortmund, Feb. 1998.

[77] L. Pun-Cheng and G. Y. Shea. Automatic bilingual name placement of 1:1000 map sheets of Hong Kong. In *Proceedings of the 19th International Cartographic Conference*, pages 925–930, 1999.

[78] Z. Qin, A. Wolff, Y. Xu, and B. Zhu. New algorithms for two-label point labeling. In *Lecture Notes in Computer Science, Volume 1879: Proceedings of the Eight Annual European Symposium on Algorithms*, pages 368–379. Springer-Verlag, 2000.

[79] G. Raidl. A genetic algorithm for labeling point features. In *Proceedings of the International Conference on Imaging Science, Systems, and Technology*, pages 189–196, 1998.

[80] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzbog, Stuttgart, 1973.

[81] F. Rothlauf and D. Goldberg. Pruefernumbers and genetic algorithms: A lesson how the low locality of an encoding can harm the performance of GAs. In M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. Merelo, and H.-P. Schwefel, editors, *Lecture Notes in Computer Science, Volume 1917: Proceedings of the Parallel Problem Solving from Nature VI Conference*, pages 395–404. Springer-Verlag, 2000.

[82] 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, Volume 1498: Proceedings of the Parallel Problem Solving from Nature V Conference*, pages 702–711. Springer-Verlag, 1998.

[83] J. D. Schaffer, R. A. Caruana, L. J. Eshelman, and R. Das. A study of control parameters affecting online performance of genetic algorithms for function optimization. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms and their Applications*, pages 51–60. Morgan-Kaufman, 1989.

[84] H. P. Schwefel. *Evolutionsstrategie und numerische Optimierung*. PhD thesis, Technical University of Berlin, 1975.

[85] R. E. Smith and E. Smuda. Adaptively resizing populations: Algorithm, analysis, and first results. *Complex Systems*, 9(1):47–72, 1995.

[86] W. M. Spears and K. A. De Jong. An analysis of multi-point crossover. In G. Rawlins, editor, *Proceedings of the First Foundations of Genetic Algorithms Conference*, pages 301–315. Morgan-Kaufman, 1991.

[87] T. Strijk. *Geometric Algorithms for Cartographic Label Placement*. PhD thesis, Utrecht University, Department of Computer Science, 2001.

[88] T. Strijk and M. van Kreveld. Practical extensions of point labeling in the slider model. In *Proceedings of the Seventh ACM Symposium on Advances in Geographic Information Systems*, pages 47–52, 1999.

[89] T. Strijk, B. Verweij, and K. Aardal. Algorithms for maximum independent set applied to map labelling. Technical Report UU-CS-2000-22, Department of Computer Science, Utrecht University, 2000.

[90] T. Strijk and A. Wolff. Labeling points with circles. *International Journal of Computational Geometry and Applications*, 11(2):181–195, 2001.

[91] G. Syswerda. Uniform crossover in genetic algorithms. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms and their Applications*, pages 2–9. Morgan-Kaufman, 1989.

[92] D. Thierens. Selection schemes, elitist recombination, and selection intensity. In T. Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms and their Applications*. Morgan-Kaufmann, 1997.

[93] D. Thierens. Estimating the significant non-linearities in the genome problem-coding. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 643–648. Morgan-Kaufmann, 1999.

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

[95] D. Thierens and D. E. Goldberg. Convergence models of genetic algorithm selection schemes. In Y. Davidor, H. P. Schwefel, and R. Männer, editors, *Lecture Notes in Computer Science, Volume 866: Proceedings of the Parallel Problem Solving from Nature III Conference*, pages 119–129. Springer-Verlag, 1994.

[96] D. Thierens and D. E. Goldberg. Elitist recombination: An integrated selection recombination GA. In *Proceedings of the IEEE International Conference on Evolutionary Computation*, pages 508–512. IEEE Press, 1994.

[97] D. Thierens, D. E. Goldberg, and A. G. Pereira. Domino convergence, drift, and the temporal-salience structure of problems. In *Proceedings of the IEEE World Congress on Computational Intelligence*, pages 535–540. IEEE Press, 1998.

[98] 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.

[99] 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, *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 188–195. Morgan-Kaufmann, 1999.

[100] S. van Dijk, D. Thierens, and M. de Berg. Scalability and efficiency of genetic algorithms for geometrical applications. In M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. Merelo, and H.-P. Schwefel, editors, *Lecture Notes in Computer Science, Volume 1917: Proceedings of the Parallel Problem Solving from Nature VI Conference*, pages 683—692. Springer-Verlag, 2000.

[101] S. van Dijk, D. Thierens, and M. de Berg. Using genetic algorithms for solving hard problems in GIS. Technical Report TR-2000-32, Utrecht University, 2000.

[102] S. van Dijk, D. Thierens, and M. de Berg. Designing genetic algorithms to solve GIS-problems. In R. Krzanowski and J. Raper, editors, *Spatial Evolutionary Modeling*. Oxford University Press, 2001.

[103] S. van Dijk, M. van Kreveld, T. Strijk, and A. Wolff. Towards an evaluation of quality for label placement methods. In *Proceedings of the 19th International Cartographic Conference*, pages 905–913. International Cartographic Association, 1999.

[104] M. van Kreveld, T. Strijk, and A. Wolff. Point labeling with sliding labels. *Computational Geometry: Theory and Applications*, 13:21–47, 1999.

[105] M. van Kreveld, R. van Oostrum, and J. Snoeyink. Efficient settlement selection for interactive display. In *Proceedings of the 13th Auto-Carto Conference*, pages 287–296. ACSM/ASPRS, 1997.

[106] J. W. van Roessel. An algorithm for locating candidate labeling boxes within a polygon. *The American Cartographer*, 16(3):201–209, 1989.

[107] O. Verner, R. Wainwright, and D. Schoenefeld. Placing text labels on maps and diagrams using genetic algorithms with masking. *INFORMS Journal on Computing*, 9(3):266–275, 1997.

[108] B. Verweij. *Selected Applications of Integer Programming: A Computational Case Study*. PhD thesis, Utrecht University, Department of Computer Science, 2000.

[109] B. Verweij and K. Aardal. An optimisation algorithm for maximum independent set with applications in map labelling. In *Lecture Notes in Computer Science, Volume 1643: Proceedings of the Seventh Annual European Symposium on Algorithms*, pages 426–437. Springer-Verlag, 1999.

[110] F. Wagner and A. Wolff. An efficient and effective approximation algorithm for the map labeling problem. In *Lecture Notes in Computer Science, Volume 979: Proceedings of the Third Annual European Symposium on Algorithms*, pages 420–433. Springer-Verlag, 1995.

[111] F. Wagner and A. Wolff. A combinatorial framework for map labeling. In S. H. Whitesides, editor, *Lecture Notes in Computer Science, Volume 1547: Proceedings of the Symposium on Graph Drawing*, pages 316–331. Springer-Verlag, 1998.

[112] D. Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms and their Applications*, pages 116–121. Morgan-Kaufman, 1989.

[113] A. Wolff, L. Knipping, M. van Kreveld, T. Strijk, and P. K. Agarwal. A simple and efficient algorithm for high-quality line labeling. In P. M. Atkinson and D. J. Martin, editors, *Innovations in GIS VII: GeoComputation*, chapter 11, pages 147–159. Taylor & Francis, 2000.

[114] M. Yamamoto, G. Camâara, and L. A. N. Lorena. Tabu search heuristic for point-feature cartographic label placement. *GeoInformatica*, 2001. To appear.

[115] M. Yamamoto, L. A. N. Lorena, and G. Camâara. Tabu search application for point features cartographic label placement problems. In *Proceedings of the Third Metaheuristics International Conference*, 1999.

[116] P. Yoeli. The logic of automated map lettering. *The Cartographic Journal*, 9:99–108, 1972.

[117] S. Zoraster. Integer programming applied to the map label placement problem. *Cartographica*, 23(3):16–27, 1986.

[118] S. Zoraster. The solution of large 0-1 integer programming problems encountered in automated cartography. *Operations Research*, 38(5):752–759, 1990.

[119] S. Zoraster. Practical results using simulated annealing for point feature label placement. *Cartography and GIS*, 24(4):228–238, 1997.

Index

adaptive population sizing, 21
additively decomposable function, 23
adequate mixing, 33
allele, 17
alphabet, 21
anchor, 137, 141
annealing, 65
area features, 1
artificial evolution, 17
automated label placement, 2

basic map-labeling problem, 35
bounded difficulty, 80
building block, 22, 42, 81
building block hypothesis, 22, 32

cartographic rules, 6
central point, 42
chromosome, 17, 29
closeness constraint, 138
collateral noise, 81
concatenated trap function, 79
connected components, 63
constraint graph, 149
convergence, 18
corresponding-bits problem, 17
cost function, 2, 17
cover constraint, 138
critical population size, 75
crossover, 27
crossover mask, 51
crossover probability, 28, 33

deceptive function, 79
decision error, 81
deviation from the original line, 119
disruption, 28, 43
distance factor, 46
Douglas and Peucker, 121

eight-position placement model, 36
elastic label, 5
elitism, 25
elitist recombination scheme, 25, 57
encoding, 21
evolutionary algorithm, 15
exploitation, 27
exploration, 27
extended polyline, 141

feasible, 17
features, 1
fitness, 16

fitness function, 17, 23
fitness landscape, 22
fitness proportionate selection, 24
four-position placement model, 35

gene, 17
generalization while preserving structure, 125
genetic algorithm, 15
genetic drift, 27
geographical information system, 2
geometrically local optimizer, 54
global optimum, 22
good labeling, 46

hitchhiking, 27

Imai and Iri, 121
initial baseline, 141
initializer, 22

label, 1
label placement, 1
labeling, 35
line features, 1
line simplification, 117
linkage, 42, 54
linkage respecting crossover, 32
local optimizer, 47
local optimum, 22

mapping, 137
mask, 46
masking, 46
mating pool, 18
mixing, 28, 33, 43
mutation, 29
mutation probability, 29

name selection, 5
natural evolution, 17
natural selection, 16
needle-in-a-haystack problem, 31
neighbor, 46, 127

one-point crossover, 27
optimization problem, 2

participant, 119
partition, 30
path graph, 150
penalty function, 23
perfect mixing, 77
perfect mixing of building blocks, 85
point features, 1
population size, 20
problem variable, 17
proximity constraints, 136

ranking, 25
recombination, 27
representatives, 125
restart, 116
rival, 42
rival crossover, 51, 52
rival graph, 63
rival group, 42
robust, 73

scale, 6
scale-up behavior, 75
schema, 30
schema theorem, 32
selection, 24
selection pressure, 24
semi-separable, 80
separable ADF, 23
setting, 17
settlement selection, 5
shifted baseline, 142
signal, 82
simplified point, 120
simulated annealing, 65
sliding label, 5
solution, 17

span, 141
spike, 121

tournament selection, 25
trap function, 79
tuning, 49
two-point crossover, 27

uniform crossover, 27
uniformly scaled ADF, 23

Walsh coefficients, 31
Walsh decomposition, 30
weighting factor, 10