Robust Genetic Algorithms for High Quality Map Labeling

Steven van Dijk Dirk Thierens Mark de Berg *

December 9, 1998

Abstract

The problem of placing labels on maps has been around for about twenty years and has proven to be a difficult one. A variety of methods has been proposed to generate good labelings, with a wide range of results. This paper will propose a stochastic approach using Genetic Algorithms to solve the problem of placing labels for point features. This method generates high quality labelings and is robust in the sense that it is built to be extendible to other problem instances (involving point features) without dramatic changes of the algorithm or much loss of quality.

*Department of Computer Science, Utrecht University, P.O. Box 80089, 3508 TB Utrecht, The Netherlands. Email: {steven, dirk, markdb}@cs.uu.nl

Contents

List of Figures

List of Tables

1 Introduction

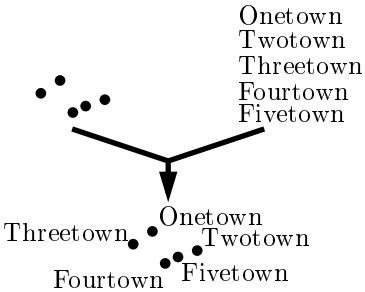

Figure 1: The map labeling problem.

Map labeling is the problem of determining a (near)optimal placing of labels (such as names of cities, rivers etc.) on a given input map (see figure 1). The input map can consist of point features such as cities or measure points, line features such as rivers or borders, and area features such as countries or forests. Each feature has an associated name (which is usually called a label), which can be placed in an infinite number of ways on the map. A good labeling has to adhere to a number of criteria which ensure the map is readable, aesthetically pleasing and gives easy access to the information it is offering [11]. Cartographers spend a lot of time on labeling maps to satisfy these criteria. However, since geographical information systems are becoming more widely used every day, a need has arisen to automate this process in such a way that high quality maps can be generated with a minimum of user control. After all, users will want to be able to compose their own maps and it is not feasible to ask the help of a professional cartographer for such maps.

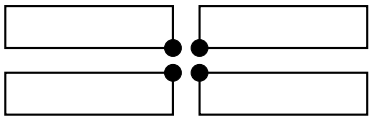

Figure 2: Possible positions for the label of a point feature.

In this paper we make several simplifications of the problem described above. First of all, we concentrate on point features. Second, each label is considered to be an axis parallel rectangle that can be placed in one of four or eight positions around the point feature (see figures 2 and 3). We will use several placement models. If the allowed positions are as shown in figure 2, we are using the four-position model. If the positions from figure 3 are also allowed, we are using the eight-position model. Finally, we will try to maximise the number of labels without overlap. We will for the moment largely ignore the other criteria that are needed for making a good map. It should however be noted that in this paper we will build a general framework in which other criteria can be incorporated easily. We will have more to say about this later (in section 6).

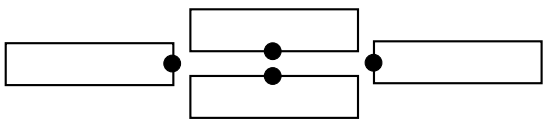

Figure 3: Additional positions for the label of a point feature.

The problem as it stands (minimise the number of overlapping rectangles considering fixed possible positions) is difficult enough. In fact, it is NP-complete [14], which means that it is not possible to construct an efficient (that is, polynomial time) algorithm which solves the problem unless P=NP which is generally not believed to be the case. We are therefore at the mercy of heuristics to try finding good results in reasonable time. Recently, a comparative study has been done for map labeling algorithms by Christensen et al. [2]. They described map labeling algorithms that have appeared in the literature and then compared them against each other and a stochastic algorithm (based on simulated annealing) of their own devise. They described a framework with a way of generating maps on which the algorithms were tested. Their conclusion suggests that stochastic algorithms are the most promising approach for producing high quality maps, because they do not get 'stuck' in difficult problem instances due to their element of stochasticity. Verner et al. [19] describe another stochastic algorithm based on genetic algorithms, which was also tested in the framework which Christensen et al. set up. A short summary of these algorithms and how they compare against each other is given in section 3. We describe a new stochastic algorithm, also based on genetic algorithms, which however has quite a different angle than the algorithm of Verner et al.

The algorithm which is presented in this paper generates high quality labelings for maps consisting of point features. The algorithm can also be easily extended to handle other problem instances involving point features, such as problems which allow labels to be deleted, have preferences for label positions, consider different priorities for different points or have different label

sizes for different points. It is also our aim to extend this method to handle line and area features as well in further research.

Besides solving the map labeling problem for point features satisfactorily, a philosophy for engineering genetic algorithms for GIS applications is explained, which involves the notion of robustness. Genetic algorithms often have lots of options that have to be 'tuned', which is a lengthy process. The genetic algorithm presented in this paper is robust in the sense that it performs well without the need for setting options by trial and error.

This paper is organised as follows. First (section 2) a bit more is said about the map labeling problem and why it is difficult. Then (in section 2.1) we will briefly say something about the properties maps have and how this influences the problem difficulty. The articles of Christensen et al. and Verner et al. are briefly summarised in section 3 to give a comprehensive picture of the state of affairs at the moment, and how the algorithms compare against each other. In section 4 we will start with describing the workings of genetic algorithms in section 4.1 and then describe the new genetic algorithm in detail. We will compare it against the other algorithms in the framework of Christensen et al. in section 5. The notion of robustness is explained throughout the paper, and in section 6.

A note on presented experimental results. In this article numerous experimental results are presented. With the exclusion of the maps from figures 26 and 46, all maps were generated randomly according to a procedure explained in section 5. For comparisons that investigate the effect of changing a part (for example, an operator) of the GA, five maps of 500 cities were randomly generated (see figure 4) which were used in all these experiments (shown in figures 12, 17, 22, 23, 25, 28, 29, 32, 33, and 42). Every graph is the average of five runs done on those maps. In the key of each figure every run is specified by a name (like "r001") and in Appendix A the various parameters used in these runs are enumerated. For the figures which show these runs, the *average* fitness of the population was plotted unless otherwise specified (the legend says 'max' when the run plots the fitness of the best individual). Graphs stop when the population is deemed converged.

The figures in which the results of different algorithms are compared (like 40) the variation is not plotted in order to avoid crowding the picture too much. Instead, the data values and standard deviation are given in Appendix A.

2 The map labeling problem

Consider again the problem of label placement for maps, and let's pose the following definition:

Definition 2.1 [Label placement problem]
Given is a set of n points in the plane, with labels associated with them. Each label can be placed in a fixed number of predefined positions and orientations. Give a positioning for each label such that the number of labels which do not intersect other labels is maximised.

Next we define the specific instances we will be studying. First we define different problem instance for the two placement models.

Definition 2.2 [Four position labeling]
The problem is as in definition 2.1. Furthermore, each label is an axis parallel rectangle. The height of all labels is equal. The width of the label can vary from label to label. Both the width and the height are fixed for every label. Each label can be placed in one of four positions as seen in figure 2.

Definition 2.3 [Eight position labeling]
The problem is as in definition 2.2. Additionally, each label can also be placed in one of four positions as seen in figure 3.

Also we will study the problem where features are deleted if the map is so crowded that it is impossible to place all the labels.

Definition 2.4 [Four position labeling with label selection]
The problem is as in definition 2.2. Additionally, it is allowed to delete a label entirely. In that case it will not be considered a free label, neither will it intersect other labels.

Several things are important in the above definitions. First of all, the problem is reduced to maximising the number of free labels and disregards other aspects of the problem such as preferences for positions. We will however sometimes address other aspects, but we will only use the above definitions of the problem for our comparison experiments with other algorithms. Also, since we assume that the names are all written in the same font and have therefore the same height, we demand the same height for all labels. Rescaling of labels is not allowed either. Although a cartographer can choose from infinitely many positions, we have discretised the problem such that there are only four positions which are allowed. Finally, only intersections between labels are considered, so a label does not intersect a point feature whose label is deleted (it is assumed that the point feature is deleted also). Note that this is just a matter of choice. For example figure 26 does consider intersections between point features and labels.

A solution for the label placement problem is a position assignment for the label of each point, therefore each labeling is a solution. An optimal solution is a solution that has the maximal number of labels without a conflict. A solution looks like the map in figure 4. Since the conflicts in that map are irresolvable, this is also an optimal solution.

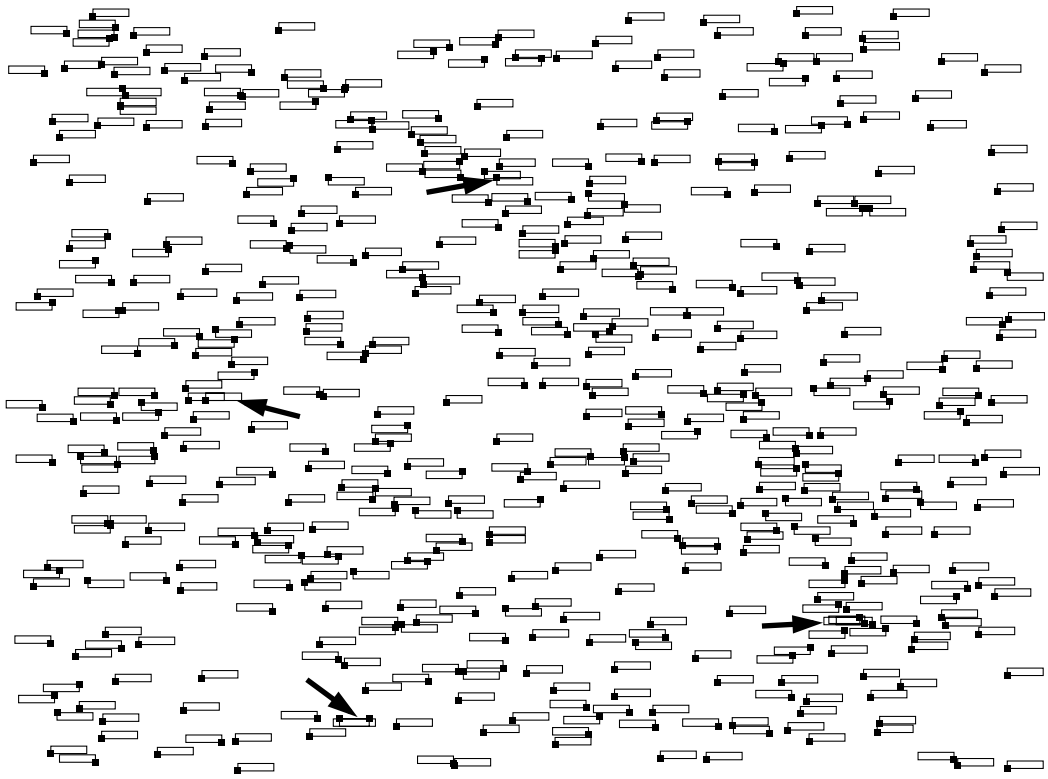

Figure 4: An example of a map which is a solution for the label placement problem. Some irresolvable overlap still exists (arrows are added to indicate where).

The right placement of a label depends on the placement of surrounding labels, which depends on their neighbours, and so on. This kind of interaction has as a result that the problem can be

very complex. Also, since this interaction can influence regions which are some distance away from the influencing label, local optimisation alone can not be powerful enough to solve the problem in a satisfactory manner. Global optimisation in some form should be present to produce high quality labelings.

2.1 Properties of maps

Every point on the map has a number of neighbours that interact with it since the placing of their label influences the placing of the label of that point. The notion of a neighbour should be more precise, so we consider only a neighbouring point which could cause a conflict. We'll define such a point as a *rival*:

Definition 2.5 [Rival]
Given is the set of points P and associated labels with the constraints of definition 2.1. A rival of some point $p \in P$ is a point $q \in P$ (with $p \neq q$) for which the set of all possible label positionings of p and q contains a label intersection.

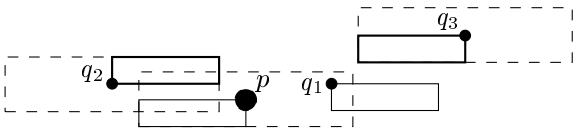

Figure 5: The rivals of the central point p.

For example, consider figure 5 (under the constraints of the problem in definition 2.2). The points q_1 and q_2 are rivals of p, but point q_3 is not. It is clear that the rival relationship is symmetric (if p is a rival of q, then q is also a rival of p). Furthermore, two points are rivals if the union of their labeling positions intersect (as can be seen in the picture when the dashed boxes intersect).

An interesting property of a map is that it can be decomposed into separate problems which can be solved independently. A point should not be labeled independently from its rivals, because it can interact with it. This also holds for the rival of that rival (being another point). The rival relationship knits points together which have to be solved as a group. As such, it induces a graph:

Definition 2.6 [Rival graph]
The rival graph of a map consisting of the set of points P is the graph $G = (P, E)$ where the nodes consist of the points in P and the edges connect rivals: $E = \{pq \mid p \text{ is a rival of } q\}$.

The separate problems mentioned above correspond with connected components (sets of nodes for which a path can be drawn between each pair of nodes in the set) in the graph.

It is possible to break down a rival graph into its connected components (see figure 6) and solve these independently, afterwards merging the results. For example, for the first map (which will be described in section 5) of 500 cities which is used in various experiments in this article (see also appendix A), there existed 86 different connected components. The two largest components contained 32 cities and the 35 smallest components contained only a single city.

We did not use this property in the rest of the article and always solve the entire map as one problem. Our reasons for this are as follows:

- The algorithms we compare our algorithm with (hillclimbing, simulated annealing and another genetic algorithm, see section 5) also did not solve connected components independently, although they could. Therefore comparing is clearer if we use the same conditions as those algorithms.
- Our algorithm is able to find high quality solutions without the need for the problem to be one connected component. If each connected component would be solved separately, the quality of the resulting solutions would not increase, but the running time of the algorithm would decrease. This is due to the implicit separate way all parts of the solution are handled (which is caused by using a kind of uniform crossover, see section 4.5.1).

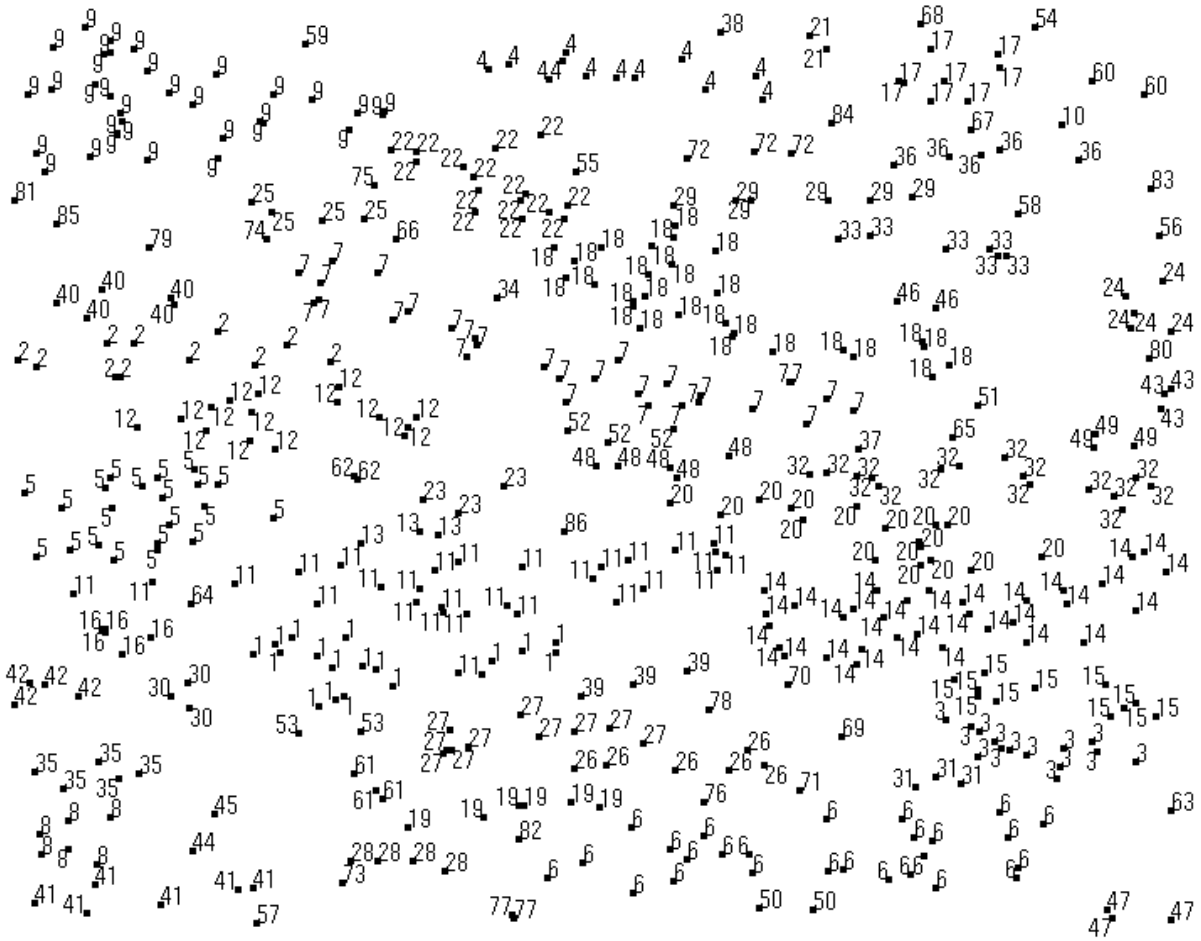

Figure 6: A map labeled with its connected components numbered (compare with figure 4).

- If the problem is extended to handle line and area features as well and the rival relationship is extended to consider intersections between features, the map will very likely contain very few connected components. For example a river would be a rival of all the points lying nearby and would therefore allow a path to be drawn in the rival graph between very remote points. Although we did not consider the problem with line and area features, we think it is wise not to build optimisations which will be useless in the future.

- If the map density increases, the number of connected components decreases (for example, a randomly generated map with 750 cities contains 27 groups, and a randomly generated map of 1000 cities contains 11 groups where 983 cities are in the same group). As a result, the benefits of solving a connected component independently dwindle to none when problems get harder and this would make comparing easy problems with hard ones more difficult.

Solving the problem as a whole only influences the population size and therefore the running time of the algorithm. The population size is directly related to the difficulty of the problem which is solved. A problem needs to be solved with a population size above a certain treshold to get a solution which has the possible amount of quality. Using a larger population size would gain little, but using a smaller population size would cause a decrease in quality (more conflicts remaining in the solution). If the problem is solved as a whole, the connected component which is hardest determines the treshold above which the population size should be. This means that the population is oversized for the connected components which are easier to solve. As a result, since the population size is related to the running time, solving the problem as a whole causes the running time to increase but the quality of the solution to remain the same. There is a situation in which the hardest connected component does not determine the population size, and that is when there are very many easier problems. Due to stochastic effects the population size has to be increased to be able to solve every connected component. In that case the hardest connected component would become oversized.

3 An overview of methods for map labeling

Several algorithms for map labeling already exist, and it is necessary to know them in order to be able to compare the new algorithm against them. This is not the place for thorough descriptions of these algorithms, but they will be briefly explained. The algorithms mentioned were all compared in the framework of Christensen et al., and more detailed descriptions and implementation issues can be found in [2]. The genetic algorithm of Verner et al. is also mentioned, and a description can be found in [19]. In [5] Djouadi describes a system for the full map labeling problem using a genetic algorithm. However, he uses methods which are not standard GA practice and which he unfortunately does not argue for. Also he does not give results which can be used to compare against. As a result, we did not consider the algorithm from Djouadi in the present paper.

We review therefore the following algorithms which give a comprehensive view of what is available in map labeling literature:

- Greedy algorithms [13, 20]: these algorithms perform local optimisation on every city in succession and then terminate. No form of global optimisation is done, which makes the algorithms fast but their solutions can be significantly inferior to other algorithms.

- Discrete gradient descent [2]: these algorithms iterate a procedure for the whole map until no improvement can be made. The procedure first enumerates all possible, allowable changes that can be made and then picks the one which gives the most improvement. Variants in which kind of changes are possible exist. For example, one could only look at repositioning a label for a city. Another choice could be to look ahead two or three steps of label repositionings, which would give better solutions at higher computational cost.

- Hirsch [10, 6]: the algorithm of Hirsch works by viewing the map labeling problem as a dynamic system of repulsing labels. Each iteration all conflicts between labels are considered

and a movement is calculated based on the amount of repulsion between labels. The amount of repulsion is defined in terms of how much the labels overlap. If a label intersect multiple labels, the repulsions are added. This results in a movement for each label which is carried out.

- 0/1 linear programming [21, 22]: the algorithm of Zoraster is based on formulating the map labeling problem as a zero-one integer programming problem and using Lagrangian relaxation and heuristic methods to solve it.
- Simulated annealing [2, 23]: Christensen et al. have devised an algorithm which is based on simulated annealing. This algorithm will be explained in more detail later.
- GA of Verner et al. [19]: a genetic algorithm which uses masking and copying genes to both children to preserve good solutions was implemented by Verner et al. This algorithm will also be explained in more detail later. (This algorithm was published after the paper of Christensen et al., and therefore is missing from figure 7. See section 5 for a comparison of this algorithm with the one from Christensen et al. and our algorithm.)

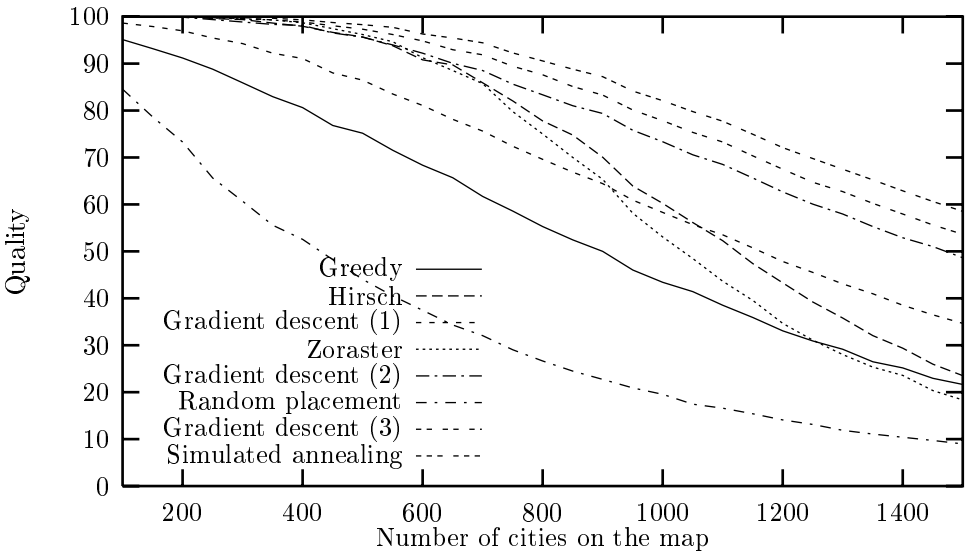

Figure 7: A comparison of the map labeling algorithms (data taken from [2]). Gradient descent has the number of allowable changes in series between brackets.

The results of Christensen et al. (given in figure 7) suggest that most heuristics fail because they can not handle certain local optima very well and because of cycling in the algorithm. The first problem is a consequence of the non-linearity of the problem. Because so many factors interact, the global optimum (the best solution) depends on the right way of interaction for all these factors. If the solution is not optimal, it is possible that in order to change the solution to a global optimum, it has to be made worse before it can get better again. Heuristics which work by strictly improving a solution, can therefore get 'caught' in these local optima. An example will help explain this. Consider figure 8. The first solution is what was found after using some algorithm (some unlabeled cities are drawn to show that no labels can be placed there). We now have found a local optimum: the solution is not the best one there is, and it can not be made better by repositioning a single label. Instead, both middle labels have to move, of which only one label can be moved at a time. The moved label will obstruct a label which was free, which degrades the solution. However, this gives room for the other label to move and the final result has

two labels free instead of one. This problem plagues all heuristics except the simulated annealing algorithm and the genetic algorithm of Verner et al.

The second problem because of which some heuristics fail is the problem of cycling. Suppose there exist two solutions named a and b which are equally good, but not the global optimum. If the algorithm improves solution a to b and b to a, it will never end. This happens with the algorithms of Hirsch and Zoraster.

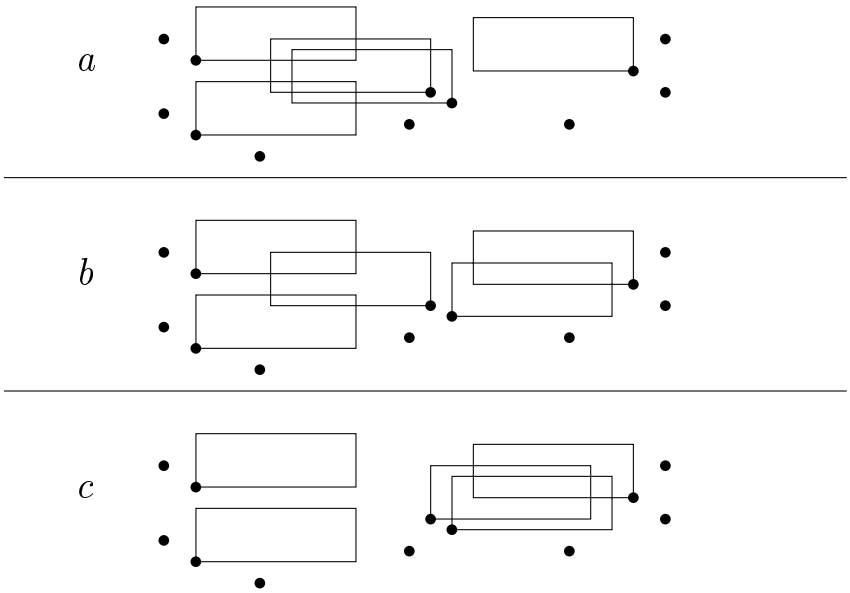

Figure 8: Leaving a local optimum can make the solution temporarily worse (also see figure 34).

It seems like the determinism of the first four algorithms is causing the trouble. For an algorithm to cope with difficult instances, it should not exhibit deterministic behaviour. The simulated annealing algorithm and the genetic algorithm both work with an element of chance. Moreover, they combine local optimisation with a form of global optimisation. Both algorithms do this in a different way, but it is clear from the comparison experiments that this is the way to go. Simulated annealing and the genetic algorithm of Verner et al.[1] are superior to the other algorithms, in terms of the quality of the resulting solution: it has the largest number of labels which do not overlap other labels.

Since the stochastic algorithms give the best performance, we describe in a little more detail the simulated annealing algorithm, the hill climbing and the genetic algorithm with masking. For fully detailed descriptions, the reader is referred to the original papers. These algorithms will be considered in section 5 where comparison experiments are presented.

Simulated annealing. The process of heating a metal until it melts and then slowly cooling it until it solidifies is called annealing. The idea is that the atoms in the metal become completely random in motion because of the heating, and the cooling is done sufficiently slow to let the atoms crystallise in a highly ordered structure. If the cooling is done too fast it is like the crystallisation is stopped half-way, 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 (see [12]). Consider a problem and a way to encode the solution to this problem in a data structure. We start with a randomly generated solution. Simulated annealing works by

[1] The genetic algorithm of Verner et al. was tested (in [19]) with a eight-position model against the algorithms in the framework of Christensen et al. which use a four-position model, so based on known results from literature it can not with certainty be said which of the two algorithms performs better. In section 5 however implementations of the algorithms are compared with each other.

randomly picking a part of the solution, altering it, and seeing if it has become any better. If it has, the change is kept and another iteration is started. If it is not better, 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 chaotic state. The probability of keeping a change is high (instead of starting with one, a temperature implicating a probability of $2/3$ is used). During the run of the algorithm, the temperature is lowered sufficiently slow 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 is heading for the global optimum. The way in which the temperature is lowered is called the *annealing schedule*.

Simulated annealing for the map labeling problem can be done by simply repositioning a label each iteration and observing the change in the number of overlapping labels. If the number has decreased (or is equal), the change is kept. If the number has increased, the change is kept with probability $P = e^{-\Delta E/T}$ where ΔE is the change in the number of overlapping labels and T is the temperature, otherwise it is made undone.

In what way does this algorithm exhibit local optimisation and global optimisation? Local optimisation arises in the form of keeping a change when it is good. The key to global optimisation is of course keeping a label when the change is bad, since a solution sometimes has to degrade before it can become better again. The annealing schedule ensures global optimisation is reached, provided of course that the schedule is not too fast in lowering the temperature.

Hill climbing. If the annealing schedule is such that the temperature is always zero, the algorithm reduces to the case of a *hill climber*: changes are made randomly and are only kept if they give no deterioration of the solution. If the problem is not too complicated, a hill climber performs surprisingly well. For instance, the problem of maximising the number of free labels can be solved relatively well with a hill climber. Alas, when the problem is extended to the case where labels have preferred positions and labels should be selected for deletion (because the map is too crowded), the hill climber performs poorly. Nevertheless, it is an important algorithm because it is so simple to implement and because the rate of improvement is very high at the start of the algorithm.

Genetic algorithm with masking. The details of how genetic algorithms work are described more thoroughly in section 4, but an intuitive explanation is given here to understand the way the algorithm of Verner et al. works. A genetic algorithm is based on inspiration from the theory of Darwinian evolution. Consider a population of organisms which compete for a certain resource (say, food). Also consider the fact that there exists a mechanism for inheritance of traits with small differences ("descent with modification" as Darwin called it). Then each individual in the population will be more or less fit for the task of getting the resource. The ones which are most fit are more likely to reproduce and they will have children with the same traits which made them fit. Since small differences occur during reproduction, there will be variation in fitness. The theory says that because of these conditions there will be *selection pressure* towards organisms that are highly fit in getting the resource. This process is called *adaptation*.

The same principles underlie genetic algorithms[2]. There is a population of encodings of solutions (this can be just an array which holds the position of a label for each city). There is no resource for which they are competing and which implicates a fitness. Instead they get a fitness explicitly assigned to them (this can for example be the number of labels which do not overlap other labels). Fit individuals produce more offspring. Mating produces children by exchanging pieces of the solution, in order to preserve variety for selection to act on. Letting this system evolve produces individuals which get more and more fit, until the population is converged. Hopefully, the solution found will be a global optimum.

[2] There are, however, major differences. Natural evolution for example works with a population which is largely converged, in contrast with a genetic algorithm which begins with a randomly generated population.

A genetic algorithm has an element of local optimisation because partial solutions which were found to be good will propagate through the population. Also, global optimisation exist because combining all these partial solutions yields the final solution, which should be the global optimum.

Many choices have to be made when designing a genetic algorithm. Verner et al. chose a form of *uniform crossover* for the reproduction operator. This means that when two individuals have been selected to make two children, the children have for each partial solution (in this case, the positioning of the label for a specific city) equal probability that it originated from one of the two parents. Therefore, this operator is highly disruptive and it is very likely that good solutions get chopped up instead of being passed integrally to their children. To overcome this problem masking and copying were used. Whenever a piece of the solution is found which has no conflicts, it is masked and thereby protected from being chopped up. If the parent with the good piece is mated with a parent which has a conflict in that piece, the good piece is copied to both children. In other choices the algorithm is fairly standard.

4 Genetic algorithms for map labeling

In this section we will explain the genetic algorithm (which we will often abbreviate as 'GA') that was developed to solve the map labeling problem. Before we delve into the details of this specific genetic algorithm, we will give some background information on GA's in general and specific elements like the elitist recombination scheme and masking we used as well. This will be done in subsection 4.1. Also the general outline of the GA will be given, after which detailed descriptions follow in the next subsections.

4.1 Introduction and overview

Genetic algorithms are stochastic algorithms which have their inspiration from the theory of Darwinian evolution in biology. They work with a *population* of solutions (encoded in some manner) which can be evaluated by a measure of quality to yield the *fitness* of an individual. The GA generates new individuals by mating members of the population which replace unfit individuals by mating fit ones. Mating usually occurs in the form of *recombination*: on the codings of the solutions *crossover* is performed to make new solutions. After recombination *mutation* is sometimes performed. The result of all this is that the solutions in the population become better as defined by the fitness measure until hopefully the desired solution (the solution with the highest fitness) is found.

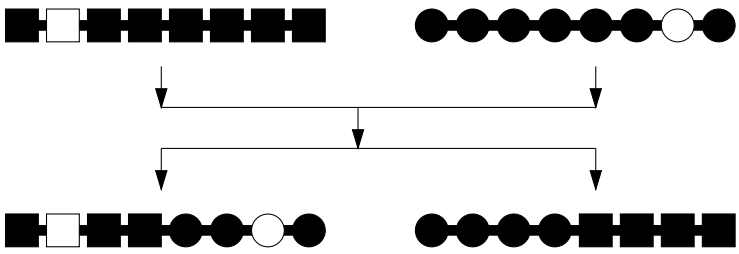

Figure 9: One point crossover acting on strings of genes from different parents. Two building blocks are accentuated.

An example can clarify this idea more. Suppose we have a complicated mathematical function of which we want to know the highest value in its range (the optimum). We can use a GA as a function optimiser for this problem. The measure of fitness is simply the function itself: higher values are better. The coding can be simply a binary string which is long enough to hold the full range of possible values of its domain. We now start with making an initial population of N individuals which are randomly generated. This means we have got N random binary strings, of

which we evaluate the value using our function. This gives us fitness measures of the individuals. The hope is that each string is build up from pieces which are 'good' (they are often called *building blocks*) and that we can put together somehow all pieces from different individuals into one individual, and get our optimum. We try this by using various operators on the individuals of the population, of which the most important are *crossover* and *mutation*. Crossover works on two parents[3] and takes a chunk from one parent and a complementary part from the other parent. This produces two children (see figure 9). Mutation replaces a small part of the encoding by a new part, in the hope that it will prove to be better. After the generation of children replacement is performed by replacing unfit parents with fit children.

This process is iterated for some time and eventually the population will consist of individuals of the same fitness (the population has *converged*). If the GA is built right, this fitness should be optimal or near optimal. However, it is in most cases almost impossible to guarantee this.

What does it mean to say a population has converged? We will use the following definition: a population which is being adapted by an algorithm is said to have converged when the algorithm can not produce fitter (as defined by the fitness function) individuals.

Convergence is a characteristic of a population *in relation to* an algorithm. This is obvious once one realises that otherwise it would always be possible to invoke exhaustive search and find the optimum.

A standard GA uses the algorithm shown in figure 10. Encodings are strings of values taken from some alphabet. We use the concept of a *mating pool* to make the mechanism of selection and producing a new population more explicit. The mating pool is the place where new individuals are made which will alter the population. In the mating pool selected individuals are placed (sometimes with multiple copies, if they are very fit), which are altered by the process of recombination, mutation and maybe other operators. Then the contents of the mating pool replaces a part (or maybe the whole) of the population, thus forming a new population. [4]

This is still very general, so we will clarify several points:

- Selection: selection of individuals can be done in several ways. For example, one could place a number of copies of an individual in the mating pool proportionate to its fitness. Unfit individuals would not reproduce. Another way of performing selection is to sort the population on fitness and look at the rank of an individual in this list. There exist a wide variety of schemes for selection. We will have more to say about this later, when we discuss the elitist recombination scheme. Another point which is important for selection is the amount of individuals that is selected. One could take a mating pool equal in size to the population (this is called a generational scheme, because every iteration involves a generation of individuals), or take only two (this is called an incremental scheme) or something in between (which has been called a steady state scheme, a scheme with a generation gap or an overlapping populations scheme by various authors; we will use the term steady state).

- Crossover and mutation: what should the values of P_c and P_m be? A large value of P_c will tend to destroy things that were found to be good by selection, but will hopefully result in building things that are better. Setting P_m to a value larger than zero will cause possible destruction of good building blocks, but might introduce information that was lost due to convergence. In [4] DeJong suggests values of 0.6 for P_c and 0.001 for P_m (where it is the probability for each location on the string to be mutated, instead of the probability for the string as a whole to be mutated).

[3] More exotic variants using more parents exist but are not widely used.

[4] Note that the metaphor of individuals which reproduce to make new children is a bit strained here. One way of looking at it would be that the members of the mating pool are children of selected individuals from the population, which means the operators were applied before placement in the mating pool took place. Another way of looking at it (still holding on to the reproduction metaphor) would be that the mating pool holds the selected individuals, after which operators are applied and the produced children are immediately placed in the new population. A more natural way to look at it is by abandoning the metaphor a little. In that case, the mating pools holds the selected individuals. Operators are applied, and the resulting individuals instantaneously replace their parents, after which the mating pool holds only the children.

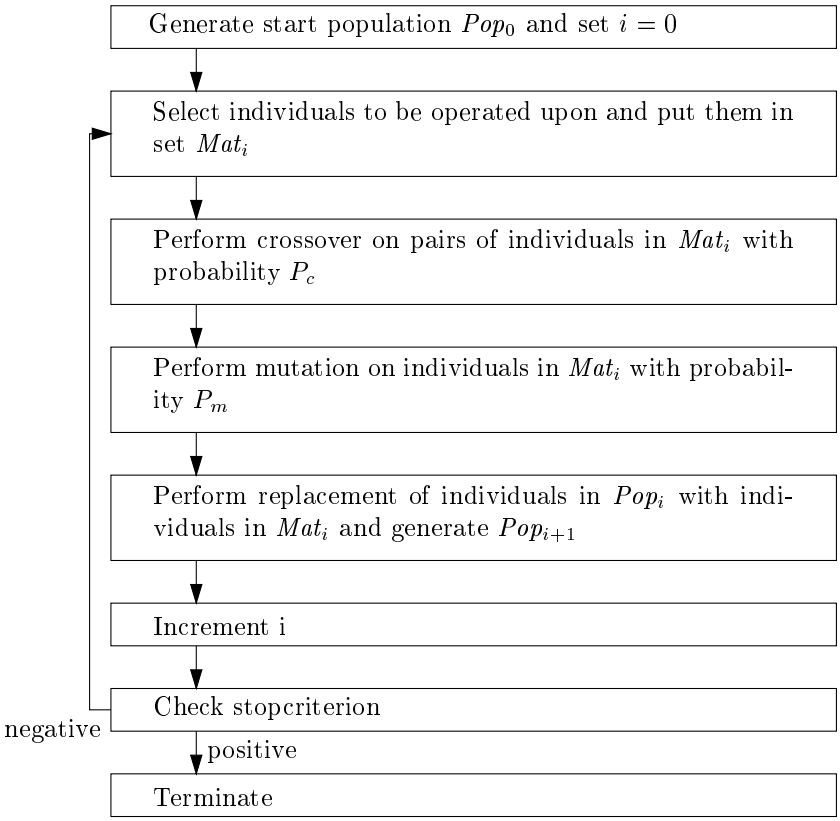

Figure 10: The algorithm for the standard GA. *Pop* is the population, *Mat* is the mating pool.

- Replacement: it has to be decided who should live and who should die, because the population has a finite size. One choice is to simply replace the population with the mating pool. This is done in the generational GA, where the mating pool is the next generation of individuals. In the incremental GA, more options are available. The two individuals in the mating pool could for example replace their parents, replace randomly chosen individuals, or replace the two worst individuals in the population. Similar choices exist for a steady state GA.

- Stop criterion: deciding when the algorithm is done is not always a trivial task, since it is not certain when there will be no more improvement. It is always possible that the individual with the highest fitness found does not represent the optimum solution. Mutation could for example introduce the lucky (but unlikely) chance that transforms a near optimum solution to an optimum one. Several criteria therefore are possible. One of them is waiting until the period of time that there was not any improvement stretches beyond a certain limit. Another stop criterion which could make sense when there is no mutation is waiting until the average fitness in the population is equal to the maximum fitness in the population.

4.1.1 Exploitation and the elitist recombination scheme

In choosing a selection scheme for use in a GA a balance has to be struck on how much exploitation one wants the GA to use. The GA exploits the information which is present in the population to converge the population to a good solution. The most exploiting selection scheme therefore would be the selection scheme that filled the whole mating pool with copies of the same, fittest individual. Clearly this is too much. The other extreme is no selection at all: the whole population is copied into the mating pool and the offspring forms the next population. In this case the population would only converge by genetic drift, which is the effect that random fluctuations in proportions build up until they overwhelm the whole population[5] (see [8] and [1] for discussions on genetic drift).

So the *selection pressure* should be something in between[6]. Note however that this can vary over time: you might want a low selection pressure to start with so you can explore different solutions, and turn to a high selection pressure later on to exploit the information you have gathered. On the other hand, you might want a high selection pressure at the start to quickly exploit the global perspective that the initial populations offer, and drop the pressure later on to give room to exploration.

The GA that was developed to solve the problem of map labeling makes use of the Elitist Recombination scheme (see [18] and [17] for a full description and discussions of properties). This scheme has several useful properties (such as a constant selection pressure), which we will describe later. First we will describe the idea of elitist recombination, which fortunately is very easy to grasp.

The elitist recombination scheme combines selection, recombination and replacement. It can be most naturally used in an incremental GA, but is also applicable with other replacement schemes. In the elitist recombination scheme, two parents are selected from the population (the size of the mating pool is therefore two). On these parents crossover is performed, which results in two children. Of these four individuals, the two with the highest fitness (choosing the children in case of ties) are placed back in the population in the place of the two parents. Therefore, if the children both have a higher fitness, they replace their parents. On the other hand, a parent can never be replaced by a worse individual (this is called *elitism*). See figure 11 for a graphical explanation.

[5] Consider for example a collection of red and blue colored balls where the number of red balls is equal to the number of blue balls. Pick randomly two balls and change the color of the second ball to the color of the first ball. If one keeps doing this, all balls will eventually get the same color. The process is called a *random walk with absorbing barriers*. The analogy with genetics is that each ball has a gene for color and reproduction is asexual (the sexual variant (with two parents) demonstrates the same effect).

[6] The intensity of the selection pressure can be more formally defined (see [15] and [17]) as "the expected average fitness of the population after applying the selection scheme to a population with standardized normal distributed fitness" (from [17]).

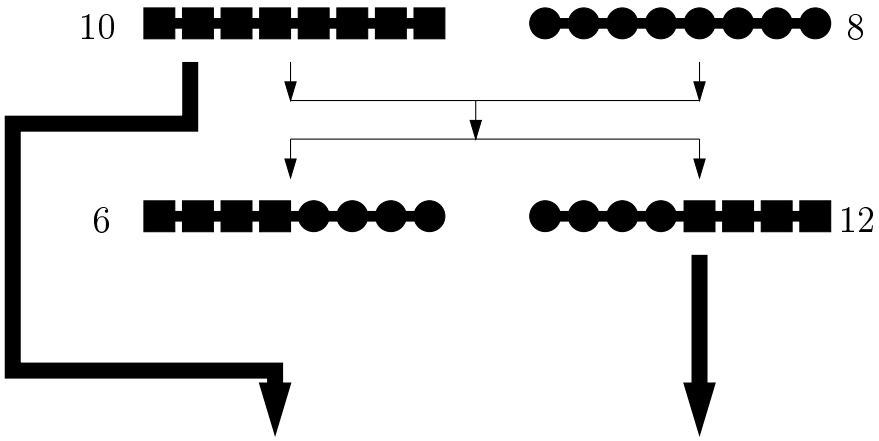

Figure 11: The elitist recombination scheme. The numbers represent the fitness of the individuals. The thick lines show which individuals replace the parents.

The elitist recombination scheme has the following useful properties:

- Tunable, constant selection pressure: the selection pressure is constant, which means that gradual but sure progress will be made. The selection pressure can easily be tuned by making a slight modification: instead of picking both parents at random, one of them can be the winner of a tournament of several randomly picked individuals.
- Elitism: since individuals are never replaced by less fit individuals, there is no fear of losing a particular good solution.
- No problems with recombination: the balance that had to be made (see also section 4.5.1) between crossover disruption and building block mixing (getting better individuals by combining good pieces of different individuals) and which led to setting P_c (the probability for performing crossover) to a specific value, is not necessary here. We can safely always perform crossover, since disrupted building blocks will degrade the fitness of the individual that has them.
- Less sensitive to undersized populations: since selection and replacement only work on the level of a family of four individuals, effects of the rest of the population are relatively minor. This does not mean that good convergence always will occur, but it prevents a undersized population from converging to the few relatively fit individuals. Tournament selection for example works by picking several individuals at random and putting the one which is most fit in the mating pool and iterates this procedure until the mating pool is filled. If the population is undersized the tournament size is relatively large compared with the population size and this will lead to a mating pool filled largely with the few lucky individuals that had relatively high fitness. This will result in premature convergence. The difference between the elitist recombination scheme and tournament selection is that the elitist recombination scheme has random selection of the parents, whereas tournament selection (as most selection schemes are) is biased in the selection of the parents (elitist recombination introduces a bias when replacing the parents with the most fit individuals). As a result with elitist recombination the mating pool will consist of different individuals but with other selection schemes the mating pool can (and probably will) contain multiple copies of very fit individuals.
- Ease of implementation.
- Conceptually simple.

Using the elitist recombination scheme immediately pays off: see picture 12 for a comparison between a run done with a generational GA using roulette-wheel selection and a run done with the (incremental) elitist recombination scheme. Since the label intersection test is the most atomic action all algorithms perform most of the time, we used this as a measure for computational effort (see also section 5 on this).

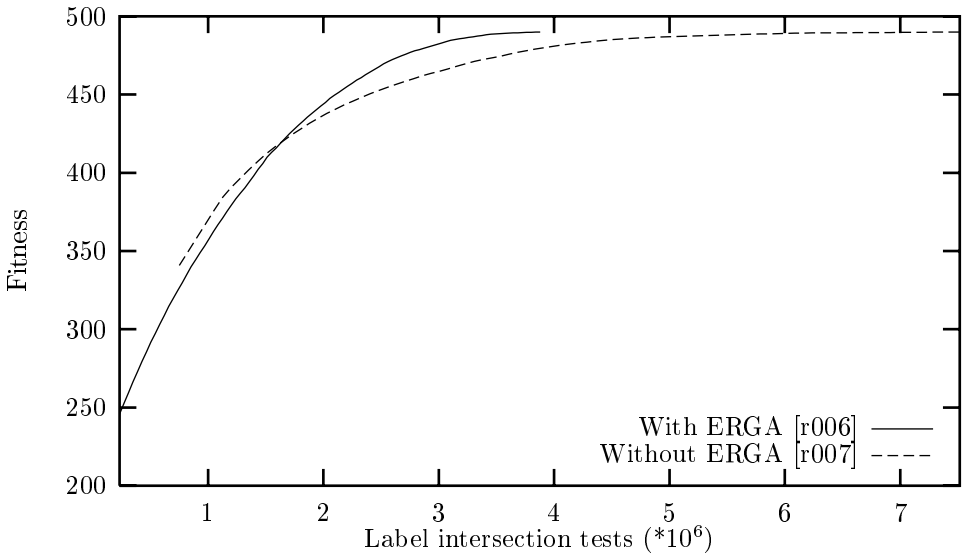

Figure 12: Comparison between roulette-wheel selection and the elitist recombination scheme. (See section 5 for an explanation of the unit of measure for the horizontal axis.)

4.1.2 Masks and multi-masks

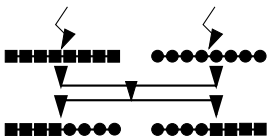

Figure 13: One-point crossover.

As we have seen in section 4.1, crossover can be performed in several ways. The most straightforward is the one-point crossover: pick a point at random in the list encoding of the two individuals and swap from that point (see figure 13). One could expand on the idea and take two-point crossover: pick two points at random and swap the string between them (see figure 14).

One could also decide for each location separately from which parent the information should be copied. This is called *uniform* crossover. This is like tossing a coin for each location to decide if it will be copied from the first or the second parent.

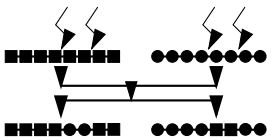

Figure 14: Two-point crossover.

Sometimes one wants to generalise the idea and provide a way of specifying what the crossover looks like. This can be done with *masking*. A mask is a bit string with the same length as the string on which the crossover operator works (see figure 15). For each location, if the bit is set, then the first child gets the value for that location from the first parent and the second child gets it from the second parent. If the bit is not set, this situation is reversed (first child gets the information from the second parent and so on). One-point crossover can be specified by setting the bits in a mask until a picked point and clearing the bits after that point. Uniform crossover can be specified by taking a random bit string.

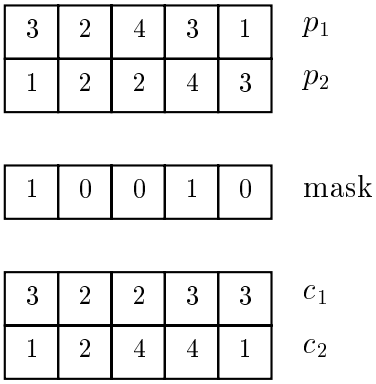

Figure 15: An example of masking. The parents are denoted p_1 and p_2, the children are denoted c_1 and c_2.

Masking has several uses. First, it can be used to specify the crossover operator a priori. In this case one initializes a set of crossover masks and chooses randomly from that set when performing crossover. It is now also easy to combine several masks by taking the inclusive OR from these masks and generating a new mask. Such a mask will be called a multi-mask, and a bit is set in the multi-mask when the bit for that location was set in any of the masks which were used to generate it.

Another use is the case in which the masks are made personal for each individual. In this way it becomes possible to protect building blocks from being disrupted, or to focus the mixing of strings in areas where things are considered not optimal. The masks will therefore change during the run and provide control for the crossover operator instead of blindly cutting strings up.

4.2 Outline of the GA for map labeling

We will briefly outline the GA for map labeling now (called loGA from now on — from '**l**ocal **o**ptimiser **GA**'). The loGA starts with generating an initial population. Procedures which generate an initial population are described in section 4.3. After that, the loGA keeps evolving the population until a termination criterion is satisfied.

The loGA makes use of the elitist recombination scheme (which we described in section 4.1.1) to evolve the population. The elitist recombination scheme performs crossover on two randomly selected individuals in order to generate the two children. The crossover operator is described in section 4.5.1. As will become apparent there, the use of local optimisers is very important, and a further section (which is section 4.5.2) is devoted to describing local optimisers. After the children are generated, mutation could be performed on them. The loGA does not make use of mutation in this way however.

The elitist recombination scheme replaces the parents with the two best individuals from this family of four. It can decide which one is 'best' using the fitness function, which gives every individual a measure of quality. Fitness functions are described in section 4.4.

A separate section (section 4.6) deals with the fact that the activity of the GA can be focussed on regions which contain conflicts and leave regions which are already optimised out of time consuming operations, thereby gaining a speed benefit.

4.3 Initialisers

GA's try to find an optimal solution by adapting solutions in the population to become more like an optimal solution. This describes what happens between populations, but what about the initial population? What are the conditions for making the first population? Since good partial solutions from different individuals should eventually end up in one individual (where the combination of good partial solutions gives the optimal global solution), it follows that these good subsolutions should come from somewhere. These subsolutions are sometimes called 'building blocks' because they are the bricks with which the desired solution is built, or so the building block hypothesis (see section 4.5.1 and [7]) tells us. There are several sources for building blocks:

- Building blocks exist integrally in the initial population.
- Building blocks get formed by chance during crossover.
- Building blocks are explicitly constructed by local optimisers during the run of the algorithm.

The GA we describe here uses local optimisers (see section 4.5.2) but does not use random mutation (see section 4.5 for the reasons) which is commonly also used as a source for building

blocks in GA's. Here we focus on the first two points and see what it means for the initial population. The population consists of individuals which are *chromosomes*: strings of *genes* (each location is a gene) which can take any value (called an *allele*) from a finite alphabet. A gene in the GA for map labeling corresponds with a city and each allele corresponds with a positioning for the label of that city.

Note that the use of the initialization operator is related to the crossover operator. Crossover operators need to mix (put together) building blocks from the parents in order to make children with more building blocks (which become fitter as a result). On the other hand, since crossover takes a piece from each parent, there always is some amount of disruption after applying a crossover operator (see figure 16). Disruption can split up a building block and therefore destroy it. Instead of disruption, building blocks can also get formed accidently, although this is a less likely scenario.

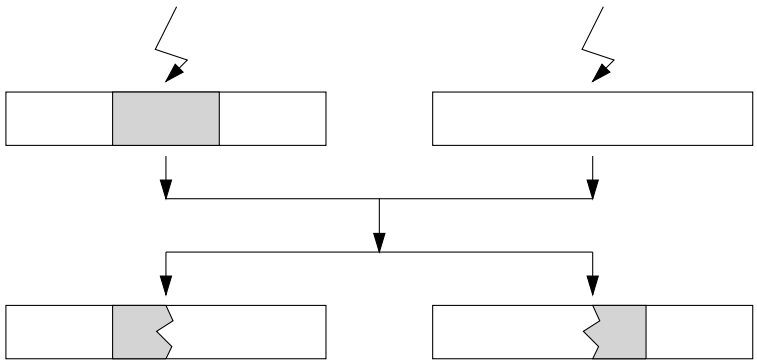

Figure 16: An example of disruption. The building block in the first parent is split by one point crossover into two fragments.

The initialization operator should achieve two goals:

1. Building block supply: constructing integral building blocks in the initial population which can be mixed later.

2. Providing variation to form building blocks with during crossover or as a basis for a local optimiser to act on.

The first point (building block supply) is based on the assumption that if the initial population contains all the building blocks, the crossover operator will be capable of putting them all together in one individual during the run of the algorithm. So initialization provides the building blocks and crossover mixes them.

The second point (providing variation) is based on the expectation that the crossover operator can make new building blocks by itself because many different combinations of the genes of the parents can be formed during crossover and these may contain new building blocks. In that case it is necessary to provide enough variation so these combinations can actually be formed. This means that all alleles should be contained in the initial population in equal proportions.

Variation is also needed if local optimisers are used. (Geometrically) local optimisers are not common in conventional GA's, but they play a very important role in the GA which is described in this paper. Other hybrid GA's do also use local optimisers, but they are different from the local optimisers used in this article. For the local optimisers in the usual hybrid GA, 'local' means 'local in the fitness landscape'. The local optimiser is for example a hillclimber. We mean 'local' in the sense of only acting on *part* of the solution, which is easy to do since we are dealing with an geometrical problem.

A local optimiser tries to construct a local solution (it is applied to one city and considers only that city and its rivals). However, in order to attain a globally very fit individual, that local solution should fit in the overall picture. This can not be ascertained by the local optimiser,

therefor it is necessary to provide enough variation so the local optimiser will be applied to different configurations. This will result in different local solutions, one of which is likely to fit in the global solution.

The conventional GA uses an initialization operator which can satisfy both goals: for each gene an allele is picked randomly out of the set of possible alleles. This provides the variation which the second goal demands. The first goal is also met since the probability of forming a building block will be high if the population size is large enough.

Another option for the initializer is to try and construct building blocks explicitly in the initial population using a local optimiser. It is of course essential that the GA designer has a reasonably good idea of what the building blocks of the problem at hand are. A discussion of what the building blocks in the problem of map labeling are is given in section 4.5.1 where it is argued that the optimal configuration of a city and its rivals is a building block. As discussed above, variation is necessary for a local optimiser, so the initialization starts with randomly assigning alleles to genes. Then each city is visited in random order and local optimisation is performed.

Besides offering a rich supply of building blocks in the initial population, an additional advantage of this method is that local optimisation is guaranteed to be applied at least once for each city (note however that this can also be done in a postprocessing step). This can be necessary if the local optimiser does more than just resolve conflicts (overlapping labels) but also considers other, softer constraints such as position preferences which would not be considered if there was no overlap. In that case a point could be conflict free and the local optimiser would never be applied, which would result in the label possibly being placed in a position which is less preferred than another possible position. This advantage of a guarantee that a local optimiser is applied at least once will become more clear when crossover (section 4.5.1), local optimisers (section 4.5.2) and the option of focussing mixing (section 4.6) have been discussed.

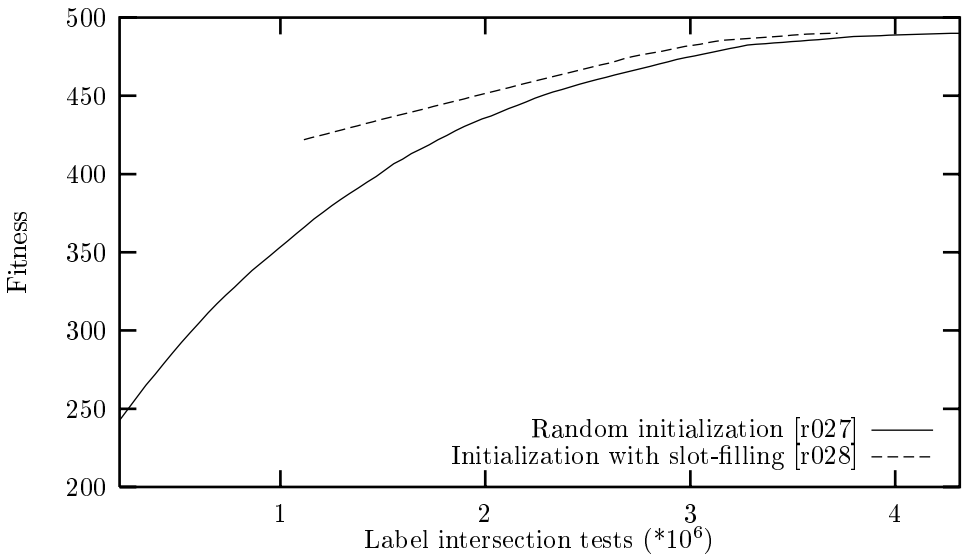

Figure 17: Difference in initialisers. The run which uses the initialiser with slot-filling spends roughly $1 * 10^6$ label intersection tests on the initialisation.

We can now propose the following initialisers:

- A random initializer: this initializer chooses a random allele for each gene.
- Initializer with slot-filling: slot-filling is a local optimiser (explained in more detail in section 4.5.2) which can locally improve a solution (for a specific city). This initializer visits each gene in a random order and performs slot-filling on it.

See figure 17 in which two graphs with the different initialisers are shown. The figure shows that both initialisers take a different route to arrive at the same number of free labels in almost the same amount of label intersection tests. The initializer with slot-filling has a slight advantage. This can be explained by the fact that the runs which were done with the random initializer also construct building blocks during the run itself and this takes time.

Another way of looking at things is in terms of *alphabet reduction*. If every city can place its label in one of four positions, the alphabet consists of four alleles and therefore has cardinality four. If we have n cities, the search space consists of 4^n distinct points. The GA has the task to find a point in that space which is good. Suppose we want to have more freedom to place our labels and increase the cardinality of the alphabet to eight. We now have a search space consisting of 8^n distinct points. Doubling the freedom of a label to be placed results in a enormous increase in the complexity of the problem. However, things are really not that bad. If local optimisers are used, the real alphabet which is processed is reduced in size. We will call this the *meta-alphabet*. A character in the meta-alphabet consists of the configuration of a city and its rivals (this means that different characters share information, but we will ignore that for now). Suppose we have a city that has four rivals. Each city can place its label in A positions, where A is the cardinality of the real alphabet. The cardinality of the meta-alphabet for that city and its rivals is therefore A^5 since there are A^5 different configurations that the city and its rivals can be in. Some of these configurations exist in optimal solutions, and many do not. Now consider the result of a local optimiser which works on the point and its rivals. For any given configuration the points are in (that is, in which positions they place their labels) the local optimiser makes a change to another configuration only if it improves the situation (results in more free labels). The result is that certain characters in the meta-alphabet (configurations the cities can be in) will disappear from the population if the optimiser is applied to that gene in all the genomes in the population. Therefore the cardinality of the meta-alphabet is reduced and the effective search space which has to be examined has been made simpler.

The fact that characters overlap complicates things a bit, but not much since only the label of the point on which the optimiser is applied can be changed by the local optimiser. This means that a configuration can not come back by changing the labels of one of the rivals (when the optimiser is applied to it at some other point during the run). In effect, only the configurations which result from appliance of local optimisers are considered by the genetic algorithm.

In [9] it was shown (using the model of the Gamblers Ruin) that for a GA which has a properly sized population of size n, an alphabet of cardinality A and a building block size of k, the following holds (all else staying equal):

$$n \sim A^k.$$

This shows that we can expect to use smaller population sizes if the alphabet which is effectively processed is smaller. Smaller population sizes allow the GA to run faster.

4.4 Fitness function

One of the main components of any GA is the fitness function. It gives every individual in the population a measure of quality and thus steers the direction of search towards desirable solutions. The choice of fitness function sometimes is trivial. For example, when using the GA as a function optimiser, the fitness function is naturally defined as the function which is to be optimised. In many other cases, the choice of the fitness function is critical to the success of the algorithm. The fitness function however is a metaphor which was taken from evolutionary biology. In a biological context an organism competes as a whole with others to survive and its fitness is necessarily a composite of many factors. The fitness metaphor has proven to be very useful for designing GA's, but it has its limits. In some problems (like the map labeling problem) there exist levels of precedence for different aspects of the problems, a situation which is radically different from the state of affairs in natural biology. Therefore, a way has to be found to make solutions adapt without necessarily measuring them as a whole. Another issue which arises because we are dealing with a geometrical problem is the difference between local and global aspects of the problem. It is

desirable to differentiate between the both in order to avoid treating local aspects of the problem in a global manner.

To make this more concrete, we start with the appropiate example of the problem of labeling a map with label selection allowed. Clearly, the most important aspect of the problem (as derived from the problem definition 2.4) is the number of free labels a map has. Now suppose that we complicate the problem a bit. Besides wanting a map with little or no conflicts, we also want that the label is placed in a specific position when possible. Also we want that cities that are capitals are labeled if possible (this means that moving or deleting labels of rivals is allowed if the rivals are not capitals). Furthermore, apart from capitals, there exist two kinds of cities with different importance. The problem now has several aspects[7] and it should be decided how they influence the form the fitness function will take. In most GA's all aspects are put together in the fitness function. This neglects the differences between aspects, which is handled by giving each aspect its own weight. This gives the new problem of tuning these weights and we will have more to say about this later.

In general, to avoid tuning a lot of weights, we would like to put as little as possible into the fitness function. We can call this the *principle of maximal delegation*: put as much as possible into the local optimiser. That way the local optimiser can produce good subsolutions which will be combined to form a good whole solution.

In solving the problem of maximising free labels for cities of different importance, with label selection, position preferences and capitals we will have to decide what should be put in the fitness function. The combinatorial most difficult aspect of the problem is of course maximising the number of free labels, so we decide to put that in the fitness function. It is necessary to put the combinatorial most difficult aspect in the fitness function because it needs a global evaluation (if it does not, a GA should not be used). Since maximising free labels is the only combinatorial difficult aspect of the problem, we delegate handling all the other aspects of the problem to the local optimiser.

In the local optimiser the precedences of the different aspects of the problem should be treated properly. For example, labeling a capital has precedence over maximising the number of free labels since we allow less free labels in order to be sure that a capital is labeled. Note that such precedence relations between different aspects is a *choice* that must be made by the one who is formalizing the problem. The local optimiser also should handle the aspect of maximising free labels, to avoid generating labelings which are bad according to the fitness function and which will therefore be selected against[8].

The aspect of preferences for label positions has the lowest precedence, since it is an aesthetical aspect only. In the local optimiser this order in precedence is followed when handling the different aspects. For example, when applying a local optimiser to a city that is not a capital, we try to move its label to a place where it is free. When there are several places, we continue with regarding the next aspect (position preferences) and choose the position which is most preferred.

We have ignored the fact that cities come in two degrees of importance so far. The set of points can be divided into sets C, P_1 and P_2, where set C contains all the capitals, set P_1 the most important cities and set P_2 the least important cities. The aspect of maximising free labels can be seen as being actually two aspects: maximising the free labels of points in P_1 and maximising the free labels of points in P_2. These two aspects have equal precedence and a weighing factor is needed to indicate how much more important $p \in P_1$ is than $q \in P_2$. This weighing factor can only be set by the user of the GA, not by the designer.

The final topic we will need to consider is the issue of label selection. It is allowed to delete labels in order to make more room for other labels. This however is not an aspect of the problem itself, but an aspect of the label placement model. The label placement model defines the search space, while the problem aspects induce constraints in that search space. Therefore no consideration

[7] For an example of a map which was labeled with label selection, position preferences, capitals and different label sizes see figure 46.

[8] There are two *adaptive mechanisms* at work here and care should be taken to let them cooperate. The two mechanisms are firstly variation (by crossover and mutation) and selection and secondly the local optimiser. More will be said about this in section 5.2.3.

of label selection was needed in the discussion above. A deleted label is just another allele like the four 'real' positions the label can be placed in. It is however better to only delete labels when it is actually needed (when the map is too crowded) instead of letting the GA filter out the deleted labels which were supplied in the initial population. For this the local optimiser provides an excellent mechanism.

Summarising, a sensible way to solve a problem with multiple levels of precedence (as often encountered in GIS-problems) is the following:

1. Divide the problem in different aspects that are as unrelated as possible.
2. Decide for each aspect whether it is combinatorial difficult and what its precedence is over the other aspects.
3. Place the combinatorial most difficult aspects in the fitness function (this should preferably be only one aspect) and delegate handling of the other aspects to the local optimiser.
4. Handle *all* aspects of the problem in the order of precedence in the local optimiser.

How should aspects be combined? Before it was said that putting different aspects in the fitness function was undesirable because it made it necessary to give each aspect a weighing factor and to tune these factors. It is however still possible to have several aspects (if they are of the same precedence) in the fitness function or the local optimiser.

It is true that combining these aspects also requires weighing factors. There is however a big difference between the weighing factors that arise out of combining different types of aspects and those that arise out of combining aspects of the same precedence. The first kind are parameters that are artifacts of the workings of the GA itself. Just because one wants to put different kinds of aspects together in the fitness function brings them to existence. The second kind of weighing factors are however directly related to the problem (like the kind that we used to indicate how much more important a city from set P_1 is than a city from set P_2). And as such the only one which can determine them is the user of the GA. Therefore, no or little tuning is necessary.

Considering all this we can come up with two possible fitness functions that can be used to solve the map labeling problem, which essentially measure the combinatorial most difficult aspect (maximising the number of free labels) in different ways:

1. Since we want to maximise the number of labels which do not intersect other labels (see definition 2.1), let's do just that: count the number of points which do not have a conflict and return it as the fitness. For a labeled set of points P, the fitness function becomes:

$$fit(P) = \sum_{p \in P} \prod_{\substack{q \in P \\ p \neq q}} (1 - conf(p, q))$$

where

$$conf(p, q) = \begin{cases} 1 & \text{when the label of } p \text{ intersects the label of } q \\ 0 & \text{otherwise} \end{cases}$$

2. We can also (conceptually) do the opposite: minimise the number of label overlap. Note that this is *not* the negated version of the previous function, since with this function an intersection with two other labels is worse than an intersection with one. See figure 18 in which an example of the difference is given. In this picture after changing the position of the bold printed label to the top left position the situation becomes worse according to the minimising fitness function (from two intersections to three) but it becomes better according the maximising fitness function (from zero free labels to one free label).

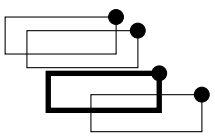

Figure 18: The difference between the first two fitness functions.

This fitness function is defined as follows:

$$fit(P) = \sum_{\substack{p,q \in P \\ p \neq q}} conf(p,q)$$

Which function is best? Since the definitions of the problem state that the number of free labels should be maximized and the second fitness function can fail to do so (as shown in figure 18), the first fitness function is preferred.

Tunability and robustness. As described above, the use of fitness functions with terms measuring different aspects has the major disadvantage of requiring a phase in which the parameters of the multiple terms are tuned. We would like to point out that this is a problem that typically arises when one designs a genetic algorithm. It is a serious problem, because it makes the algorithm significantly less robust since a new, time-consuming tuning phase may be required every time either the problem instance or the scale of the problem changes. Verner et al. use such a fitness function in their algorithm which solves the map labeling problem:

$fit(P) = a \cdot$ (Number of overlapping labels.)
$+ b \cdot$ (Total area of the overlapping labels.)
$+ c \cdot$ (Sum of the *distance factors* for points with overlapping labels.)
$+ d \cdot$ (Average label value with labels valued according to preference of position.)

They make the following remark in their article after introducing their fitness function with the tunable parameters denoted by a, b, c and d:

> The d factor in the fitness function was not used in our analysis. The various combinations of values for a, b and c were arrived at by *numerous* trial and error testing to determine the best combinations to use.
>
> ([19], emphasis added)

The time needed for solving a problem now not only depends on the problem itself, but also lots of runs are needed to tune the GA itself. Sometimes this has to be done only once. Verner et al. derive values of 1, 0, 0.0001 for a, b and c respectively from runs done on small maps. Then they look at a more dense map (a randomly generated map with 500 cities called R500) and use a value of 0.00001 for c. They say:

> Dataset R500 proved to be a very interesting problem. [..] In this example a c value of 0.00001 was used to keep the factor in line with the weight of the other parameters.
>
> ([19])

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 thematic maps for different uses), a new tuning phase is needed for every fresh problem, which dramatically increases the time needed to solve a given problem.

Another pitfall is also concerned with the scale of problems. One often tunes the GA on small problems, since these can be solved relatively quickly. The implicit assumption is then made that the same constants are also optimal for problems of larger scale, which is questionable.

These problems are typical for a lot of genetic algorithms that combine combinatorial difficult aspects with other constraints in their fitness function and it is a problem which is often overlooked. Robustness (see also section 6) is a very important property any genetic algorithm should have. We propose several strategies to increase robustness. One of them is the right use of the fitness function combined with local optimisers.

4.5 Exploration

Genetic algorithms can not work by selection alone. Selection is the mechanism that *exploits* the information present in the population (on a string level). Besides that, some amount of *exploration* is needed to make new, different strings that represent new information that can be exploited. Another way of picturing this is by visualising what is called the *fitness landscape*. The fitness landscape consists of $l + 1$ dimensions. The first l dimensions are derived from the l independent units of which the solution is build. For example, the map labeling problem for n points has n dimensions, because every city can place its label independent of every other city. It can place its label in one of four positions, so every dimension has four discrete points. The space which uses these dimensions as a basis is called the *configuration space*. The extra dimension of the fitness landscape follows from the fitness value each point in the configuration space has. For a problem with two dimensions, one gets a mountain-like landscape, with peaks, ridges, valleys and pits. Now we can get a more intuitive idea of what a genetic algorithm does. It has a population of n sample points in this space. It has therefore a limited view of what the fitness landscape looks like. Exploitation works by directing the search towards the regions which contain sample points with a high fitness value. Exploration on the other hand introduces new sample points which give new information about the landscape.

GA's therefore need operators which allow for exploration and produce new sample points in the configuration space. There are two operators which are commonly used: recombination (also called crossover) and random mutation. The effect of random mutation is choosing one of the dimensions in the configuration space and fixing all the others, and making a random jump in that dimension. We do not use random mutation in our GA for two reasons:

- The chance on improvement is very small due to the non-linearity of the problem.
- Random mutation is a blind operator, so it can degrade good solutions.

Instead of using random mutation, we will use recombination together with local optimisation. Local optimisation is a kind of mutation which uses the information of a local situation to strictly improve it. It therefore does not suffer the disadvantages of random mutation. In the next two subsections we will discuss the recombination operator and the local optimisation routines.

4.5.1 Recombination

The goal of recombination is twofold. First, it should place the good pieces of each parent together in the children. This is called *mixing*. The idea behind it is that each solution is built out of highly fit sub-solutions which have a small chance to be disrupted. These sub-solutions are called *building blocks* and the hypothesis that combining building blocks yields the desired optimum is called the building block hypothesis (see [7]).

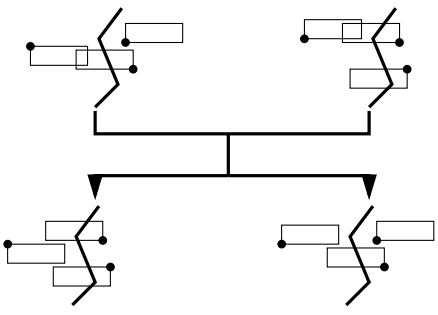

Figure 19: The bad subsolutions of the parents are recombined to produce good subsolutions in the children.

The second use of recombination is that partial, bad solutions recombined a different way produce new, good solutions. See figure 19 for an example of this. One can imagine building blocks getting formed like this, after which they get combined with other building blocks (building blocks can also already exist in the initial population, see section 4.3). The notion of a building block is certainly not mathematically precise, but it should suffice as an intuitive notion of what is going on. A building block is just some partial sub-solution which is an element of the optimal solution which the GA should find.

The theory of schemata makes talking about building blocks and strings a little easier. A *schema* is a string which has the same possible values on every location as normal strings (these values are called *fixed*), but it can also have the "don't care" value (denoted by a star: $*$). A string *matches* a

certain schema if for every location on the string either the values match, or the schema has the "don't care" value for that location. See figure 20 for a schema and some strings that match it. A schema has a fitness which is the average fitness of all the strings that match it. We can now see a building block as a schema with high fitness which is not likely to be disrupted. This means that the concept of a building block depends on both the fitness function used, and the possibly disruptive operators which are used. For most operators it is true that short schemas are less likely to be disrupted than long schemas.

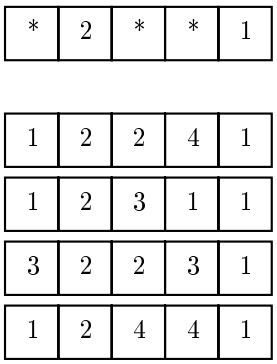

Figure 20: A schema and some matching strings.

If building blocks are so important, it becomes natural to ask what the building blocks in the solution of the map labeling problem might be and how we can ensure that the crossover operator does not disrupt them. A reasonable choice would be to consider the local solution for one point as a building block. This local solution is build out of the positioning of the label of the point (which we will sometimes call the *central* point) and it neighbours. A natural choice for which kind of neighbours we will use, is to use neighbours that are rivals in the sense of definition 2.5. In this way, only the labels that are in direct conflict are considered.

It is important to distinguish between *partitions* and building blocks. A building block is a specific partial solution (a schema) with the property that it is the best of all possible partial solutions (all schemata which have fixed values for the same positions). A partition is this set of all partial solutions. Consider a bit-string of length three where each entry can be either zero or one. We use a GA to optimise the bit-string to contain all ones (this problem is called bit-counting or one-max). Obviously, the desired building blocks in this problem are the ones, since they give a string higher fitness when set. For a specific location, the bit can be `0` or `1`. A partition P contains all possible sub-solutions for specific fixed positions. Let us assume we are interested in the partition which holds the building block for the leftmost bit: in this case $P = \{0**, 1**\}$. The building block `1**` is member of the partition. This is by definition always the case. A partition is the set of all partial sub-solutions for some specified part. A building block is a member of a partition with the highest fitness. We could by the way have chosen differently for our partition. We could have used a partition containing the substrings of length two. In that case, our partition $P^{'}$ would be: $P^{'} = \{00*, 01*, 10*, 11*\}$. The building block of this partition is `11*`. If necessary one can talk about low order and high order building blocks, depending on the number of fixed positions (the length of the substring). The schema `1**` would be a first order building block and the schema `11*` would be a second order building block. In this paper when we talk about building blocks, we mean low order building blocks.

We now have some idea about what the building block looks like (a city with its rivals in optimal configuration), and therefore a group of rivals seems to be a good choice for a partition (the fixed positions are the labels of the cities in the rival group and the "don't care" positions are the labels of all the other cities on the map). Hence, we want a crossover operator which makes use of this geometric structure. We do this by sampling *rival groups*. A rival group is simply a certain point together with its rivals. We randomly select points until the number of points in the union of the rival groups of the selected points exceeds half the total number of points. For each parent, we transfer the union of all rival groups to one of the children and the complement of that union is transferred to the other child. This results in two new children.

A useful way of doing this operation is by the use of masking, as was explained in section 4.1.2. We precompute the rivals of every point. Then we precompute a set of masks, in which each mask is the rival group of a point. When performing crossover, we sample points by randomly choosing masks and building a multi-mask (which is the inclusive OR of the masks). We keep sampling until the multi-mask has passed the point where more bits are set than there are cleared[9]. This

[9] Actually, testing if half of the multi-mask is set every time a sample is taken would be an expensive (time-consuming) operation. Instead the number of masks is fixed and the multi-mask is build by taking a fixed number

multi-mask serves as a crossover mask, because every entry in the mask specifies if we should take the label positioning for the point from the first or the second parent.

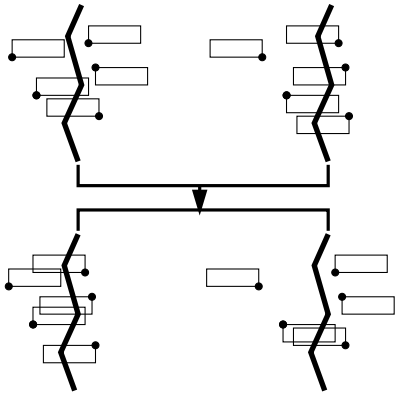

Figure 21: Crossover generates new conflicts.

A valid objection at this point to this choice of crossover would be that the crossover operator is still to likely to be disruptive. Consider for example figure 21. This is exactly the opposite of what we want the crossover operator to do, since it now breaks up useful building blocks. This situation can be remedied as follows. After we perform crossover, we build a set of points that are on the border between the points from the different parents. Each point in that set has a rival which came from the other parent. Only for these points new conflicts can have arisen. For all these points we check if there is a conflict, and apply a local optimiser on the point if there is (we call this *repairing*). This reduces the disruption of the crossover to a minimum, while being sufficiently fine grained to transfer building blocks separately (this prevents 'hitchhiking', which is the phenomenon that bad partial solutions get transferred with good building blocks and win over (better) alternatives because of the superiority of the building block). See figure 22 for a comparison of this crossover operator with other possibilities, such as uniform crossover, one point crossover and a crossover which swaps subtrees of a precomputed kd-tree[10]. The need of sampling multiple masks can be seen from this picture since separate runs were done (for the treeSwap crossover) with the number of masks fixed at one and the number of masks variable.

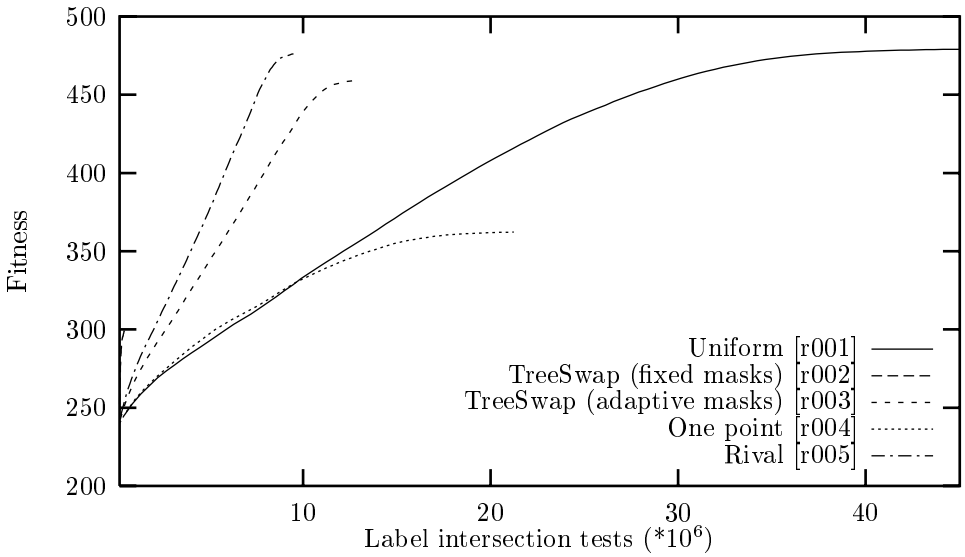

Figure 22: Comparison of different crossover operators.

The next section will discuss local optimisers which can be used. Note however that this kind of repairing is not essential to make the GA work. If the population is large enough, there will be enough variation from which to build the optimal solution. Also, the elitist recombination scheme

of samples. After the multi-mask is build the number of bits set in the multi-mask is counted. If it is less than the length the number of samples is increased for the next crossover, and if it is more the number is decreased.

[10] A kd-tree is a geometric data structure which recursively splits the set of points in half with a hyperplane (a line in two dimensions), which results in a tree with the points in the leafs. Since points in the same subtree tend to be close together, swapping subtrees might seem like a good crossover operator.

will not allow degraded solutions to replace their parents so degraded solutions will not make it into the population. We would however like to have as small a population as needed (because large populations take longer to converge) and as many successful recombinations as possible. In figure 23 the effects of using the repair function are seen. A run which does not use the repair function gives an obvious inferior result compared to a run with the same settings but in which repairing is permitted. Doubling the population size gives a better end result, but at high computational cost.

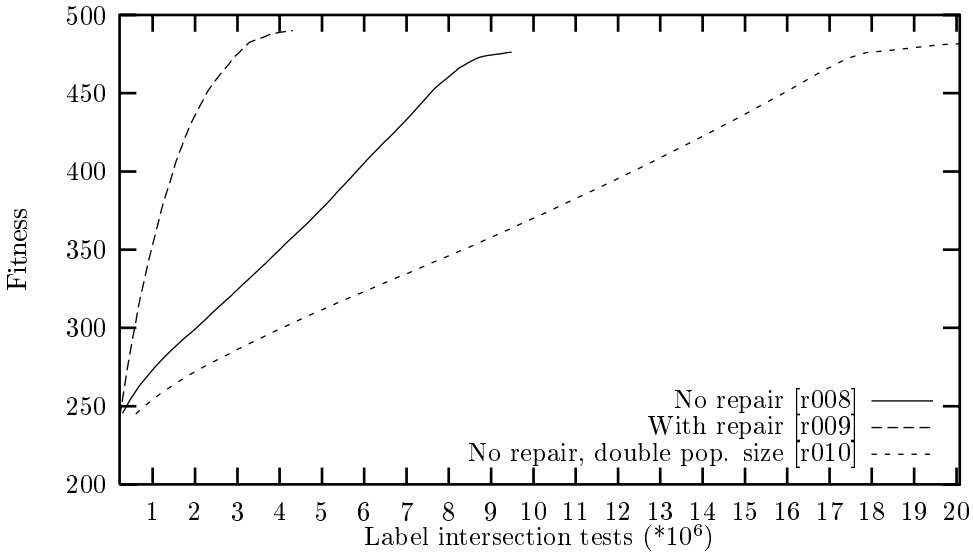

Figure 23: The effect of repairing conflicts.

Besides their use in repairing, there is also another reason why the use of local optimisations is preferred, which will be explained in the next section.

4.5.2 Dealing with interaction

As explained in the previous section, local optimisers are useful to resolve conflicts that arise during crossover. In fact the use of local optimisers can be put in a broader perspective than just the repairing of the border of the crossover. The way a GA works is by looking at the population of sample points of the configuration space and using this global view how the fitness landscape at those sample points looks like to decide where the global optimum is. So we can use local optimisers to locally construct good partial solutions, and rely on the GA to combine the local solutions into a global solution. This is important because a local optimiser now does not have to worry about the interactions that are some distance away, but which it does influence. It can forget about everything which does not fall in his scope of view and try to make the best it can.

However, a local optimiser should worry about producing enough alternatives. If the variation in local solutions is low, the global optimiser does not have enough choice to decide which local optimum was best in the overall picture. Ideally, a local optimiser produces a set of alternative solutions and chooses randomly from them. Of course, a certain amount of variation is produced just by the fact that the local optimiser will not always be applied to exactly the same situation, which might be enough.

Local optimisers get used to solve two kinds of conflicts:

- Old conflicts which arose during the initialization of the first population.
- New conflicts that arise on the border between parts from different parents during crossover.

The new conflicts are all in the border, so we can solve them by applying local optimisers on the cities in the border. The old conflicts can occur everywhere on the map. That means we want to apply a local optimiser on every point with the same probability. This can be done during sampling of central points when building the crossover mask. Instead of altering the children, we try to improve the parents. Each point on the map has equal probability to be chosen as a sample. To this point we apply the local optimiser. Note that we can use a 'focus' (see section 4.6) to guide the GA to regions with conflicts to limit the computational cost.

We can now consider some local optimisers for the problem of optimising the labeling of a point relative to the labelings of his rivals. Note that for these optimisers, the scope is very limited: it consists of just one point. The optimiser therefore can only reposition the label of the central point. Consider the following local optimisers:

- Random repositioning: just randomly choose another position for the label and place it there. If the map is not too crowded, chances are high that it will solve the conflict. The repositioning will result in a change in the fitness function and the repositioning will be made undone when the fitness decreases. Advantages of this approach are that it is independent of the fitness function and preserves variety. A disadvantage is that it is blind, it does not really use the information about the positions of the labels of the rivals.

- Slot-filling: consider every possible position for a label to be a *slot* and call a slot filled when a label of one of the rivals intersects the label of the central point if it was positioned there (see figure 24). Now determine of all slots whether they are filled or not, and choose from the free slots in order of preference if positions have different preferences, otherwise choose randomly. When no free slots are available, leave the label where it is, or delete it if label selection is allowed. An advantage of this approach is that it will solve the conflict if possible. A disadvantage is that it makes implicit assumptions about the fitness function because it has to consider what the fitness function counts as a conflict in order to solve it.

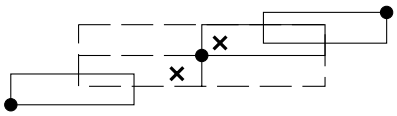

Figure 24: Slotfilling: determine free slots and choose from them.

A GA uses adaptive mechanisms (see also section 4.4) to adapt solutions to become more like the desired solution. Every GA uses the adaptive mechanism of variation (as provided by crossover and mutation) combined with selection. Local optimisers can be used as another adaptive mechanism, by letting them change solutions as they see fit. The slot-filling local optimiser is like that. It is however possible to make a local optimiser which only provides variation and uses the fitness function to check if it has become better, which is what the random repositioning optimiser does.

If one decides to make the local optimiser a new adaptive mechanism in its own right, care should be taken that it operates in concert with the other adaptive mechanism (variation and selection). We want to avoid adapting solutions which are better according to one mechanism and worse according to the other. In practice this means that all aspects of the problem should be handled in the local optimiser, including the aspect which is measured in the fitness function (which should be the combinatorial most difficult aspects, as argued in section 4.4). Also, all aspects should be handled in order of precedence. This means for example that if label positions have preferences they are only considered when there is a choice between two positions which result in an equal number of free labels.

An 'intelligent' local optimiser (such as slot-filling) has several uses in the GA. Summarising:

- It is used to repair the border.

- It constructs locally good solutions from which the GA can construct a globally good solution.

- It allows for the use of the maximal delegation principle which removes aspects of the problem from the fitness function which are not combinatorial difficult.

- It allows for the handling of precedences in GIS-problems.

Clearly, only the first (and in some sense the second) point holds for the random repositioning optimiser. It is for these reasons that we think slot-filling is the most preferred optimiser. Experimental results confirm this. See for example figure 25 where the GA was run with respectively the slot-filling optimiser and the random repositioning optimiser. Both converge to the same optimum, but the use of slot-filling makes the speed of convergence much higher. In this case, plugging in a tailor made optimiser allows the GA to run faster without loss of quality. Indeed, if the algorithm is run without the use of a local optimiser but with the same population size (also shown in the picture), performance is significantly inferior.

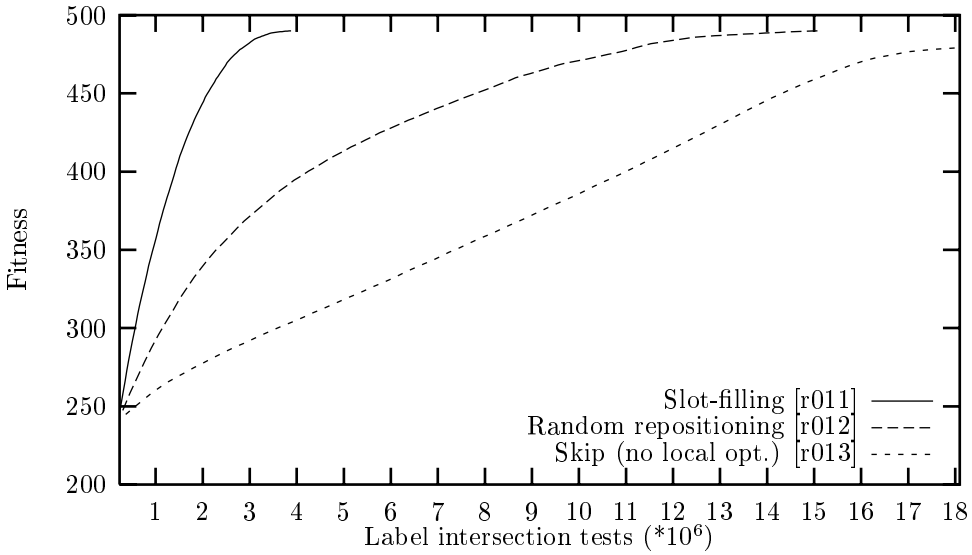

Figure 25: Three runs of the GA with different optimisers .

Slot-filling is essentially a local optimiser which has to resolve conflicts resulting from label overlap. It was however no trouble to augment it to handle preferred positions and automatic label selection as well. This confirms the statement that combining tailor-made local optimisers with the powerful search mechanism of the GA is a good way to solve the map labeling problem. See figure 26 for a dense map of two hundred cities which was labeled using the slot-filling local optimiser where preferences were followed and label selection was allowed. Note that it is important what counts as an intersection. For this picture a label is considered to intersect when it overlaps either another label or another point. Other users would choose differently. For example, if points with deleted labels are also deleted an intersection would be defined as overlap with another label.

4.6 Focusing on conflicts only

During the run of the genetic algorithm, conflicts will be solved. However, even when a conflict is solved for a certain point, it will still be active in the process of crossover, mixing and local optimisation. This could be a waste of time if the conflict was solved satisfactory. To speed things up, a way can be constructed to avoid spending too much time on parts of the solution which are good. We can focus on unsolved conflicts only by keeping track of which conflicts were solved and excluding them from crossover (more accurately, from mixing). This is easily done using the technique of masking. Each solution carries a *conflict mask* in which a bit is set if the conflict for that point in the solution is unsolved. When performing crossover we again use the masking technique. Masks are sampled (by picking points on the map) and each mask corresponds with

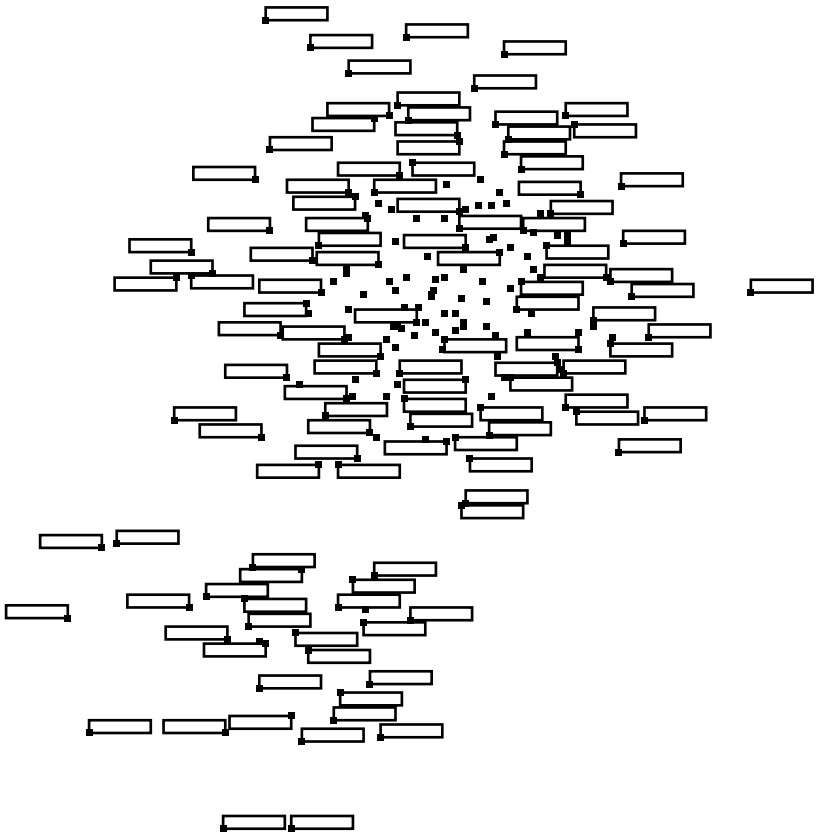

Figure 26: A map of twohundred cities which was labeled with preferences and automatic label selection.

a rival group (a bit is set if a city is member of the rival group). We now make sure only rival groups are sampled which contain at least one point which has a conflict or whose label is deleted (if label selection is allowed). This is not difficult to do, since the conflict mask of the solution contains the information we need to test a whole rival group for conflicts. Effectively, this will focus the activity of the GA on the regions which apparently are difficult. Note that the union of the sampled rival groups should now be half of the focused region, instead of half of the whole map. These ideas become more clear by looking at figure 27. The picture shows a map with a region which has no conflicts, and a region which contains only conflicts. Normally a map would of course not have such neatly divided regions, but this is just to make the point. In the picture on the left, the map is divided in two parts which are complementary. When a child is generated, one of the parts is taken from the mother and the other from the father. This division is made without focusing. In the picture on the right, we do focus on regions with conflicts. We now only sample in the region which contains conflicts. The complementary parts now are not equal in size anymore, which is the key for the speed-up since the border between the part from the father and the part from the mother is not so large anymore.

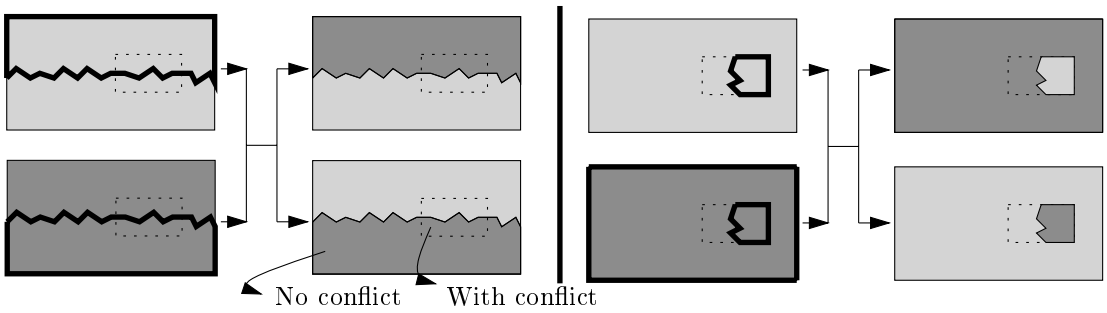

Figure 27: Crossover without (on the left) and with (on the right) focus.

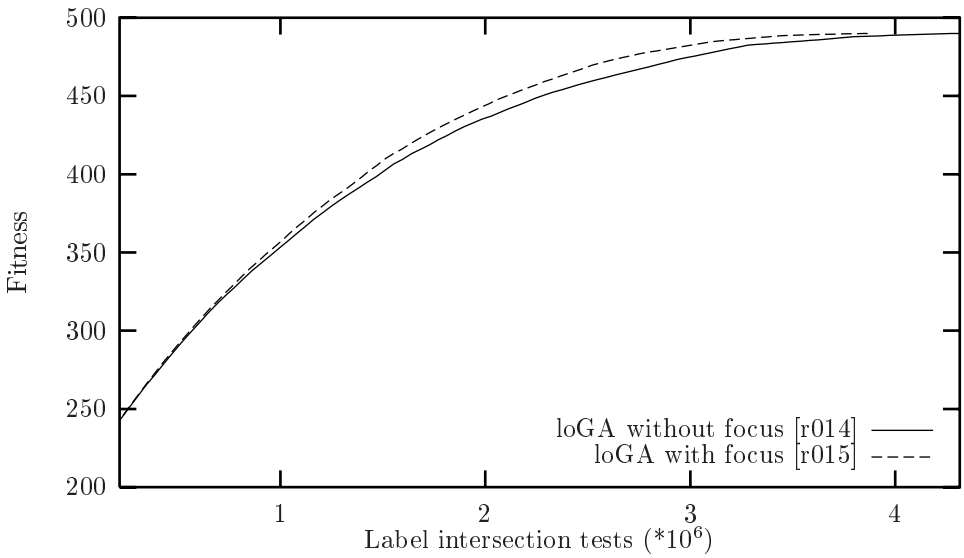

Figure 28: The difference in speed with focus turned on or off.

Figures 28 and 29 show that the difference in speed with focus turned on or off is not very dramatic, since the gain comes from the pieces of the map which are solved in both parents, for

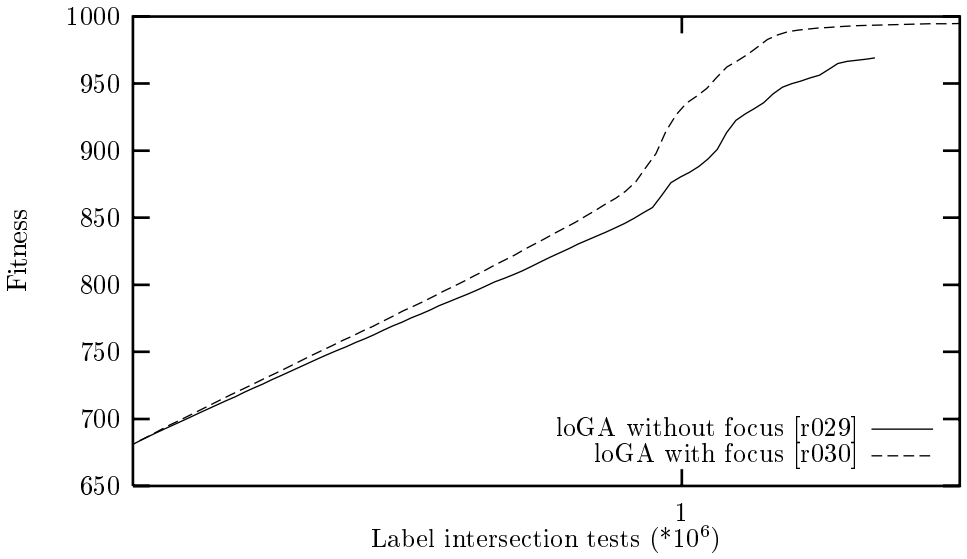

Figure 29: The difference in speed with focus turned on or off for a large map. The map has width 1582, height 1222 and has 1000 cities placed on it.

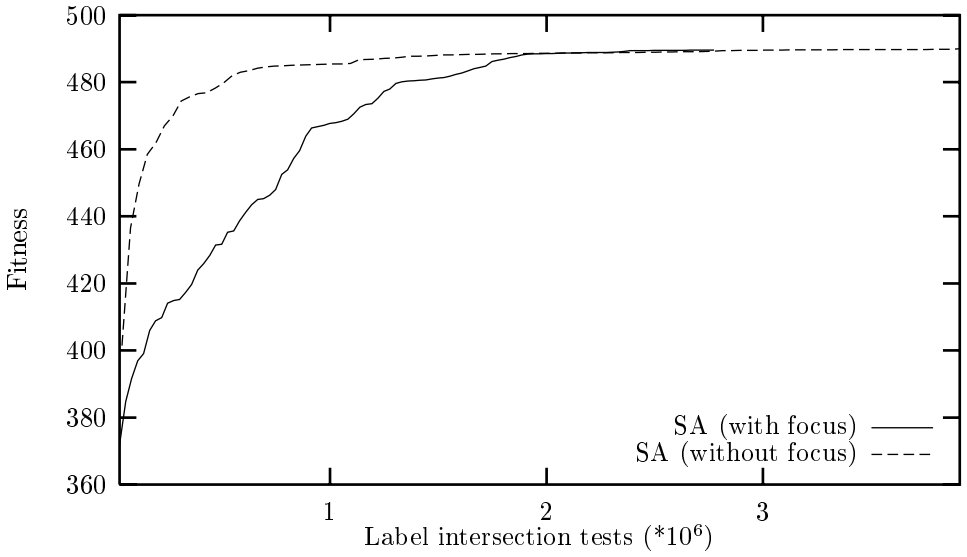

Figure 30: The difference in speed with focus turned on or off for the SA.

a whole rival group. This is seldom the case. However, focus does not have any bad side-effects so it can not hurt to use it. Also, possibly ways can be devised to make the focus more effective, but research on that topic is still in progress. The focus option is not only applicable for Genetic Algorithms, but also for the Simulated Annealing algorithm and the lazy hillclimber. See figure 30 for a run of the SA with and without focus. Again no deterioration of quality is observed. The genetic algorithm uses its conflict mask for fast fitness evaluations. Using the focus option is a bonus which can be used at no extra cost. For the SA, the conflict mask has to be updated during the run which is not necessary if focus is not used. This extra bookkeeping costs time (in the form of label intersection tests) which can be seen in the figure by the different rate of improvement. However, since the SA is forced to finish its schedule in order to be sure it found the best solution, the run with focus wins out in the long run (no pun intended).

4.7 Design of the loGA — summary

Different aspects of the loGA were investigated in the preceding sections. In this section we recapitulate and give the design choices which should be made for the loGA.

The selection scheme should be the elitist recombination scheme. This allows the probability of crossover to be 1.0 and the probability of mutation to be 0.0.

The initialiser can be both random or with the use of slotfilling. This mainly depends on the task at hand. If all one wants to do is label a map and maximise free labels, simpler settings can be used than when one also wants to consider preferred positions, automatic label selection, etc. In the case of just making the map conflict free, the random initialisers should be used because it gives the most variation. If one wants to consider more rules, the other initialiser should be used since it guarantees each city undergoes local optimisation even when it is not engaged in a conflict.

The fitness function used should be the fitness function which maximises the number of free labels, for the reasons mentioned in section 4.4.

The crossover operator should be the rival based crossover, in cooperation with local optimisers. Again, depending on the task at hand a choice has to be made whether local optimisation should only be performed on the border or on the parents also (see section 4.5.2). The local optimiser used should be slot-filling.

Focus should always be turned on, since it speeds up the algorithm and has no disadvantages.

5 Comparison experiments

As was argued in section 3, stochastic algorithms seem to work best at solving the map labeling problem. Several algorithms have proven to work well, and we will compare our algorithm with them. These algorithms are the simulated annealing algorithm (from [2]), the genetic algorithm from Verner et al. (from [19]) and the hill climber algorithm. We will present the comparison results in two subsections: in section 5.2 we will compare the quality of the solutions of the different algorithms, and in section 5.3 we will explore the issue of how fast the algorithms find their solutions. First however, we devote section 5.1 to implementation details. Finally, in section 5.4 we will give some conclusions that we can derive from these experiments.

All comparisons need some fair method of measurement for computational effort. When comparing two different genetic algorithms, counting the number of fitness evaluations is a fair method. However, this would give an unfair advantage to the genetic algorithm when it is compared to the simulated annealing algorithm or the hill climber, since those algorithms do not need a full fitness evaluation of the whole map. Instead the most atomic action that all algorithms perform is the test if two labels intersect each other. Therefore, all graphs show the number of label intersection tests against the solution quality (or fitness) which was obtained after that many tests[11].

[11] The correlation coefficient for the correlation between the total number of label intersection tests and the total amount of time needed was always higher than 0.984 for every experiment.

map density	100	150	200	250	300	350	400	450	500	750	1000	1500
loGA pop. size	150	150	150	150	150	150	150	150	150	300	500	1000
copyGA pop. size	200	250	250	250	400	400	400	400	400	400	500	1500

Table 1: The population sizes used for various map densities.

Since we will compare our own genetic algorithm against the genetic algorithm of Verner et al., we will refer to our own algorithm as the local optimiser GA (loGA) and sometimes refer to their algorithm as the copyGA (since the power of its crossover operator lies in copying good parts to both children).

The maps which the algorithms have to label are derived from randomly generated datasets. These datasets are alike to the maps Christensen et al. used when they performed their comparison experiments mentioned in section 3 (since they are random, they are unlikely to be the same). On a grid of 792 by 612 units n points are randomly placed. Each point can place its label in several fixed positions, as depicted in figures 2 and 3. Unless otherwise specified, a four-position placement model is used. Each label has fixed dimensions of 30 by 7 units. Increasing the number of points therefore causes the amount of interaction to rise and the problem becomes harder.

The population sizes for the loGA and the copyGA are shown in table 1 for different map densities[12]. The population sizes of the copyGA are the same as those which Verner et al. used. Verner et al. did not experiment with problems with 1500 cities on the map, so we choose a population size for those problems which is certainly large enough. The reader is warned not to conclude anything from the population sizes used. We do not claim these values are optimal and further research in this important area is definitely needed.

5.1 Implementation

5.1.1 The lazy hillclimber.

The hill climber algorithm excels in simplicity. It simply tries to change the solution (reposition one label) and reverses the change if the solution degrades in quality (according to the fitness function, so a degradation means less free labels in this context). We call this hillclimber a 'lazy' hillclimber since it invests as little time as possible in deciding which change will be best. Contrast this with an 'eager' hillclimber which picks the best change of all possible, allowable changes (this is like the gradient descent algorithm discussed in section 3).

The hill climber was implemented as a reduced version of a genetic algorithm. The population size was one, and no crossover was performed. Mutation was always performed, and this allowed the solution to improve. Since mutation can also degrade solutions, the mutation was implemented as a local optimiser which checks the solution after the change to see if it degraded. If it did, the change is made undone. The check is done by comparing the fitness of the changed solution with the original fitness. However, since this change was local (consisting of only one point) the change in fitness could be derived by looking at the point and its rivals. This makes the computational cost of a single iteration (consisting of one mutation) very small. Of course, many iterations were needed to find a good solution.

5.1.2 The simulated annealing algorithm.

The simulated annealing algorithm was implemented according to the directions of Christensen et al. in [2]. The reader is referred to that article for more details. After implementing the basic algorithms, several changes were tried. It was found that the algorithm performed better when 'focussing' (see section 4.6) was turned on. In the original algorithm, a change could happen to every point which was given as input. With focus turned on, a change can not happen to a point if its label is free and the labels of its rivals are free also. Another version used the slot-filling procedure (see section 4.5.2) instead of making a random change.

[12] See section 5.1.4 for an additional experiment with constant population size.

5.1.3 The copyGA.

The algorithm of Verner et al. basically consists of a standard GA (using a generational model with roulette wheel selection, a random mutator and a fitness function containing several aspects) with a new crossover operator. The crossover operator makes use of masking to preserve good subsolutions. Each string (a solution consisting of an array with label positions for each point) has a mask associated with it which specifies if the point at that location 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 neighbour of a point with an intersecting label, otherwise the labeling is considered bad. A neighbour of a point p is defined in the article of Verner et al. as one of the four points that are closest to p (using the Euclidean metric). This contrasts with the use of the definition of a neighbour as a rival (definition 2.5) which was used in the rest of this article.

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 31: The use of masks with the mask crossover.

Crossover is performed by looking at a random bit mask and the two masks of the parents. For every location the following reasoning is applied to decide whether to copy the contents for the location from the first or the second parent (see figure 31 for a summary of this procedure). 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 for one parent and bad for the other parent, both children get the information from the same parent.

Crossover operators are usually complementary: the children can be split up in two pieces each which can be joined to form the parents again. The masking crossover is different in that it can copy more from one of the parents than from the other one. This results in a very exploiting scheme since good subsolutions immediately are distributed to both the children. The risk of such an exploiting scheme is that premature convergence could become a problem.

As we said before in section 3, besides the crossover operator the algorithm is fairly standard using a generational scheme with roulette wheel selection, blind mutation, elitism (the best individual of the population always makes it into the next population), and a fitness function measuring different aspects of the problem. We felt that the basic idea of their crossover operator was interesting but that the other (standard) choice could be improved on by adopting the choices we made for our own algorithm: elitist recombination, no blind mutation and a fitness function which measures only the number of free labels (Verner et al. also included two terms (with parameters b and c) to minimise the area of label overlap which is unnecessary for the comparison experiments). This resulted in an algorithm we denote by 'copy Xover ERGA' in our presentation of the results.

Also we experimented with the use of the neighbours when copying. When we say 'no neighbours' we mean that a labeling for a point is considered good when the label does not intersect another label.

5.1.4 The loGA.

The local optimiser GA was implemented following the recommendations of section 4.7. In addition, care has been taken to allow the algorithm to run as fast as possible. To achieve this every genome has a *conflict mask* as an attribute. The conflict mask specifies for each point whether the label of the point is free or not. This allows the fitness function to be computed quite fast, since no time consuming label intersection tests have to be performed. This conflict mask needed to be computed for every genome at the start of the run and stay updated during the run. Therefore, after crossover was performed, on points where new conflicts can have arisen the conflict masks are updated. The result was that fitness evaluation did not have to cover the whole map and the GA runs faster[13].

The focus option uses the conflict mask to determine which points can be sampled (as explained in section 4.5.1). The larger the map, the more the GA will benefit from using the focus option.

Locals optimiser were only applied if a test indicated that the point had a conflict.

The population size of the loGA was chosen such that the quality was comparable to the SA of Christensen et al. Note however that since the most difficult building block oversizes the population size for the other building blocks (as explained in section 2.1) we could use a smaller population size and sacrifice little quality. In fact, since building block gets bigger when the problem instance gets bigger, the amount of characters in the meta-alphabet (as described in section 4.3) stays about the same. To demonstrate this we have also done experiments with the loGA run with a population size of 150 for all maps, which is always clearly indicated.

5.2 Quality

5.2.1 The lazy hill climber versus the loGA

An option to solve the map labeling problem could be to simply let the hill climber run from different random initialisations, and pick the best solution. If the fitness landscape is rough, this method would depend on choosing in one of the runs the right hill with the global optimum (by accident), after which it is climbed. However, for label placement with four positions which are all equally good and no label selection, the fitness landscape is not rough. So hill climbing actually works quite well. It makes rapid progress at the start, but this slows down when the hard pieces of the map have to get solved (which consist of finding a way out of a plateau). Then it takes a long time to solve these problems.

Since the lazy hillclimber is such a simple algorithm, it can serve to give an indication of the point where the problem gets tough. If one devises a new algorithm, one would at the very least want it to give better results than the lazy hillclimber, which does not use any sophistocated techniques at all.

In figure 32 several runs are plotted (for the loGA, the best available solution in the population at any given moment is used for comparison). The hillclimber (hillclimber 1) finds 482 free labels (averaged over five maps). Also plotted is a run of the GA, to compare how much time it would take to gain the same quality (number of free labels). Note that this run uses a very small population size (using a smaller population size would degrade the quality further however). Also plotted is a run with a close to optimal population size for the best quality the loGA is able to find (490 free labels). A run where the hillclimber used the focus option (from section 4.6) showed worse performance, so focus does not work well with the lazy hillclimber.

Experiments suggest that the hill climber gets trapped in a local optimum at the end of the run. In figure 34 the difference in solutions (taken from actual solutions that were found) from both algorithms is seen.

[13] This optimisation is more difficult to do with the algorithm of Verner et al. since the conflict mask would then have to be updated to represent the number of overlapping labels instead of simply whether there is a conflict. The "copy Xover GA" does have the optimisation.

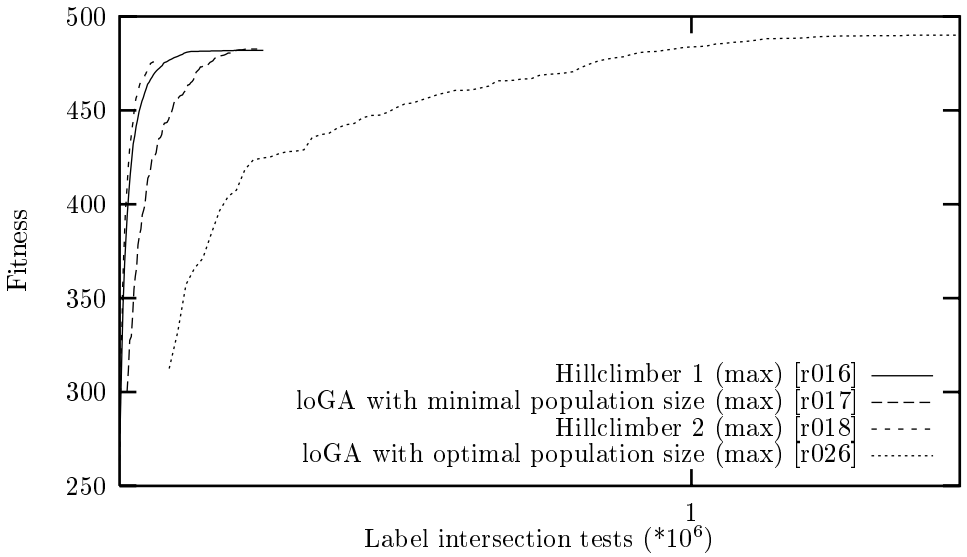

Figure 32: A comparison of the loGA with the lazy hill climber. (Hillclimber 1 uses focus and random positioning and hillclimber 2 uses no focus and random positioning.)

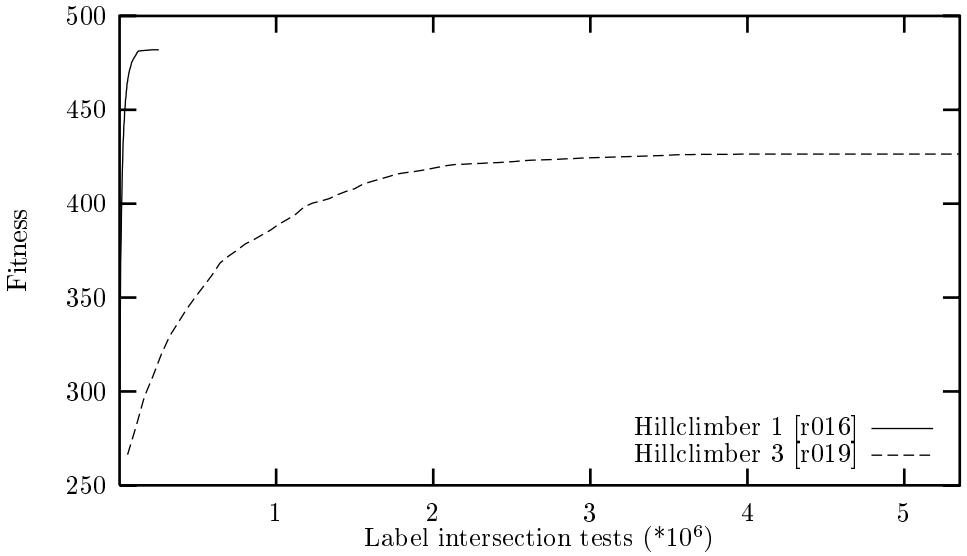

Figure 33: A comparison of the lazy hill climber with different mutators. (Hillclimber 3 uses no focus and slot-filling.)

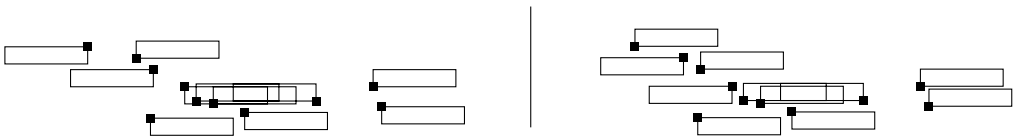

Figure 34: The difference between solutions from the hill climber (on the left) and the loGA (on the right).

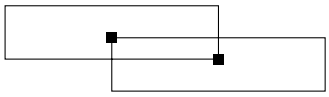

Figure 35: Slot-filling can fail.

Shown in figure 33 are runs of the hillclimber with a focus option on (see section 4.6 in which the same mechanism as used in the loGA is described) and with slot-filling used as the mutator. The run with slot-filling shows very poor performance which is as expected because slot-filling can not solve problems like the one shown in figure 35 since it can not find any free slots if either point is chosen as the central point.

A comparison of implementations of the lazy hillclimber and the loGA for different map sizes was done and the results are shown in figure 36.

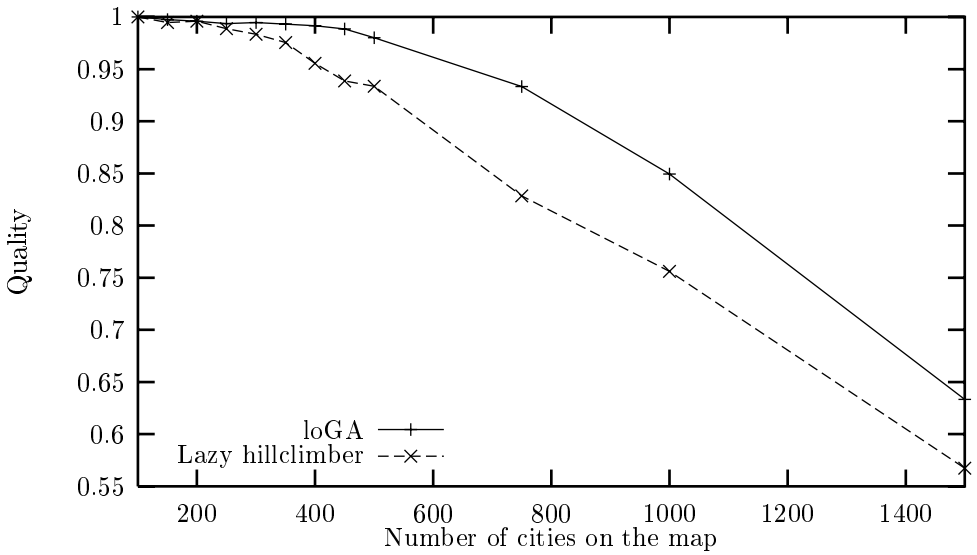

Figure 36: The lazy hillclimber compared with the loGA. See appendix A for the exact datapoints with standard deviation.

5.2.2 The loGA versus the simulated annealing algorithm

It was shown in [2] that in terms of solution quality simulated annealing works very well. Indeed, it performed better than all the other algorithms tested in that article. We did an experiment in which an implementation (according to the description from the original article) of the simulated annealing algorithm was compared against the loGA. The results of this experiment are plotted in figure 37.

Three versions of the simulated annealing algorithm were compared against the loGA. They all had the same annealing schedule. The first version is the one that is the most faithful implementation of the directions from the article of Christensen et al. Since the loGA gains much of its power from the local optimiser, for completeness sake it was tried with the simulated annealing algorithm also. This gives poor performance, as could be expected since the local optimiser strictly improves the situation and therefore lacks a method to escape a local optimum by generating an inferior solution. The third version has the focusing capabilities that were originally developed for the loGA turned on. This had as a result that the performance was increased, since the algorithm did not have to waste iterations improving already good labelings. Also the speed of the algorithm was increased.

Interesting is that both the simulated annealing algorithm with focus and the loGA produce exactly the same solutions for the low density maps. Since these results are averages from five randomly generated maps, this leads us to suspect that these solutions are optimal. The quality of

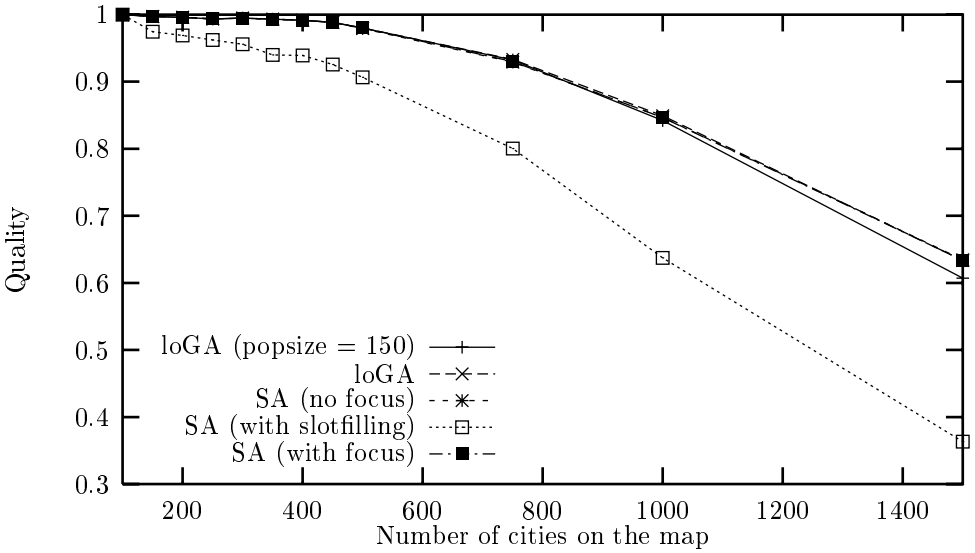

Figure 37: The simulated annealing algorithm compared with the loGA. See appendix A for the exact datapoints with standard deviation.

the results of both algorithms are very similar and it can not really be said that one is significantly better than the other (see however section 5.3 for a discussion on the running times).

In another experiment we changed the problem to the one described in definition 2.4, where the placement model also contains the possibility of entirely deleting a point and its label from the map. This way more labels can be placed since unresolvable conflicts can be solved by deleting one or more of the labels. This adds an extra level of complexity to the problem. In figure 38 three runs are plotted: the loGA which uses the slot-filling initialization described in section 4.3 and optimises parents, the SA with and the SA without focus. We used the same cooling-schedule for the SA as before (which is the same as Christensen et al. recommend). Although the loGA seems to perform better one should take into account that the cooling schedule could have been lowered more slowly since the SA takes less time to converge.

We were also curious to see how both algorithms would perform if they used the eight position placement model, since this results in an enormous increase in the search space The results are plotted in figure 39. Again we see that the loGA starts performing better than the SA when the problem complexity increases. Here also we have to consider that the cooling schedule of the SA could be extended, which would result in better results. I seems to be however that the loGA is very robust since changing the problem instance still produces acceptable results, whereas the SA would have its cooling schedule to be tuned for every problem instance. Also we suspect that there is a boundary where the problem instance becomes so complex that the SA becomes computationally infeasible while the loGA still produces good results in reasonable time. This is because the loGA uses its local optimisers to transform the alphabet into a much simpler meta-alphabet (see section 4.3) which reduces the size of the search space drastically. The SA still has the whole search space to cope with. Since real map labeling problems include many rules and constraints, this boundary is likely to be crossed relatively soon. In other words, map labeling problems become very complex very fast and an algorithm like the loGA which reduces the search space seems to be the best approach for solving them.

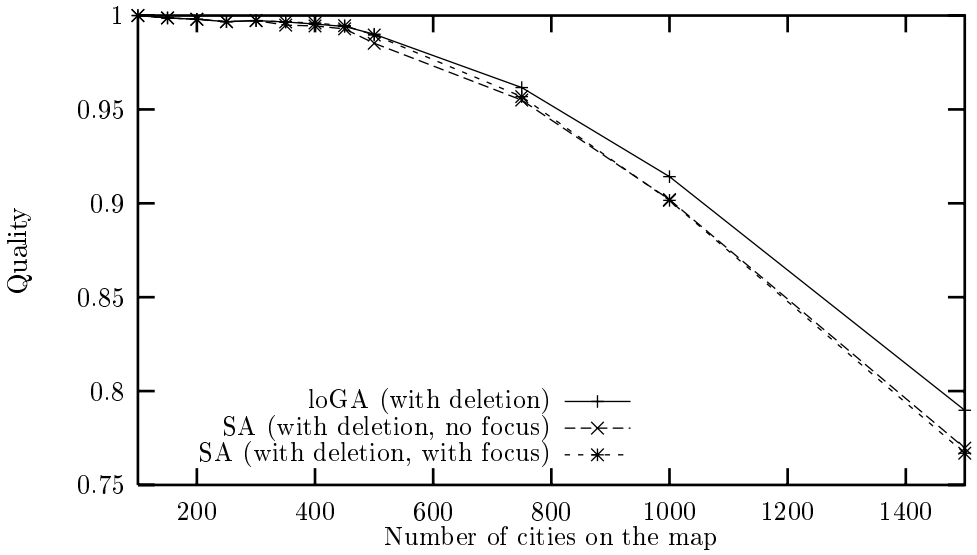

Figure 38: The simulated annealing algorithm compared with the loGA for the problem with label selection. See appendix A for the exact datapoints with standard deviation.

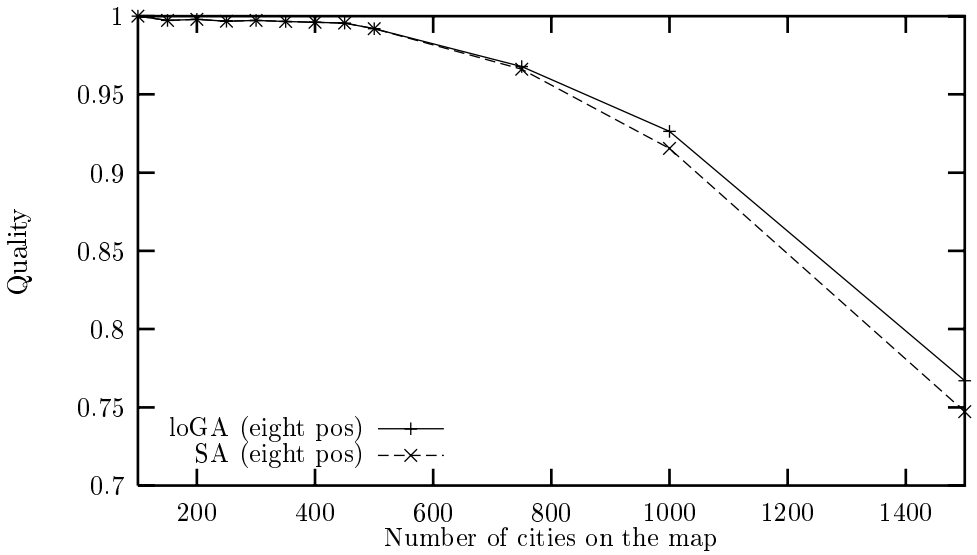

Figure 39: The simulated annealing algorithm compared with the loGA using an eight position model. See appendix A for the exact datapoints with standard deviation.

5.2.3 The loGA versus the copyGA

How does the GA of Verner et al. compare against the loGA? In figure 40 the results for the randomly generated datasets are plotted. Note that these results contrast with the results given in the original article of Verner et al. Unfortunately we were unable to reproduce the results of Verner et al, and figure 40 shows the results that our own implementation of the algorithm produced. Note that there even is a point where (our implementation of) the algorithm of Verner et al. starts performing worse than the hillclimber, a much simpler algorithm. We also plotted a run of the GA which used only the crossover operator of Verner et al. (see section 5.1.3). This produced good results, but could not top the loGA.

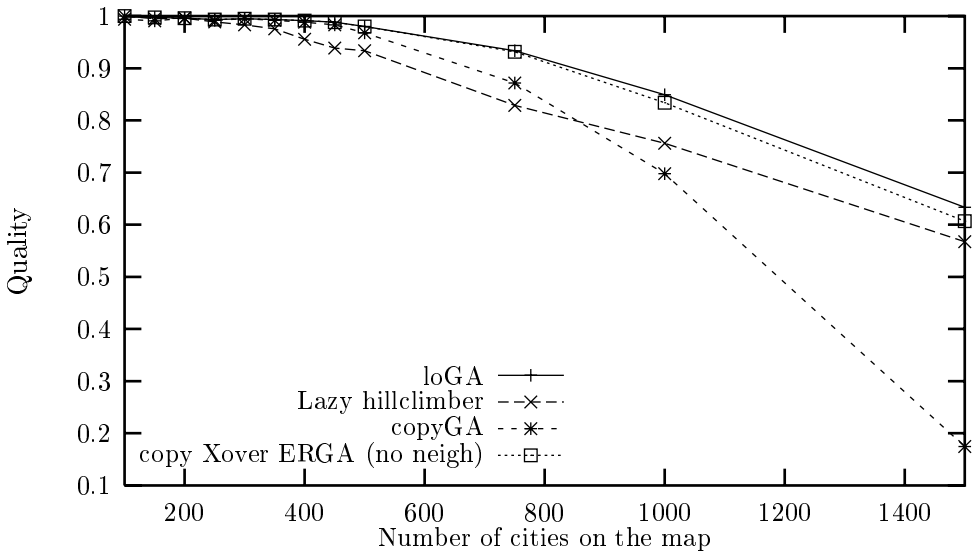

Figure 40: The loGA compared against the GA of Verner et al.. See appendix A for the exact datapoints with standard deviation.

We did try several approaches to find out why our implementation produced different results. In figure 41 several runs are plotted together with the data from [19]. These runs were all done in the eight-position model (a label can be placed in any of eight possible positions, according to definition 2.3) since this model was used also in their article to compare their results with the simulated annealing algorithm (which, however, used a four-position model).

From the experiments which are visualised in figure 41 we can conclude the following:

- The position model was not the cause of the discrepancy, since the run which uses the eight-position model still performs worse than Verner et al. reported in their article.
- Curiously enough, the run which did not consider neighbours (see 5.1.3) performed comparable with the reported results.
- The loGA performs better than the reported results.

Besides failing to produce better results than the loGA, there is also the important question of how easily the copyGA can be extended to other problem instances. We will argue that this is difficult, but we will have to examine the workings of the GA closer before understanding why.

A genetic algorithm is an adaptive process: it applies adaptive mechanisms to adapt solutions to become more like the desired solution. In the conventional GA, the only adaptive mechanism is provided by the combination of variation (as induced by initialisation, crossover and mutation) and selection. This is often called the exploration/exploitation mechanism. With this mechanism,

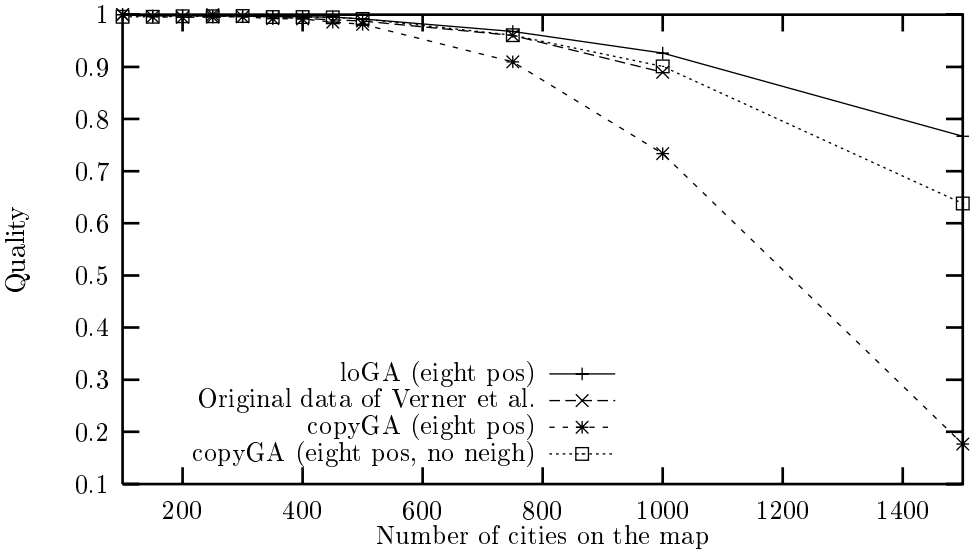

Figure 41: Several approaches of the copyGA compared with original results. See appendix A for the exact datapoints with standard deviation.

given two random parent strings, crossover is equally likely to make a child which is better as to make a child which is worse. However, it is possible to put a bias in the crossover operator so that it is more likely to generate good children. This introduces a new adaptive mechanism. This is exactly what the crossover with masking does (see figure 42 in which the influence of the adaptiveness of selection is removed). The crossover from the loGA which swaps rival groups and repairs using local optimisers (called the rival crossover from now on) also puts adaptive powers in the crossover operator (because it uses local optimisers). There is nothing wrong with making the genetic algorithm adaptive in this way. However, it is a design decision which has several consequences.

Care must be taken so that the two adaptive mechanisms (from variety/selection and from adaptive crossover) do not act against each other, thereby crippling the genetic algorithm. In a way, the GA has two drivers at the wheel since both mechanisms could drive the solutions in a different direction. We took care when we devised our local optimisers that they never made a local solution worse according to the fitness function. This means that repositionings (for example placing a label in a preferred position) were only allowed if the number of free labels did not decrease. The copyGA also combines the two adaptive mechanisms without opposing them.

The map labeling problem has the characteristics that it combines different aspects which can be global or local and has different precedence levels for different aspects. When dealing with such a problem, special care should be taken that each aspect is handled appropriately. As we saw in section 4.4, this could easily be done with the loGA. The copyGA fares less well. Recapitulating from section 4.4, local aspects should not belong in the fitness function since the principle of maximal delegation prescribes that the fitness function should only contain combinatorial difficult (and therefore global) aspects of the problem. Aspects of different hardness should also not be together in the fitness function since precedence levels can not, or very awkwardly, be considered then. To make this more concrete, consider for example the problem of maximising the amount of free labels combined with preferences for positions. Both are global aspects, and the conflict aspect is harder than the preference aspect (one generally does not want to have more overlap just because some extra labels can be placed in more preferred positions). Verner et al. did propose a fitness function which included a term for position preferences. Unfortunately they excluded this term in the rest of the article and did not provide any experimental results on this matter. Since

the aspects are of different hardness, this is not the preferred approach anyway.

Another option to handle more complex problems (besides changing the fitness function) is to adjust the crossover operator, since it is the only other adaptive mechanism. It is however far from clear how to do this using the technique of masking and copying. Instead we propose the use of local optimisers.

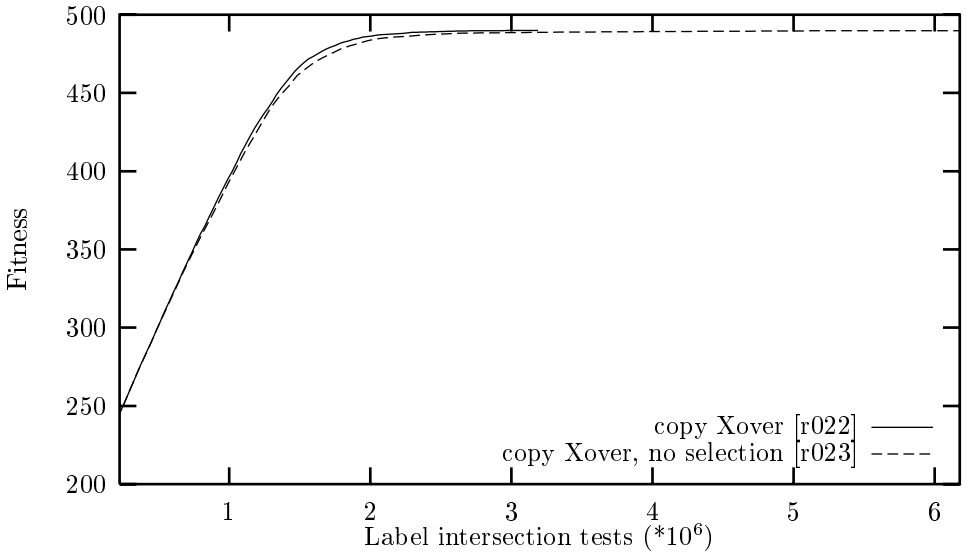

Figure 42: The copyGA without selection.

5.3 Speed

The algorithms that were used in the experiments produced results that differed in quality, and always the loGA produced better or equal results than the competing algorithm. Quality is however linked to computational effort. If an algorithm is allowed to run longer, it *may* produce better results. For a rough indication, figures 43 and 44 show how much time the algorithms used needed to produce the results which were reported.

As argued in 5.2.2, the loGA is likely to perform better (in terms of computational expense) relative to other algorithms when the *complexity* of the problem instance increases. On the other hand, as can be observed in the figures, the running time of the loGA increases drastically when the *size* of the problem instance increases. A solution for this is the use of additional techniques to keep the region of the map which is labeled small. Research in that area still has to be done.

For a different perspective on how the algorithms find their solutions, the runs of the four algorithms are plotted for the maps with 750 cities.

5.4 Concluding observations

We can from these experiments and the above discussions draw several conclusions:

- The lazy hillclimber with random repositioning performs surprisingly well, but is still no match for the loGA.
- Simulated annealing handles the basic problem with four positions well and (on large maps) fast. The loGA achieves the same quality with higher computation costs (on large maps). It is expected that loGA will outperform SA on more complex problems, but this was difficult

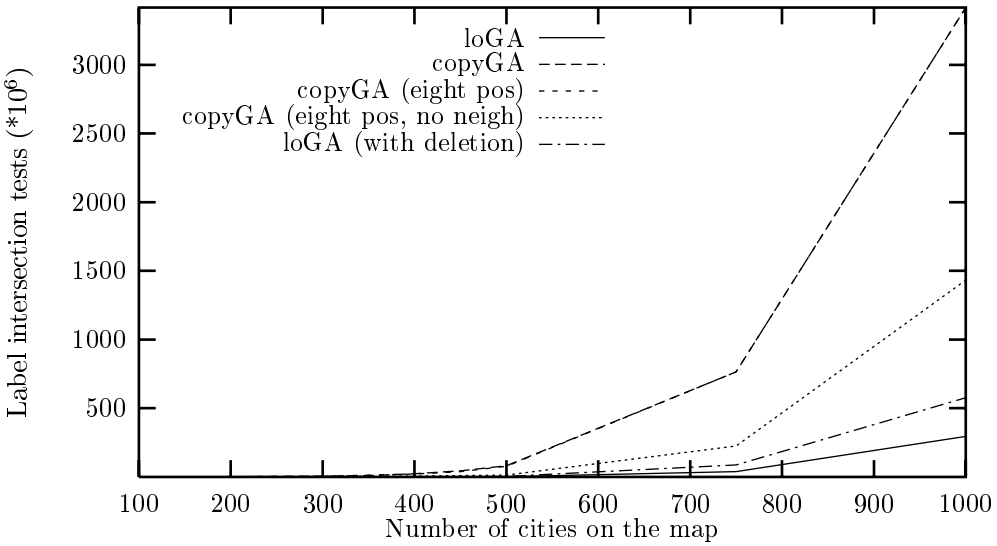

Figure 43: Time needed by the algorithms in the comparison experiments - most time consuming algorithms. Note that this is given as a rough indication of running times. The number of label intersection tests given is the amount needed to achieve the quality reported in section 5.2.

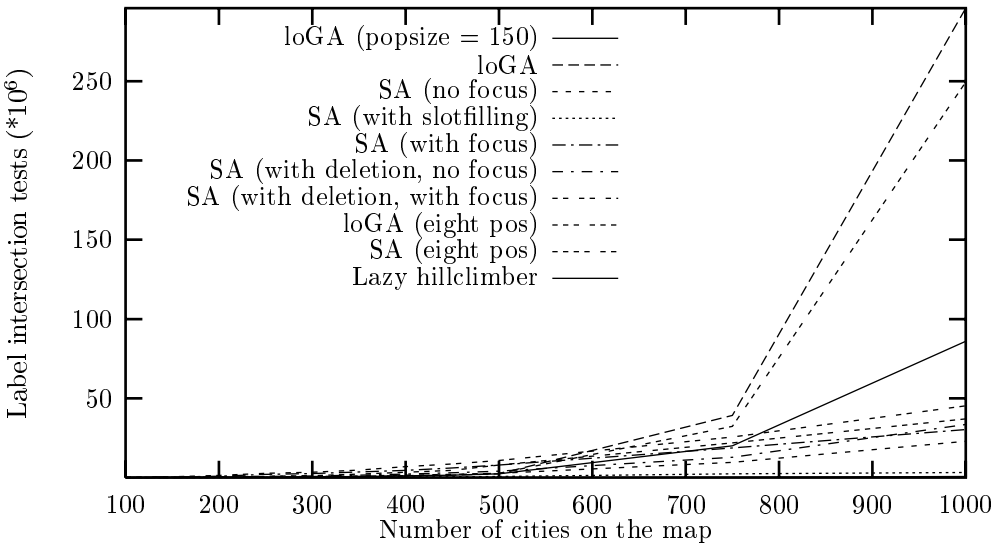

Figure 44: Time needed by the algorithms in the comparison experiments - least time consuming algorithms. See the comment in figure 43.

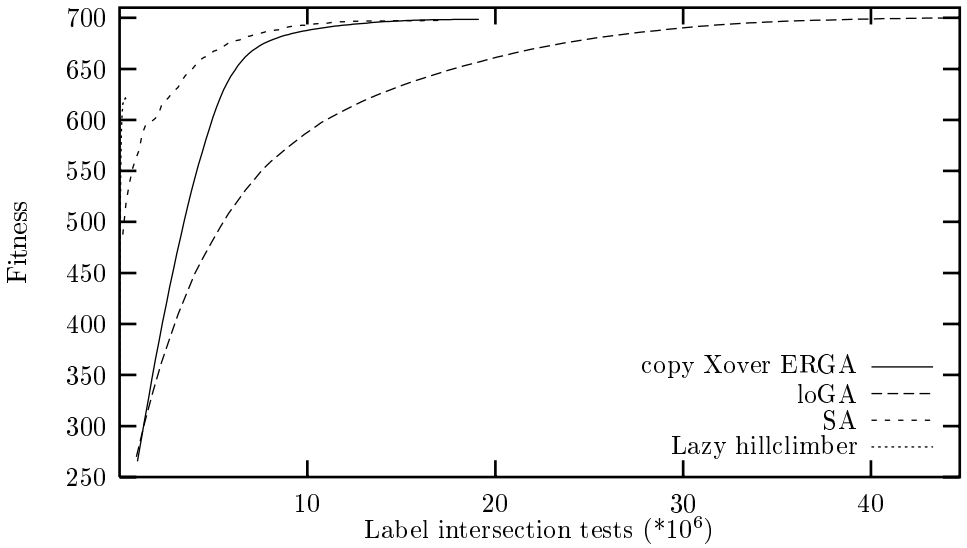

Figure 45: A plot of the running times.

to verify experimentally since no recommendations for the cooling schedule were given by Christensen et al. for more complex problems.

- The algorithm of Verner et al. is outperformed in terms of quality and speed by the loGA.
- The loGA can easily be extended to other problem instances.

6 Robustness of the GA

The notion of robustness is something that we consider to be quite important, so we want to summarise and stress the points about this topic that were mentioned earlier.

Most GA's get used as a 'black box': fabricate a nice fitness function, use a standard library and see what happens if we let it run. Sometimes this works well, but in practice this happens only for problems which were easy to begin with. Most seasoned GA-builders (see for example [3]) therefore acquire a 'feeling' for constructing GA's which depends on carefully choosing the representation of the solution, picking the right operators and so on. This produces GA's that are successful at tackling difficult problems. A disadvantage is that building GA's is more like an art than a science.

Other GA-builders try to gain a deeper understanding in how GA's really work and come up with analyses of concepts such as schema processing, convergence speed, deception and multi-modality. Research in this area is progressing well, but an illuminating theory on GA's which explains all is far from being.

The map labeling problem is a difficult one and a careful balance has to be made between art and theory. Not only is even the most basic problem (minimizing overlap for points with four fixed positions) NP-complete, also a wide variety of quality measures have to be used due to the large variety in possible conflicts (for example point with river, point with sea, etc.). It is therefore wise to keep this in mind when building a GA that has to work on these problems. It is not very practical (in terms of design effort) to build a GA that works well on point features but which can not be extended to other problem instances. Summarizing, we want a GA which is *robust* in the sense that we can extend it to other problem instances without much difficulties.

Which these thoughts in mind, we have constructed a GA which has the following points which account for its robustness:

- P_c, the crossover probability, can safely be set to 1 because of the elitist recombination scheme.
- P_m, the mutation probability, can be set to 0 (avoiding the disadvantages of random mutation) because local optimisers are used.
- No tuning of fitness weights is necessary because multiple-term fitness functions are avoided.
- Hitchhiking is prevented because crossover masks are fine grained as the result of sampling.
- Genetic drift is avoided because of the constant selection pressure which acts on all parts of the solution (all points on the map).
- Extendibility is offered through the use of local optimisers.
- The alphabet is reduced using the local optimisers so the search space remains tractable.

In the GA as it stands (which works well on the problem of labeling points) more 'intelligence' can easily be built in using the local optimisers. This means that we expect to be able to extend this algorithm without dramatic losses in quality to more difficult problems. This can be done without recalibrating the algorithm, which is a major advantage when building a algorithm for a problem which is so multifaceted as map labeling.

One point of calibration is still an open question however: that is the question of the optimal population size. Too large a population is a waste of valuable computing resources, too little a population means running the risk of getting inferior solutions. There are two answers to this problem. First, we could make a analytic model of the process and derive a formula for the population size. This is difficult, but it would produce valuable insights in the workings of the GA. However, this analysis might not be extendible when the problem changes. The other method is by making the population size adaptive, as is described in [16]. Further research is being done to tackle this problem. Note however that since only one parameter needs to be tuned this is something the user of the GIS can do because he can see (literally on the resulting map) what the effect is of changing the population size.

7 Conclusion

This paper described a new genetic algorithm for the problem of labeling maps. It was shown that the results of this algorithm are equal or better (as tested in the framework of Christensen et al.) than the results of other algorithms presently known in literature. Besides constructing an algorithm which could solve the problem of labeling point features well, care was taken to make the algorithm robust in the sense that it would be possible to extend the algorithm to other problem instances without major changes or time consuming recalibrations of user defined constants.

Topic of further research are:

- How to speed up the genetic algorithm further, especially on large or very dense maps.
- Extending the framework of the GA to include line- and area features
- Gaining insight in the relation between problem complexity and population size.

Acknowledgements

The software for this work used the GALib genetic algorithm package, written by Matthew Wall at the Massachusetts Institute of Technology.
Thanks go to Roger Wainwright for providing the data which was used in figure 7.

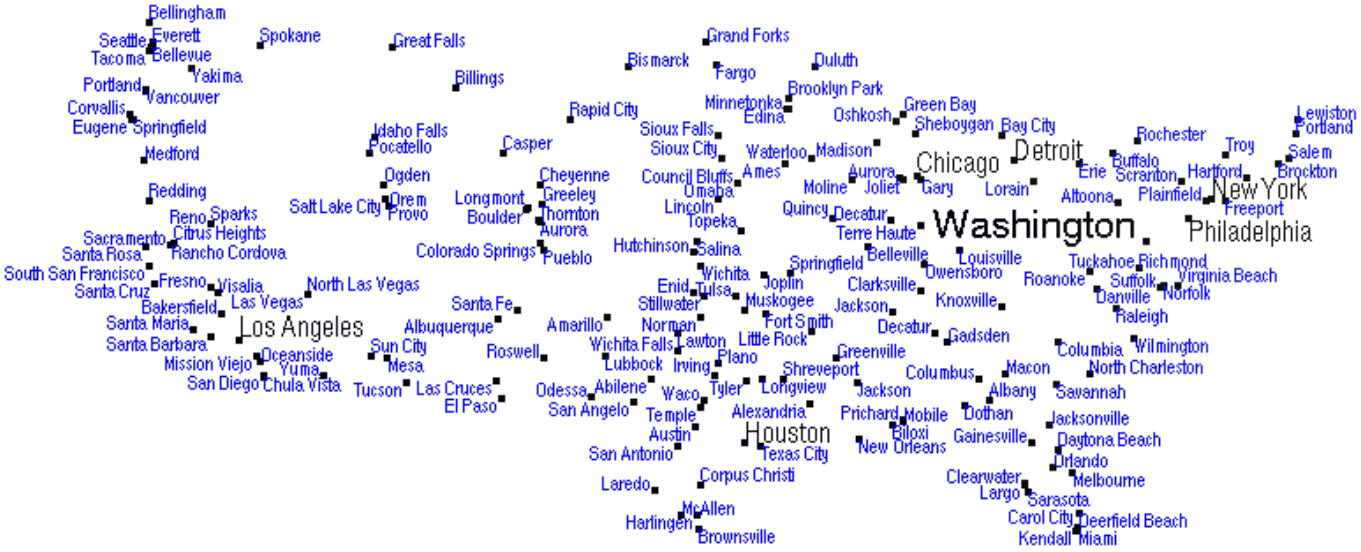

Figure 46: A labeling of major cities in the USA.

References

[1] H. Asoh and H. Mühlenbein. On the mean convergence time of evolutionary algorithms without selection and mutation. In Y. Davidor, H. Schwefel, and R. Männer, editors, *Lecture Notes in Computer Science, Vol. 866: Parallel Problem Solving from Nature PPSN-III.*, pages 98–107. Springer-Verlag, 1994.

[2] 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.

[3] L. Davis. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, New York, 1991.

[4] K. DeJong. *Analysis of Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, The University of Michigan, 1975.

[5] Y. Djouadi. Cartage: A cartographic layout system based on genetic algorithms. In *Proc. EGIS*, pages 48–56, 1994.

[6] M. Feigenbaum. Method and apparatus for automatically generating symbol images against a background image without collision utilizing distance-dependent attractive and repulsive forces in a computer simulation, 1994. Assigned to Hammond Inc., Maplewood, New Jersey. U.S. Patent filed 11/5/93, received 10/11/94.

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

[8] D. Goldberg and P. Segrest. Finite markov chain analysis of genetic algorithms. In *Genetic algorithms and their applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 1–8, 1987.

[9] G. Harik, E. Cantú-Paz, D. Goldberg, and B. Miller. The gambler's ruin problem, genetic algorithms, and the sizing of populations. In *Proceedings of the 1997 IEEE International Conference On Evolutionary Computation*, pages 7–12, 1997.

[10] S. Hirsch. An algorithm for automatic name placement around point data. *The American Cartographer*, 9(1):5–17, 1982.

[11] E. Imhof. Positioning names on maps. *The American Cartographer*, 2(2):128–144, 1975.

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

[13] G. Langran and T. Poiker. Integration of name selection and name placement. In *Proc. Auto-Carto 8*, pages 50–64, 1986.

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

[15] H. Mühlenbein and D. Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm: Continuous parameter optimization. *Evolutionary Computation*, 1(1):25–49, 1993.

[16] R. Smith. Adaptively resizing populations: An algorithm and analysis. Technical report, University of Alabama, February 1993.

[17] D. Thierens. Selection schemes, elitist recombination, and selection intensity. In T. Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97)*, San Francisco, CA, 1997. Morgan Kaufmann.

[18] D. Thierens and D. Goldberg. Elitist recombination: An integrated selection recombination ga. In *Proceedings of the First IEEE International Conference on Evolutionary Computation*, pages 508–512. IEEE Service Center, Piscataway, NJ, 1994.

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

[20] P. Yoeli. The logic of automated map lettering. *The Cartographic Journal*, 9:99–108, 1972.

[21] S. Zoraster. Integer programming applied to the map label placement problem. *Cartographica*, 23(3):16–27, 1986.

[22] S. Zoraster. The solution of large 0-1 integer programming problems encountered in automated cartography. *Operations Research*, 38(5):752–759, 1990.

[23] S. Zoraster. Practical results using simulated annealing for point feature label placement. *Cartography and GIS*, 1998. to appear.

A Datafiles

This appendix summarises the experimental results of the runs which were done for the figures 12, 22, 23, 25, 28, 32, 33. All these runs were done on the same map. This map was a randomly generated map (according to the specifications of Christensen et al.): the dimensions of the map were 792 by 612, those of the labels were 30 by 7 and the points were randomly placed. A four-position model was used.

For all the runs the following factors were kept constant, unless mentioned otherwise in the comment of table 3:

- Deletion of labels was not allowed.
- The fitness function counted the number of free (without an intersection) labels on the map.

- No preferences for label positions were considered.
- The initialization procedure was the initializer that chooses for each gene a random allele in the range of valid positions.

In table 2 and 3 the settings of the runs are enumerated. For every entry in the tables five runs were done and averaged. The following terms were used:

- P_c : probability of crossover.
- P_m : probability of mutation.
- uniform : uniform crossover.
- 1pt : one point crossover.
- rival : rival based crossover.
- treeSwap : kd-tree based crossover.
- copy : copy crossover with masking of Verner et al..
- LO-mutator : a mutation operator which invokes the current local optimiser on a randomly chosen point.
- RR-mutator : a mutation operator which invokes random repositioning on a randomly chosen point.
- convergence : a stop criterion that terminates the algorithm when the average fitness in the population equals the fitness of the best individual
- flatline/x : a stop criterion that terminates the algorithm when the last tenth of iterations did not increase the average fitness in the population. The algorithm is not terminated unless the number of iterations exceeds x. Notes that for a generational GA an iteration equals a generation and for an incremental GA an iteration equals a recombination followed by mutation.

Table 2 describes for each run the mutator, crossover, local optimiser, population size and operator probabilities. When an entry is not applicable (e.g. which mutator when the probability of mutation is 0.0) the dash (-) is shown.

Table 3 describes several options which could be turned on or off for a run, and also shows the optimum (averaged over five runs) these runs found. The options are:

- Optimise parents : Apply local optimisers to the central point of a sample in the parent? See section 4.5.2.
- Use focus : Focus on regions with conflicts? See section 4.6.
- Repair : Repair border between genetic information from parents? See section 4.5.1.
- Adapt nr. of masks : as described in section 4.5.1 a multi-mask is composed of several masks. The number of these masks is adapted during the run of the algorithm. The treeSwap crossover was meant to be a geometric kind of one point crossover, with a fixed mask of one (the implementation uses masks to specify the nodes of the tree). This did not work well, so we also tried adapting the number of masks.
- Use ERGA: Use the elitist recombination scheme or a generational scheme with roulette-wheel selection? See section 4.1.1.

Tables 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 and 15 show the differences in quality for different algorithms ('S.D.' means 'Standard Deviation').

Run	Mutator	Crossover	Local Optimiser	Stop criterion	Population Size	P_c P_c	P_m P_m
r001	-	uniform	-	convergence	200	1.0	0.0
r002	-	treeSwap	-	convergence	200	1.0	0.0
r003	-	treeSwap	-	convergence	200	1.0	0.0
r004	-	1pt	-	convergence	200	1.0	0.0
r005	-	rival	-	convergence	200	1.0	0.0
r006	-	rival	slotfilling	convergence	200	1.0	0.0
r007	-	rival	slotfilling	convergence	200	1.0	0.0
r008	-	rival	-	convergence	200	1.0	0.0
r009	-	rival	slotfilling	convergence	200	1.0	0.0
r010	-	rival	-	convergence	400	1.0	0.0
r011	-	rival	slotfilling	convergence	200	1.0	0.0
r012	-	rival	random repos.	convergence	200	1.0	0.0
r013	-	rival	skip	convergence	200	1.0	0.0
r014	-	rival	slotfilling	convergence	200	1.0	0.0
r015	-	rival	slotfilling	convergence	200	1.0	0.0
r016	LO-mutator	-	random repos.	flatline/10000	1	0.0	1.0
r017	-	rival	slotfilling	convergence	10	1.0	0.0
r018	LO-mutator	-	random repos.	flatline/10000	1	0.0	1.0
r019	LO-mutator	-	slotfilling	flatline/4000	1	0.0	1.0
r020	-	copy	slotfilling	flatline/500	200	1.0	0.0
r021	RR-mutator	copy	slotfilling	flatline/25	200	0.9	0.1
r022	-	copy	slotfilling	flatline/500	200	1.0	0.0
r023	-	copy	slotfilling	flatline/500	200	1.0	0.0
r024	-	rival	slotfilling	convergence	200	1.0	0.0
r025	-	copy	slotfilling	flatline/500	200	1.0	0.0
r026	-	rival	slotfilling	convergence	75	1.0	0.0
r027	-	rival	slotfilling	convergence	200	1.0	0.0
r028	-	rival	slotfilling	convergence	200	1.0	0.0
r029	-	rival	slotfilling	convergence	200	1.0	0.0
r030	-	rival	slotfilling	convergence	200	1.0	0.0

Table 2: Settings of the runs used – 1.

Run	Optimise Parents?	Use Focus?	Repair?	Adapt nr. of masks?	Use ERGA?	Optimum found:	Comment:
r001	-	-	-	-	Yes	479	
r002	No	No	No	No	Yes	300.8	
r003	No	No	No	Yes	Yes	459.2	
r004	-	-	-	-	Yes	362.2	
r005	No	No	No	Yes	Yes	476.2	
r006	No	Yes	Yes	Yes	Yes	490	
r007	No	Yes	Yes	Yes	No	490	
r008	No	No	No	Yes	Yes	476.2	
r009	No	No	Yes	Yes	Yes	490	
r010	No	No	No	Yes	Yes	481.6	
r011	No	Yes	Yes	Yes	Yes	490	
r012	No	Yes	Yes	Yes	Yes	490	
r013	No	Yes	Yes	Yes	Yes	479.2	
r014	No	No	Yes	Yes	Yes	490	
r015	No	Yes	Yes	Yes	Yes	490	
r016	Yes	Yes	Yes	Yes	Yes	482	
r017	No	Yes	Yes	Yes	Yes	482.6	
r018	Yes	No	Yes	Yes	Yes	476.4	
r019	-	-	-	-	Yes	426.4	
r020	-	-	-	-	Yes	490	
r021	-	-	-	-	No	490	
r022	-	-	-	-	Yes	490	
r023	-	-	-	-	Yes	489.8	No selection.
r024	No	Yes	Yes	Yes	Yes	490	
r025	-	-	-	-	Yes	490	
r026	No	Yes	Yes	Yes	Yes	490	
r027	No	No	Yes	Yes	Yes	490	Slot-filling init.
r028	No	No	Yes	Yes	Yes	490	
r029	No	No	Yes	Yes	Yes	969.2	Different map.
r030	No	Yes	Yes	Yes	Yes	994.8	Different map.

Table 3: Settings of the runs used – 2.

Algorithm	Quality	S.D.	run 1	run 2	run 3	run 4	run 5
copy Xover ERGA (no neigh)	1	0	100	100	100	100	100
loGA (with deletion)	1	0	100	100	100	100	100
loGA	1	0	100	100	100	100	100
copyGA (eight pos)	1	0	100	100	100	100	100
loGA (eight pos)	1	0	100	100	100	100	100
loGA (popsize=150)	1	0	100	100	100	100	100
Lazy hillclimber	1	0	100	100	100	100	100
SA (with deletion, with focus)	1	0	100	100	100	100	100
SA (with focus)	1	0	100	100	100	100	100
SA (with slotfilling)	1	0	100	100	100	100	100
SA (with deletion, no focus)	1	0	100	100	100	100	100
SA (no focus)	1	0	100	100	100	100	100
SA (eight pos)	1	0	100	100	100	100	100
copyGA (eight pos, no neigh)	0.996	0.008	100	100	98	100	100
copyGA	0.994	0.012	97	100	100	100	100

Table 4: Differences in quality for maps of size 100.

Algorithm	Quality	S.D.	run 1	run 2	run 3	run 4	run 5
loGA (with deletion)	0.998667	0.00266667	150	150	150	149	150
SA (with deletion, with focus)	0.998667	0.00266667	150	150	150	149	150
SA (with deletion, no focus)	0.998667	0.00266667	150	150	150	149	150
copy Xover ERGA (no neigh)	0.997333	0.00533333	150	150	150	148	150
loGA	0.997333	0.00533333	150	150	150	148	150
loGA (eight pos)	0.997333	0.00533333	150	150	150	148	150
loGA (popsize=150)	0.997333	0.00533333	150	150	150	148	150
SA (with focus)	0.997333	0.00533333	150	150	150	148	150
SA (no focus)	0.997333	0.00533333	150	150	150	148	150
SA (eight pos)	0.997333	0.00533333	150	150	150	148	150
copyGA (eight pos)	0.996	0.008	150	150	150	147	150
copyGA (eight pos, no neigh)	0.996	0.008	150	150	150	147	150
Lazy hillclimber	0.994667	0.00653197	150	150	150	148	148
copyGA	0.990667	0.00997775	149	150	150	146	148
SA (with slotfilling)	0.974667	0.0180862	145	150	148	142	146

Table 5: Differences in quality for maps of size 150.

Algorithm	Quality	S.D.	run 1	run 2	run 3	run 4	run 5
loGA (with deletion)	0.998	0.00244949	199	200	200	199	200
loGA (eight pos)	0.998	0.004	200	200	200	198	200
SA (with deletion, with focus)	0.998	0.00244949	199	200	200	199	200
SA (with deletion, no focus)	0.998	0.00244949	199	200	200	199	200
SA (eight pos)	0.998	0.004	200	200	200	198	200
copyGA (eight pos, no neigh)	0.997	0.006	200	200	200	197	200
copy Xover ERGA (no neigh)	0.996	0.00489898	198	200	200	198	200
loGA	0.996	0.00489898	198	200	200	198	200
loGA (popsize=150)	0.996	0.00489898	198	200	200	198	200
Lazy hillclimber	0.996	0.00489898	198	200	200	198	200
SA (with focus)	0.996	0.00489898	198	200	200	198	200
SA (no focus)	0.996	0.00489898	198	200	200	198	200
copyGA	0.996	0.00489898	198	200	200	198	200
copyGA (eight pos)	0.995	0.00632456	200	198	200	197	200
SA (with slotfilling)	0.969	0.00663325	193	194	194	192	196

Table 6: Differences in quality for maps of size 200.

Algorithm	Quality	S.D.	run 1	run 2	run 3	run 4	run 5
loGA (with deletion)	0.9968	0.0016	249	249	249	249	250
loGA (eight pos)	0.9968	0.00391918	250	248	250	248	250
copyGA (eight pos, no neigh)	0.9968	0.00391918	250	248	250	248	250
SA (with deletion, with focus)	0.9968	0.0016	249	249	249	249	250
SA (with deletion, no focus)	0.9968	0.0016	249	249	249	249	250
SA (eight pos)	0.9968	0.00391918	250	248	250	248	250
copyGA (eight pos)	0.9952	0.00587878	250	247	250	247	250
copy Xover ERGA (no neigh)	0.9936	0.0032	248	248	248	248	250
loGA	0.9936	0.0032	248	248	248	248	250
loGA (popsize=150)	0.9936	0.0032	248	248	248	248	250
SA (with focus)	0.9936	0.0032	248	248	248	248	250
SA (no focus)	0.9936	0.0032	248	248	248	248	250
copyGA	0.992	0.00505964	248	248	248	246	250
Lazy hillclimber	0.9888	0.00688186	247	248	245	246	250
SA (with slotfilling)	0.9624	0.02049	238	242	237	236	250

Table 7: Differences in quality for maps of size 250.

Algorithm	Quality	S.D.	run 1	run 2	run 3	run 4	run 5
loGA (with deletion)	0.997333	0.00133333	299	299	299	299	300
loGA (eight pos)	0.997333	0.00326599	300	298	300	298	300
copyGA (eight pos, no neigh)	0.997333	0.00326599	300	298	300	298	300
SA (with deletion, with focus)	0.997333	0.00133333	299	299	299	299	300
SA (with deletion, no focus)	0.997333	0.00133333	299	299	299	299	300
SA (eight pos)	0.997333	0.00326599	300	298	300	298	300
copyGA (eight pos)	0.996	0.00326599	298	298	300	298	300
copy Xover ERGA (no neigh)	0.994667	0.00266667	298	298	298	298	300
loGA	0.994667	0.00266667	298	298	298	298	300
loGA (popsize=150)	0.994667	0.00266667	298	298	298	298	300
SA (with focus)	0.994667	0.00266667	298	298	298	298	300
SA (no focus)	0.994667	0.00266667	298	298	298	298	300
copyGA	0.994667	0.00266667	298	298	298	298	300
Lazy hillclimber	0.983333	0.0107497	296	296	292	291	300
SA (with slotfilling)	0.956	0.018306	282	293	282	283	294

Table 8: Differences in quality for maps of size 300.

Algorithm	Quality	S.D.	run 1	run 2	run 3	run 4	run 5
loGA (with deletion)	0.996571	0.00213809	348	349	349	348	350
loGA (eight pos)	0.996571	0.00279942	348	348	350	348	350
SA (with deletion, with focus)	0.996571	0.00213809	348	349	349	348	350
SA (eight pos)	0.996571	0.00279942	348	348	350	348	350
copyGA (eight pos, no neigh)	0.995429	0.00427618	346	348	350	348	350
SA (with deletion, no focus)	0.994857	0.00379043	348	349	348	346	350
copy Xover ERGA (no neigh)	0.993143	0.00427618	346	348	348	346	350
loGA	0.993143	0.00427618	346	348	348	346	350
loGA (popsize=150)	0.993143	0.00427618	346	348	348	346	350
SA (with focus)	0.993143	0.00427618	346	348	348	346	350
SA (no focus)	0.993143	0.00427618	346	348	348	346	350
copyGA (eight pos)	0.992571	0.0049816	345	348	348	346	350
copyGA	0.992	0.00279942	346	348	348	346	348
Lazy hillclimber	0.975429	0.00713714	342	340	340	339	346
SA (with slotfilling)	0.94	0.0190381	339	333	320	324	329

Table 9: Differences in quality for maps of size 350.

Algorithm	Quality	S.D.	run 1	run 2	run 3	run 4	run 5
loGA (eight pos)	0.996	0.002	398	398	398	398	400
SA (with deletion, with focus)	0.996	0.002	398	398	398	398	400
SA (eight pos)	0.996	0.002	398	398	398	398	400
loGA (with deletion)	0.9955	0.00244949	398	398	397	398	400
copyGA (eight pos, no neigh)	0.9955	0.00244949	398	398	398	397	400
SA (with deletion, no focus)	0.9945	0.00331662	396	398	397	398	400
copyGA (eight pos)	0.992	0.00509902	396	398	394	396	400
copy Xover ERGA (no neigh)	0.9915	0.0043589	396	396	395	396	400
loGA	0.9915	0.0043589	396	396	395	396	400
loGA (popsize=150)	0.9915	0.0043589	396	396	395	396	400
SA (with focus)	0.9915	0.0043589	396	396	395	396	400
SA (no focus)	0.9915	0.0043589	396	396	395	396	400
copyGA	0.988	0.00244949	394	396	394	396	396
Lazy hillclimber	0.9555	0.0134536	384	383	372	384	388
SA (with slotfilling)	0.939	0.0162481	373	388	369	373	375

Table 10: Differences in quality for maps of size 400.

Algorithm	Quality	S.D.	run 1	run 2	run 3	run 4	run 5
loGA (eight pos)	0.995556	0.00281091	446	448	448	448	450
SA (eight pos)	0.995556	0.00281091	446	448	448	448	450
copyGA (eight pos, no neigh)	0.995111	0.00294811	446	448	447	448	450
SA (with deletion, with focus)	0.994667	0.00226623	446	448	448	447	449
loGA (with deletion)	0.994222	0.00226623	446	448	447	447	449
SA (with deletion, no focus)	0.992889	0.00326599	444	448	447	447	448
copy Xover ERGA (no neigh)	0.988444	0.00514482	441	446	445	444	448
loGA	0.988444	0.00514482	441	446	445	444	448
loGA (popsize=150)	0.988444	0.00514482	441	446	445	444	448
SA (with focus)	0.988444	0.00514482	441	446	445	444	448
SA (no focus)	0.988444	0.00514482	441	446	445	444	448
copyGA (eight pos)	0.985778	0.0101932	436	448	441	445	448
copyGA	0.983111	0.00989825	434	446	442	444	446
Lazy hillclimber	0.938667	0.0114698	420	420	416	431	425
SA (with slotfilling)	0.925778	0.0135974	407	419	413	419	425

Table 11: Differences in quality for maps of size 450.

Algorithm	Quality	S.D.	run 1	run 2	run 3	run 4	run 5
loGA (eight pos)	0.992	0.00357771	494	496	494	498	498
copyGA (eight pos, no neigh)	0.992	0.00357771	494	496	494	498	498
SA (eight pos)	0.992	0.00357771	494	496	494	498	498
loGA (with deletion)	0.99	0.00219089	494	495	494	495	497
SA (with deletion, with focus)	0.9896	0.00366606	493	495	493	495	498
SA (with deletion, no focus)	0.9852	0.00271293	492	493	491	492	495
copyGA (eight pos)	0.9816	0.0082365	488	494	484	494	494
copy Xover ERGA (no neigh)	0.98	0.00657267	487	490	487	490	496
loGA	0.98	0.00657267	487	490	487	490	496
loGA (popsize=150)	0.98	0.00657267	487	490	487	490	496
SA (with focus)	0.98	0.00657267	487	490	487	490	496
SA (no focus)	0.9792	0.00627375	486	490	487	490	495
copyGA	0.968	0.0167809	470	486	482	486	496
Lazy hillclimber	0.9336	0.0109836	463	473	462	474	462
SA (with slotfilling)	0.9064	0.0172696	445	456	443	467	455

Table 12: Differences in quality for maps of size 500.

Algorithm	Quality	S.D.	run 1	run 2	run 3	run 4	run 5
loGA (eight pos)	0.967733	0.00770108	715	731	725	730	728
SA (eight pos)	0.966133	0.00811473	714	731	722	729	727
loGA (with deletion)	0.9616	0.00751413	712	728	723	718	725
copyGA (eight pos, no neigh)	0.960533	0.00882824	708	726	720	722	726
SA (with deletion, with focus)	0.9568	0.0100346	707	729	717	713	722
SA (with deletion, no focus)	0.954933	0.00644429	708	720	716	715	722
loGA	0.933333	0.0153883	680	712	704	695	709
loGA (popsize=150)	0.932533	0.0145229	680	711	701	697	708
copy Xover ERGA (no neigh)	0.931467	0.0154252	679	712	703	693	706
SA (with focus)	0.929867	0.0165806	676	714	699	695	703
SA (no focus)	0.929333	0.0172201	676	713	698	691	707
copyGA (eight pos)	0.9096	0.0185405	661	688	671	697	694
copyGA	0.871467	0.0286415	622	685	661	639	661
Lazy hillclimber	0.828533	0.0313103	614	596	663	605	629
SA (with slotfilling)	0.800533	0.0142261	582	607	600	599	614

Table 13: Differences in quality for maps of size 750.

Algorithm	Quality	S.D.	run 1	run 2	run 3	run 4	run 5
loGA (eight pos)	0.9264	0.00884534	917	935	915	936	929
SA (eight pos)	0.9154	0.0110018	902	929	903	924	919
loGA (with deletion)	0.9142	0.00453431	912	918	911	921	909
SA (with deletion, no focus)	0.902	0.00626099	898	913	897	905	897
SA (with deletion, with focus)	0.9016	0.00560714	902	908	898	907	893
copyGA (eight pos, no neigh)	0.901	0.0105641	895	915	888	912	895
loGA	0.8494	0.00733757	842	861	844	855	845
SA (no focus)	0.847	0.0069857	844	858	842	852	839
SA (with focus)	0.8464	0.00760526	838	856	840	855	843
loGA (popsize=150)	0.842	0.00961249	833	858	834	848	837
copy Xover ERGA (no neigh)	0.8336	0.00722772	826	843	826	841	832
Lazy hillclimber	0.7562	0.00847113	761	741	753	762	764
copyGA (eight pos)	0.7336	0.0208672	715	762	716	756	719
copyGA	0.6976	0.0101509	689	703	702	711	683
SA (with slotfilling)	0.6374	0.015819	622	650	629	662	624

Table 14: Differences in quality for maps of size 1000.

Algorithm	Quality	S.D.	run 1	run 2	run 3	run 4	run 5
loGA (with deletion)	0.789733	0.00237019	1187	1182	1190	1184	1180
SA (with deletion, no focus)	0.7696	0.00416547	1157	1153	1161	1158	1143
loGA (eight pos)	0.766933	0.00346667	1145	1147	1160	1151	1149
SA (with deletion, with focus)	0.766933	0.00302875	1153	1147	1144	1157	1151
SA (eight pos)	0.7472	0.00154344	1123	1122	1118	1118	1123
copyGA (eight pos, no neigh)	0.638133	0.00571003	956	943	969	956	962
SA (no focus)	0.634667	0.0065047	959	959	947	960	935
SA (with focus)	0.633733	0.00443421	957	947	958	951	940
loGA	0.633467	0.00552006	955	955	956	951	934
loGA (popsize=150)	0.607067	0.00527636	908	923	916	905	901
copy Xover ERGA (no neigh)	0.606533	0.00472158	919	916	907	908	899
Lazy hillclimber	0.567467	0.0125532	867	830	879	845	835
SA (with slotfilling)	0.363067	0.00782702	557	558	545	534	529
copyGA (eight pos)	0.176533	0.0095163	256	278	283	263	244
copyGA	0.174667	0.00893433	273	278	256	240	263

Table 15: Differences in quality for maps of size 1500.