Map Labeling

Diplomarbeit von Alexander Wolff

Freie Universität Berlin

Fachbereich Mathematik

Mai 1995

„Die Farbe der Zahl Acht ist blau."

Katja Plaisant

Introduction

Map lettering is one of the classical key problems that has to be solved in the process of map production. Usually the map producer does not only want to show the exact geographic positions of the features depicted but also explain properties of these features. She has to arrange this information on the map so that:

- for every piece of information it is intuitively clear which feature is described;
- the information is of legible size;
- different texts do not overlap.

These and in addition a lot of esthetic criteria are described by Imhof [I75] in an attempt to characterize good quality map lettering having mostly manual map making in mind. Nowadays there is an increasing need for large, especially technical maps, for which legibility is much more important than beauty.
The application which brought the problem to our attention is the design of groundwater quality maps by the municipal authorities of the City of Munich. They have a net of drillholes spread over the city. The map has to contain the location of these holes and for every hole a block of information such as measuring results or the ground water level.
The growing importance of such technical maps induces a need for the computerization of map making, the need for fully automated algorithms. Typically, labels in technical maps are axis-parallel rectangles of identical size. By rescaling one of the axes we can assume that the rectangles are squares. An adequate formalization is as follows:

The MAP LABELING Problem

Given n distinct points in the plane. Find the supremum σ_{opt} of all reals σ such that there is a set of n closed squares with side length σ, satisfying the following two properties.

1. Every point is a corner of exactly one square.
2. All squares are pairwise disjoint.

We call σ_{opt} the *optimal size*. A set of non-intersecting squares fulfilling (1) and (2) is called a *valid labeling*, see Figure 0.1 and 0.2.

Formann and Wagner showed that the problem is $\mathcal{NP}$-hard [FW91]. The main result of that paper is an approximation algorithm A that finds a valid labeling of at least half the optimal size. In addition, it is shown that no polynomial time approximation algorithm with a better quality guarantee

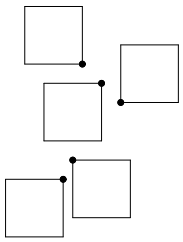

Figure 0.1: A valid labeling

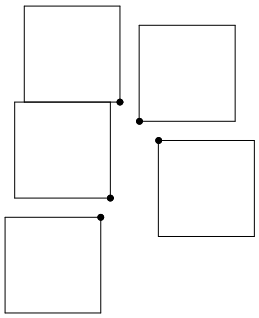

Figure 0.2: An optimal labeling for the example of Figure 0.1

exists if $\mathcal{P} \neq \mathcal{NP}$. Related results were reported in [AIK89] and [IA86]. The running time of A is in $\mathcal{O}(n \log n)$. In [W94] Wagner showed that there is a matching lower bound on the running time.

A conceptually works as follows: We start with infinitesimal equally sized squares attached to each point in all four possible positions. Then all squares are expanded uniformly. In order to resolve conflicts between them, we eliminate all those which would contain another point if they were twice as big. It is easy to show that after this process, a point p can not have more than two squares left which overlap other squares.

If we consider p a boolean variable and associate its squares with the values p and $\bar{p}$, we can generate a boolean formula consisting of clauses which encode all conflicts. Suppose square p was overlapping square $\bar{q}$ of a point q, this would give us the clause $\overline{(p \wedge \bar{q})} = (\bar{p} \vee q)$ meaning that we do *not* want p and $\bar{q}$ to be simultanously in the solution. If we join all such clauses with the $\wedge$–operator, the satisfiability of the formula tells us exactly whether there is a solution of the current size. Since all clauses consist of two laterals, the formula is of 2-SAT type, and can be evaluated in time proportional to its length [EIS76].

This works only because we make sure that no point has more than two squares left after the elimination phase. In order to achieve that, we often have to eliminate both of two conflict partners, where it would have sufficed to delete one to resolve the conflict. This seems to be the reason for the practically very bad behaviour of A. In fact, A usually produces solutions not much better than 50 percent of the optimum, which makes it nearly useless for practical problems. On the other hand this means that twice the size of A's solution is a good upper bound for the optimal size.

In Chapter 1 we present a heuristical approach that shows very convincing results [WW95a]. Instead of eliminating the squares as early as possible, it eliminates a square just when it is clear that it cannot be in any solution of the current size. The bad side effect of this is, that some points might have three or four squares left after the elimination phase. In order to handle this, three different methods H, I, and J are suggested to bring their number down to two. Method I is the winner of the experimental contest.

We suggest a hybrid algorithm that first runs A, and then uses its result to control the heuristics in two ways:

1. The approximation size of A gives a lower and an upper bound on the optimal size. These bounds are used to show that there is only a linear number of conflicts between overlapping squares. In addition to that, we use twice the upper bound as the width of a window with which we sweep across the sites to detect these conflicts in $\mathcal{O}(n \log n)$ time.

2. The result of the hybrid algorithm is the maximum of A's and the heuristics' result, thus guaranteeing the optimal approximation quality.

The simplest of the heuristics, H, is used by the City of Munich for the application mentioned above, by the PTT Research Labs of the Netherlands to produce on-line maps for mobile radio networks, and in a computer system for the automated search for matching constellations in a star catalogue [WKA94] as a tool to label the output on the screen. With a very similar algorithmic approach Formann and Wagner [FW] were able to solve the so-called METAFONT labeling problem posed by Knuth and Raghunathan [KR92].

In Chapter 2 we integrate the two parts of the hybrid method into a provably good and efficient approximation algorithm B of even better quality [WW95b]:
In order to achieve A's running time efficiency of $\mathcal{O}(n \log n)$, we introduce a new way of detecting conflicts between overlapping squares. It is based on an algorithm to find closest neighbours, which was suggested in [F92]. We maintain A's quality guarantee of 50 percent without being obliged to run A beforehand and use its result. In order to do so, we perform a test whether a solution can still be constructed from a subset of the squares which have not been eliminated so far during a certain phase of the algorithm. We improve this elimination phase with the help of an additional deletion rule for squares which are not needed in a valid solution.

We compare A, B, and the heuristics in an experimental evaluation using three different classes of random problems and a selection of problems arising in the production of groundwater quality maps by the authorities of the City of Munich.

Contents

Chapter 1

The Heuristics

1.1 A Theoretical Foundation

Definition 1 *For a point p in the plane, a real $\sigma \geq 0$, and $i \in \{1,2,3,4\}$, denote by σp_i an axis-parallel square with side length σ and p in its southwest, southeast, northeast respectively northwest corner. The enumeration is chosen like that of quadrants.*
We will call p_i a candidate *of the* site *p. Where the edge length σ is omitted, we refer to a candidate of the current label size.*
A solution of size σ *is a valid labeling with candidates of side length σ.*

For technical reasons, we will from now on consider a candidate an open square, plus the open edges incident to the site. Note that this excludes all corner points, especially the site itself. The idea is that we shrink the squares by a tiny bit, so that an optimal labeling is a valid labeling, too.

Definition 2 *of some special label sizes:*

σ_{dead}	=	*largest label size at which all sites still have a candidate which does not contain a site.*
σ_{opt}	=	*size of the maximum valid solution. This is equivalent to the previous definition of σ_{opt}.*
σ_{lower}	=	*size of the solution of the Approximation Algorithm A*
σ_{upper}	=	$2\sigma_{lower}$

Corollary 3 $\sigma_{lower} \leq \sigma_{opt} \leq \sigma_{upper} \leq \sigma_{dead}$

Proof. $\sigma_{opt} \leq \sigma_{upper}$ is of course due to A's approximation guarantee, see [FW91].
$\sigma_{upper} \leq \sigma_{dead}$: A stops at the latest at size $\sigma_{dead}/2$, because then there is a site all of whose candidates are eliminated. Therefore $\sigma_{lower} \leq \sigma_{dead}/2$. □

We say that two candidates *overlap* or have a *conflict* if they intersect and neither contains a site. Analogously, two sites are in conflict if any of their candidates are. One of the key words in the description of the heuristics is that of a *conflict size*. For a pair of candidates we define its conflict size as the largest edge length at which they do not intersect. We call a conflict size *interesting*, if it is not larger than σ_{upper}.

Lemma 4 *The number of interesting conflict sizes is linear.*

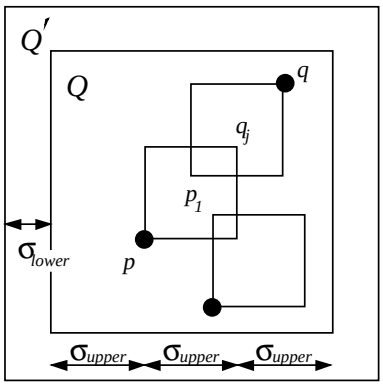

Figure 1.1:

Proof. Let s be the vector $(\sigma_{upper}, \sigma_{upper})$, and $\prec$ the lexicographical order on $\mathbb{R}^2$. Given a candidate p_i, say p_1, we define two squares as in Figure 1.1, $Q := \{z \in \mathbb{R}^2 \mid p - s \preceq z \preceq p + 2s\}$ and $Q' := \{z \mid p - \frac{3}{2}s \preceq z \preceq p + \frac{5}{2}s\}$, such that $\sigma_{upper} p_1 \subset Q \subset Q'$. Then clearly all sites q with candidates q_j, which might have a conflict with p_1 of size not greater than σ_{upper}, must lie within Q, because its border runs around p_1 at a distance of σ_{upper}. We know that there must be a partial solution of size σ_{lower} for the sites in Q. All candidates of such a solution must lie in Q', so Q cannot contain more than 64 sites. Therefore the number of conflicts of interesting size per candidate is constant. □

1.2 2-SAT

1.2.1 What's The Problem?

The 2-SAT problem is the satisfiability problem for conjunctive normal forms with at most two literals per clause. In other words:

The 2-SAT Problem

Given a formula f of the following type

$$f = (l_{11} \vee l_{12}) \wedge (l_{21} \vee l_{22}) \wedge \ldots \wedge (l_{m1} \vee l_{m2})$$

where the l_{ij} are literals of boolean variables $x_1, \ldots, x_n$. This means that $l_{ij} = x_k$ or $\bar{x_k}$ for some k. Then the 2-SAT problem is to decide whether there is a truth assignment to the x_i which makes f true.

1.2.2 The Link to Map Labeling

Suppose every site p_i had only two candidates of some given size. Then we could encode these two positions with x_i and $\bar{x_i}$. A conflict between two candidates, say x_j and $\bar{x_k}$ would then give us the clause $\overline{(x_j \wedge \bar{x_k})} = (\bar{x_j} \vee x_k)$, meaning that we do *not* want to have x_j and $\bar{x_k}$ simultanously in the solution because these candidates overlap. By joining all such clauses with the logical AND,

we get a formula of 2-SAT type. It is false as soon as any of the clauses is, or in map labeling language: as soon as a conflict cannot be resolved. Finding a truth assignment of this formula is therefore equivalent to finding a solution for the corresponding restricted Map Labeling decision problem — that is finding a mapping of the sites to one of their *two* candidates of given size such that these do not overlap.

1.2.3 A 2-SAT Algorithm

In [EIS76] an algorithm is given which solves a special case of the timetable problem. The restrictions to the general case which is $\mathcal{NP}$-complete, are quite severe: there is only one class, and all teachers are "binary", that is, they are available during two lessons, but teach exactly one. Again, if two teachers want to work at the same time, we have a conflict which can be encoded as above, suppose x_i and $\bar{x_i}$ were the lessons the i^{th} teacher could teach. So the question whether all lessons can be taught by a different teacher is equivalent to the 2-SAT problem, and therefore also to the restricted Map Labeling problem with two candidates per sites.

We modify the description of the algorithm suggested in [EIS76] to our vocabulary. The aim is to progressively map all sites to one of their two candidates. We call a candidate *chosen* if we map a site to it. A site is *impossible* if both its candidates are overlapped by candidates which have already been chosen, and *implied* if one of its candidates is overlapped by a chosen candidate.

```
procedure 2-SAT
  while not all sites have been mapped do
    switch
      case 1:  all unmapped sites are neither impossible nor implied
        make all temporary mappings permanent
        if there is still an unmapped sites s then
          map s arbitrarily to one of its two candidates
        PHASE := 1
      case 2:  there is an unmapped implied site
        map it temporarily to that of its two candidates
          which is not overlapped by a chosen candidate
      case 3:  there is an unmapped site which is impossible
        if PHASE = 2 then return FALSE
        else
          undo all temporarily mappings
          map s to its other candidate
          PHASE := 2
  return TRUE and the mapping computed
end 2-SAT
```

This algorithm clearly returns a positive answer only if a solution is found. We still have to show that it always discovers a solution if one exists.
Let a component of the evaluation be a set of sites whose mappings gained permanency simultanously in Case 1. These components may depend on arbitrary choices and on the order in which the sites are considered in Case 1. A new component is started only when no more site is implied. Therefore none of its sites has a candidate which is overlapped by a candidate chosen in a previous component. For that reason, we can focus on one component.

If we take one of its sites, say t, map it arbitrarily to one of its candidates (Case 1), follow all the implications (Case 2), and eventually fail (Case 3) — then we know that the candidate we chose can not be part of the solution. Therefore t's other candidate must be part of it, and so do all candidates to which we map the sites implied by this revised decision. All sites of our component are implied by definition.

What about the running time? The algorithm presented is a branching algorithm with limited branching depth — a decision is redone only once (Case 3). In the worst case, it could happen that we always initially map a site to the wrong candidate, which might imply $\mathcal{O}(n)$ other sites before we find an impossible site and have to undo everything, thus resulting in a time complexity of at least $\mathcal{O}(n^2)$. To avoid this, we try both decisions in Case 3 in "parallel" — that is swapping from one path of implications to the other after every step. These paths replace the two phases in the algorithm above. Now the quicker success stops the evaluation of the other possibility, so we can charge the successful path with the cost of the other attempt. If we find a solution, the successful path has mapped all sites only once, and the other path has not made more steps. The same idea holds for the case of a negative answer. By using appropriate data structures (like stacks for the temporary mappings of both paths and for the implications of a decision) we were able to implement this version of 2-SAT in $\mathcal{O}(n)$.

Our practical experiments showed however that the parallel version applied to our examples, in average mapped up to nearly 50 percent more sites temporarily. This might be due to the fact that many cases allow several solutions, and that a random choice of the candidate in Case 1 runs into the worst case only with a very small probability.

1.3 Structure

All three heuristics use a common framework. We first need to run the Approximation Algorithm A to get σ_{upper} and a solution of size σ_{lower}. What the heuristics do then, can be split up into the following parts:

1. Find all interesting conflict sizes.
2. Do a binary search on the interesting conflict sizes between σ_{lower} and σ_{upper}, and check for each size you look at, whether there is a solution or not, by going through the following three phases:

Phase I:	Preprocessing.
Phase II:	Eliminate candidates which can not be part of the solution.
Phase III:	For those points which still have two or more candidates left, choose exactly two, and check, whether this remaining problem is solvable by 2-SAT, as described in Section 1.2.

The heuristics differ in the way in which they choose those two candidates in Phase III.

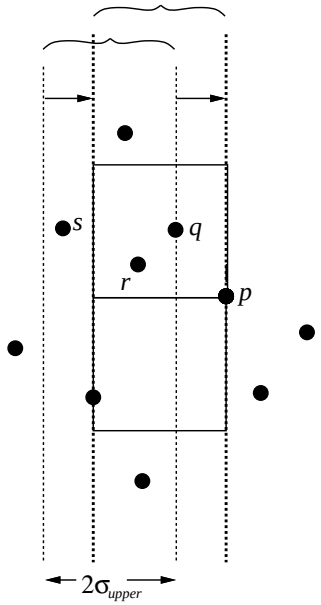

Figure 1.2: A window sweeping across the plane from left to right. The squares indicate which sites within the window might have a conflict of interesting size with p.

1.4 Finding Conflict Sizes

Since A supplies us with σ_{upper} which is an upper bound for σ_{opt}, we know that during the search for an optimal solution, only conflicts between sites at a distance of at most $2\sigma_{upper}$ in the $L_\infty-$metric, have to be considered. Therefore, we can use a sweep line — or rather, sweep window, approach to determine these conflicts of interest. As usually, the sites must be ordered lexicographically. Then we need two data structures: firstly, an event point queue as horizontal structure. This is a queue into which we insert sites entering the window on its right side and from which we delete points leaving it on the left. Further, we need a vertical structure, the sweep window status, which allows us to look up efficiently neighbours of new sites entering the window according to their y-coordinate. The width of the window is of course $2\sigma_{upper}$, so that sites entering it cannot be in conflict with those which have already left it on the other side.

The result of the sweep is a list of all conflict sizes between σ_{lower} and σ_{upper}. We do not have to consider any other label size, since the conflict graph does not change inbetween two consecutive interesting conflict sizes. We use this list afterwards to do a binary search for the best possible solution. In addition to this long list, for every candidate p_i we create a short list consisting of pointers to other candidates q_j, which are overlapping p_i before p_i touches the first site which we call $\delta(p_i)$, or reaches the size σ_{upper}. So for every p_i we need to know $\delta(p_i)$ and $d(p_i) := \|p-\delta(p_i)\|_\infty$ or ∞ if there is no site in the i^{th} quadrant relative to p. This information can be obtained by eight plane sweeps — one for the closest site in every 45^o octant — in $\mathcal{O}(n \log n)$ time according to [F92].

What happens when the right border of the window moves to the lexicographically next site? We want to keep the invariant that we have computed all interesting conflict sizes between the candi-

dates of all sites left of the right border of the window.

OUT: Since there cannot be any such conflict between the new site p entering the window on its right, and sites s leaving it on the left side (see Figure 1.2), we remove them from both the event point queue and the sweep window status. This can be done in constant time per site.

IN: Then we look at all successors (and predecessors) r of p in the vertical structure and compute all conflicts between r's and p's four candidates. With similar arguments as in the proof of Lemma 4 we show that there can only be a constant number of other sites r with $\|p - r\|_\infty \leq 2\sigma_{upper}$ in the window, and only the conflicts between those sites r and p are interesting.

We use (2, 4)–trees to implement the sweep window status, so inserting p costs $\mathcal{O}(\log n)$ time [MN95], but accessing a successor or predecessor r of p, or deleting p can then be done in constant time, computing the conflicts between r's and p's candidates of course, too.

This sums up to a running time of $\mathcal{O}(n \log n)$ for sorting the sites and for the sweep. As a consequence of Lemma 4, it requires only linear space — for the list of all conflict sizes and the short lists stored with every candidate, which have constant length.

1.5 Checking whether there is a solution for a fixed label size σ

1.5.1 Phase I: Preprocessing

We run through all candidates p_i. If $d(p_i) < \sigma$ we *eliminate* p_i, i. e. we will not consider it any more during the search for a solution of size σ, because then σp_i contains $\delta(p_i)$. Otherwise we create a new list of overlap information which is an excerpt from p_i's conflict list, refer to Section 3.1. We use the fact that two overlapping candidates remain in conflict until either contains a site if they are blown up simultanously. The elements of the new list consist of pointers to the overlap information of those candidates which actually overlap p_i for the given label size σ, the area of the intersection (needed for Heuristic J), and a pointer back to the candidate it belongs to. This can be done in linear time since the sum of the lengths of all conflict lists is linear, confer Section 1.4.

1.5.2 Phase II: Eliminating Impossible Candidates

We run once through all sites p. There are three cases:

- If all candidates of p have been eliminated, we stop and return "no solution" to the program which does the binary search on the conflict list.
- If p has candidates free of intersections with other candidates, we choose an arbitrary one of them (say p_i), and eliminate all other candidates p_j of p. Before their deletion, we have to do some updates for each of them: we delete its list of overlap information and the symmetric entries stored with those candidates which overlap it.
- If p has only one candidate p_i left, we do the same updates with all candidates q_j which overlap p_i, and then delete them.

While we do this we maintain a stack. On this stack we put all those candidates which now fulfill the same properties as p_i did before, i. e. do not intersect any other squares, or are the last candidates of their sites. Before we look at the next site p, we do all the decisions waiting for us on the stack. Since there is just a linear number of conflicts, and we can detect and delete each of them in constant time, Phase II takes us linear time.

Corollary 5 *If there is a solution of the current label size σ, then there is still one after Phase II.*

Proof. Suppose to the contrary that p_i is the first candidate after whose elimination the remaining problem becomes unsolvable. Then the following statement is true:

($\star$) Every solution π of the problem just before this elimination must contain p_i.

Consider the circumstances under which p_i could have been eliminated:

1. p_i contains a site q. This contradicts ($\star$).
2. p_i does not overlap other candidates, but the same holds for some p_j, and the algorithm decides to eliminate p_i.
   In this case we could replace p_i in π by p_j, contradicting ($\star$).
3. p_i overlaps q_j which is the last candidate of q.
   Then also q_j must be part of π, which again contradicts ($\star$).

□

At the end of Phase II we are done if all sites have exactly one candidate left. Otherwise we know that candidates of sites with several candidates — call them *active* — never intersect with those that are "the last of their breed", i. e. belong to sites with exactly one square left, because then the former ones would have been eliminated. So it is enough to focus on active candidates from now on. The others are already chosen as part of the solution, and do not interfere with the active ones any more.
As a consequence of Corollary 5 we also know that we have not yet returned "no solution" if there is one of size σ. So we could still find a solution with the help of 2-SAT as described before if no site had more than two candidates left. If some do, our heuristics try to get rid of the additional candidates in different ways until they all hand over the remaining problem to 2-SAT. Eliminating candidates, is of course, where we might lose a possible solution of the current size.

1.5.3 Phase III: The Heuristics Come into Play

Heuristic H We randomly choose two of the possible four candidates left per site, before we hand them over to 2-SAT. To increase the probability of a choice which enables a solution, this process can be repeated in case of a negative answer. Three repetitions yield good results without prolonging the running time too much.
Since we look at a (hopefully small) part of the linear number of conflicts, we will only get a linear number of clauses, resulting in a running time of $\mathcal{O}(n)$ for 2-SAT, and for this part of Heuristic H as well.

Heuristic I Here we run through all sites with active candidates twice. In the first run, we only look at those with four candidates left, eliminate the one with most conflicts, and make all decisions of the type we did in Phase II. During the second run, we do the same for sites which still

have three active candidates. Then the remaining problem (consisting only of sites with exactly two active candidates) is handed over to 2-SAT.
This takes linear time.

Heuristic J For the third variant, we put all active candidates left into a priority queue according to the sum of all intersection areas of a candidate p_i. We then delete the minimum p_i from the queue, and eliminate all candidates q_j which overlap it, and the other active candidates p_k belonging to p. If any of these decisions induces new ones according to the pattern used in Phase II, then these are made as well, before the next minimum is deleted from the queue. Naturally the sizes of the intersection areas, and the data structure, have to be updated accordingly. This process is repeated until either a site runs out of candidates ("no solution"), or no site has more than two of them left, so the remaining problem can be handed over to 2-SAT.
Using Fibonacci heaps to realize a priority queue that allows inserting and minimum deletions in $\mathcal{O}(\log n)$, and decreasing a key in constant time, this part of Heuristic J can be implemented to run in time $\mathcal{O}(n \log n)$, since there is just a constant number of conflicts to be resolved per candidate we look at.

1.6 Running Time Analysis

Phase I, II, and III can all be done in linear time, except for Heuristic J where Phase III takes $\mathcal{O}(n \log n)$ time due to the use of a priority queue. Since we have to look at $\mathcal{O}(\log n)$ conflict sizes during the binary search for the best solution, Step 2 is in $\mathcal{O}(n \log n)$, and $\mathcal{O}(n \log^2 n)$ for J. It dominates the total time complexity: Calling Approximation Algorithm A beforehand costs $\mathcal{O}(n \log n)$ time. This guarantees that finding conflicts in Step 1 takes $\mathcal{O}(n \log n)$ as well, so Heuristics H and I are in $\mathcal{O}(n \log n)$, while J is in $\mathcal{O}(n \log^2 n)$.

1.7 Experiments

1.7.1 The Exact Solver

The exact solver we used was implemented by Erik Schwarzenecker from Saarbrücken in C++. It uses some ideas of our Heuristic H but solves the problem in Phase III exactly. Thanks to its fine tuning it handles examples of up to 300 points even slightly faster than the heuristics, but we were forced to introduce a time limit of 5 minutes for larger *hard* and *dense* problem sets (see Section 1.7.2) to be able to perform any test row in reasonable time. This exact algorithm X shows exponential behaviour. For small examples it is very fast, for larger ones it is unreliable. Only few of the largest *hard* and *dense* examples took less than five minutes, and we have observed that the solution of examples beyond that bound then easily takes half an hour or much more. The CPU times of X are not comparable to those of the heuristics, since the latter are implemented in a very different way.

Still X is much better in practice than the Exact Solver S with a subexponential time bound suggested in [KMPS93]. It normally runs out of memory for more than 60–80 points, which we could improve to 120–150, when we made it solve only the problem remaining in Phase III. Even splitting this up into its connected regions, and dealing with those seperately, did not help a great deal.

1.7.2 Example Generators

We run the heuristics, and the Approximation Algorithms A and B on each of the examples produced by the four problem generators described below. For every size we average the approximation quality and running time over 30 tests. The information about the optimal size is yielded where possible by the Exact Solver X.

Random. We choose n points uniformly distributed in a square of size $10n \times 10n$.

Dense. Here we try to place as many squares as possible of a given size σ on a rectangle. We do this by randomly choosing points p and then checking whether σp_1 intersects with any of the σq_1 chosen before. We stop when we have unsuccessfully tried to place a new square 200 times. In a last step we assign a random corner point to each of the squares we were able to place without intersection, and return its coordinates. This method gives us a lower bound for the label size of the optimal solution. The size of the rectangle on which we place the squares is $\lfloor \alpha\sqrt{n} \rfloor \times \lceil \alpha\sqrt{n} \rceil$. α is a factor chosen such that the number of successfully placed squares is approximately n, the number of sites asked for.

Hard. In principle we use the same method as for Dense, that is, trying to place as many squares as possible into a rectangle. In order to do so, we put a grid of cell size σ on it. In a random order, we try to place a square of edge length σ into each of the cells. This is done by randomly choosing a point within the cell and putting a fixed corner of the square on it. If it overlaps any of those chosen before, we try to place it into the same cell a constant number of times.

Real World. The municipal authorities of Munich provided us with the coordinates of roughly 1200 ground water drill holes within a 10 by 10 kilometer square centred approximately on the city centre. From this list we extract a given number of points being closest to some centre point according to the L_∞–norm, thus getting all those lying in a square around this extraction centre, where the size of the square depends on the number of points asked for. For our tests we chose five different centres; that of the map and those of its four quadrants in order to get results from different areas of the city with strongly varying point density. This is due to the fact that many of the holes were drilled during the construction of subway lines which are concentrated in the city centre, see Figure 1.10.

1.7.3 Experimental Set-up

Actually we do not use σ_{upper} (that is twice the result of A) as an upper bound for the conflicts the heuristics have to look at, because then we would have to add the computation time of A to that of the heuristics. Though losing the theoretical bounds, it turned out to be much faster and to yield results of the same quality if we compute σ_{dead} and work with a longer list of conflict sizes (between 0 and σ_{dead} instead of between σ_{lower} and σ_{upper}) on which we do the binary search. Even the longer conflict lists of each candidate did not play a great roll, because σ_{upper} and σ_{dead} normally do not differ a lot anyway, especially not for large hard or dense examples where we have the highest number of conflicts per candidate.

1.7.4 Results

We show the two classical kinds of plots; time and quality. Quality here means the quotient of the solutions of a heuristic and the exact solver. Time is measured in CPU time, which is sufficient since it is closely related to the number of square–square conflicts, compare Figure 1.3 and 1.4

with Graphs 2 and 5 in Figure 1.5. The number of such conflicts on the other hand determines the number of crucial steps, namely finding all interesting conflicts once (Graph 2), and then extracting those valid for a certain σ in every step of the binary search (Graph 5).
The results both for time and quality are averaged only over those tests the exact solver managed within the time bound.
The standard deviation is represented by the length of the vertical bars in each point of the result plots in Figure 1.3 and 1.6 to 1.9.

Running Time

In Figure 1.3 we plot the running times of Heuristic I on the different example sets. H is slightly faster, J slightly slower. The test rows were performed on a Sun SPARC station 10. For a more detailed analysis of the running time of Heuristic I on hard examples, see Figure 1.4. The detailed time plots for other example classes look similar except for the gradients of the graphs showing the time needed for the binary search for the best solution and for finding 'general' conflicts once. These gradients depend on the average amount of conflicts per candidate. They are lower for random or real world examples. Initially we had great difficulties in understanding the empirically linear running time until we noticed that due to the use of integer coodinates the sizes of many different conflicts were equal. So even for the largest dense and hard examples we did not get more than 200 different conflict sizes, because the side length of the squares which our generators threw on the plane, was 100.

When we changed this square size to ten times the number of points of an example set, we received the linear number of conflict sizes we had expected, see Graph 4 of Figure 1.5. We have not yet found out why the number of conflicts summed up over all conflict sizes checked during the binary search (see Graph 5 of Figure 1.5), and the experimental running time (see Figure 1.3) still seem to be in $\mathcal{O}(n)$ instead of $\mathcal{O}(n \log n)$. It is also interesting to see that sorting (see Figure 1.4) which undoubtedly is in $\mathcal{O}(n \log n)$, has practically no influence on the running time for the problem sizes tested.

Approximation Quality

In Figures 1.6, 1.7, 1.8, and 1.9, the approximation quality of the three heuristics on the different example sets is plotted. On random and real world problems all three heuristics yield extremely good results. For an example, see Figure 1.11 and 1.12. On dense examples the differences between the heuristics become more clearly visible. Heuristic I is the best, yielding results of very high average quality. The behaviour on hard examples is still quite good but clearly becoming worse with an increasing number of points.
The quality of Algorithm A is extremely bad on Hard and Dense, and still useless from a practical point of view on random and real world examples.
A remark on the examples for which X did not give a result within the time bound: As mentioned above we did not include those in the calculation of the quality plots. But using the bound σ_{upper} resulting from Approximation Algorithm A, and taking into consideration the typical quality of A, we found out that the behaviour of the heuristics on those examples does not differ significantly from that on the other examples.

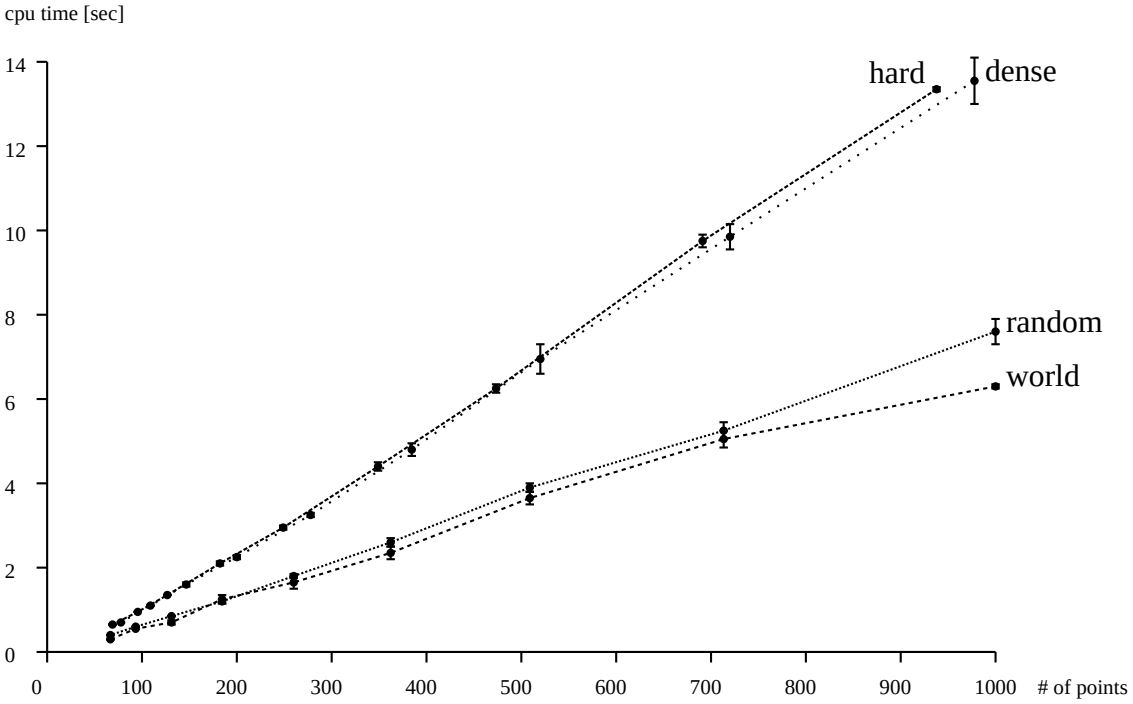

Figure 1.3: Running time of Heuristic I on different example classes

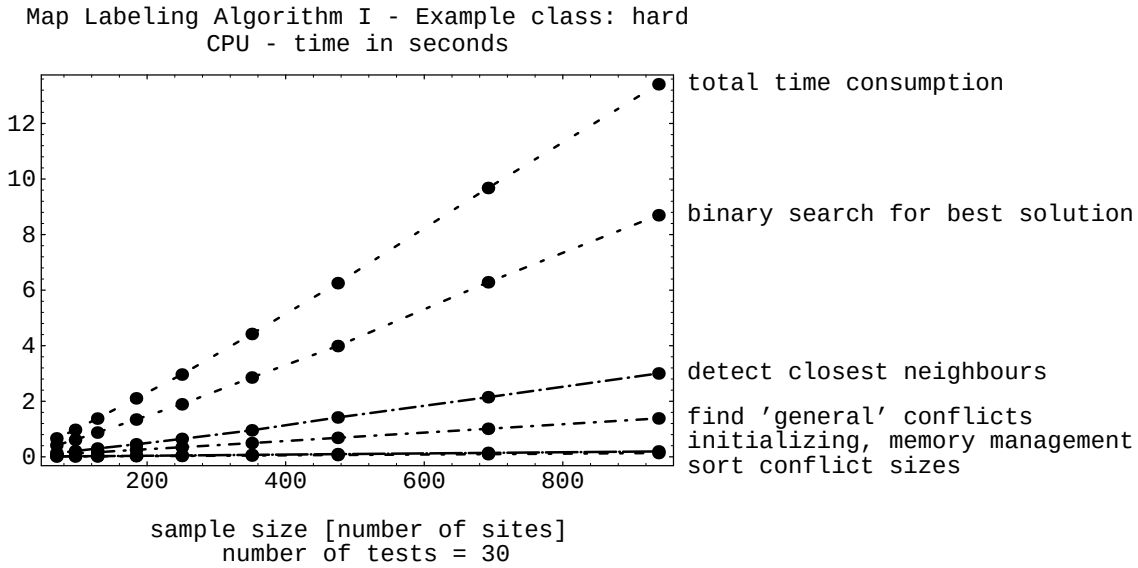

Figure 1.4: Running time of Heuristic I on hard examples in detail

```
Map Labeling Algorithm I - Example class: hard - number of tests = 30
1.    number of points looked at when detecting neighbours
2.    number of 'general' square - square conflicts
3.    number of conflict sizes including multiples
4.    size of array of conflict sizes used for binary search
5.    number of conflicts over all conflict sizes checked
6.    number of temporary assignments performed by 2-SAT
```

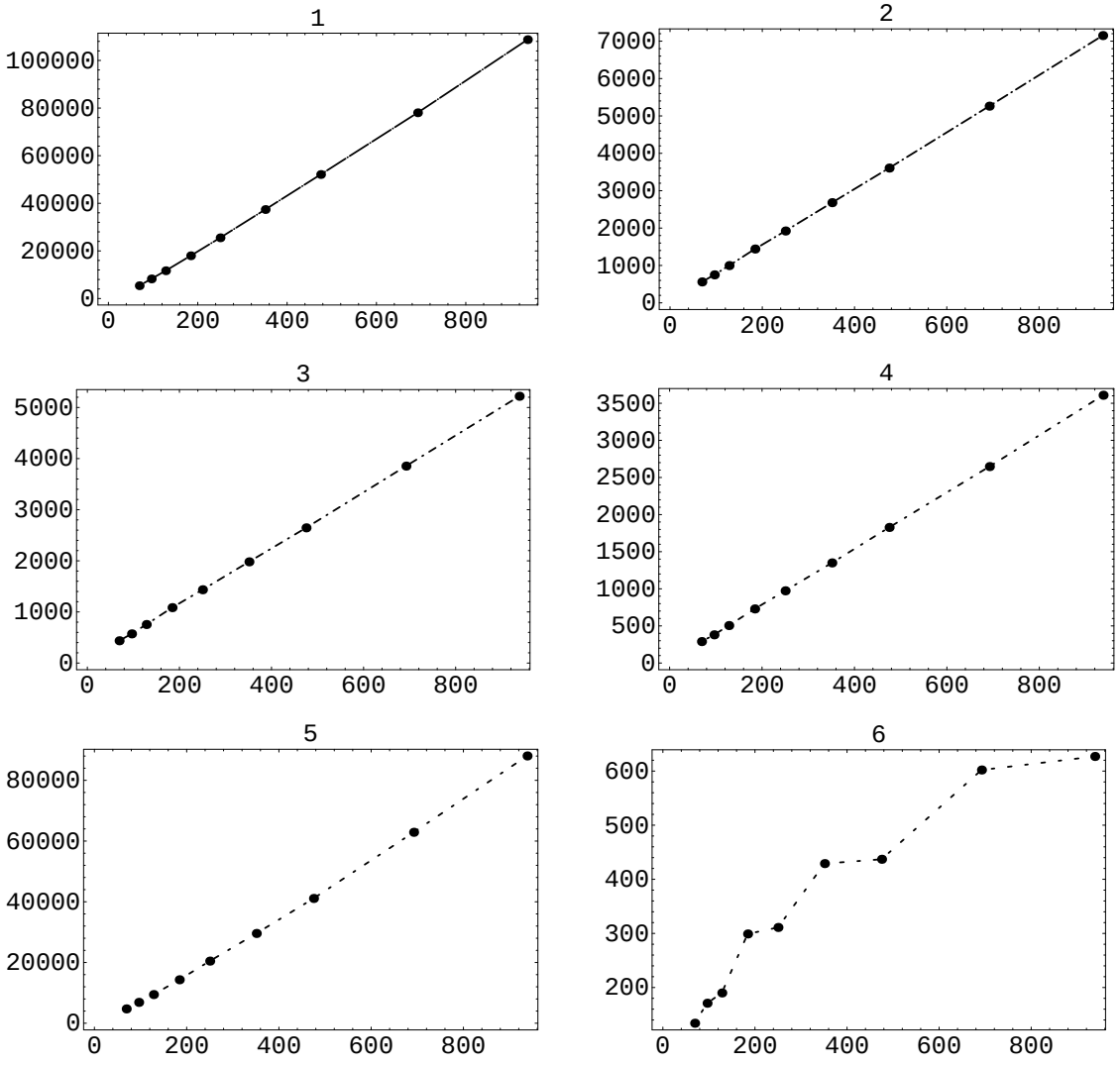

Figure 1.5: Average values of various counters which were installed to get a better understanding of the run time behaviour of the heuristics.

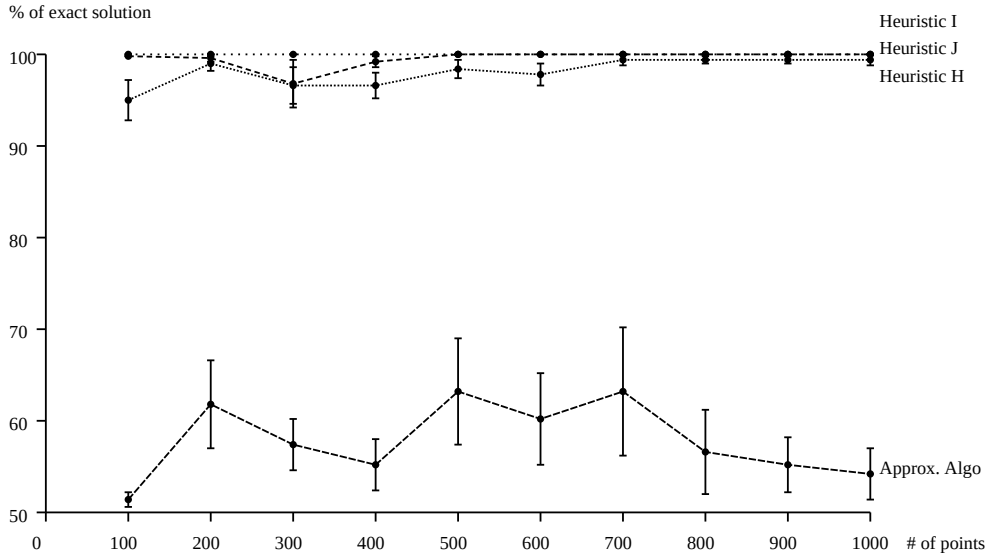

Figure 1.6: Quality of the algorithms on real world examples

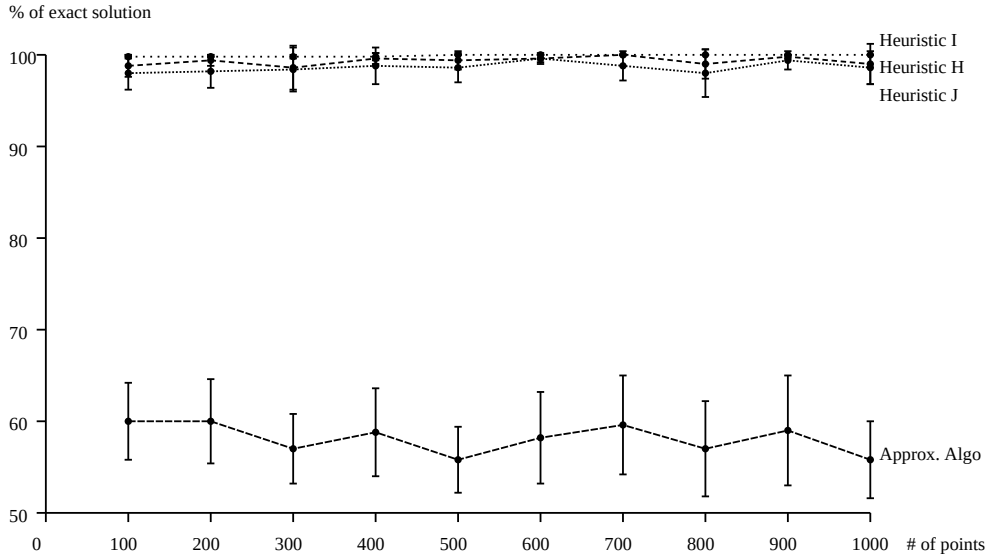

Figure 1.7: Quality of the algorithms on random examples

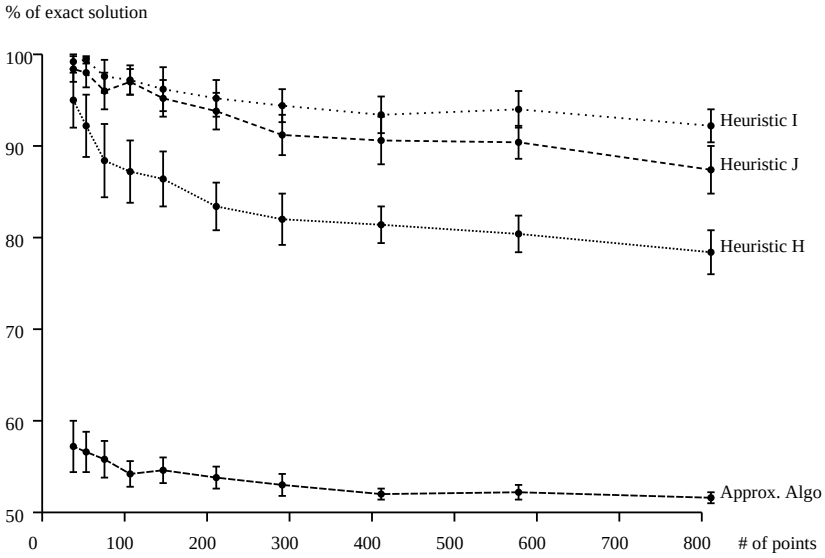

Figure 1.8: Quality of the algorithms on dense examples

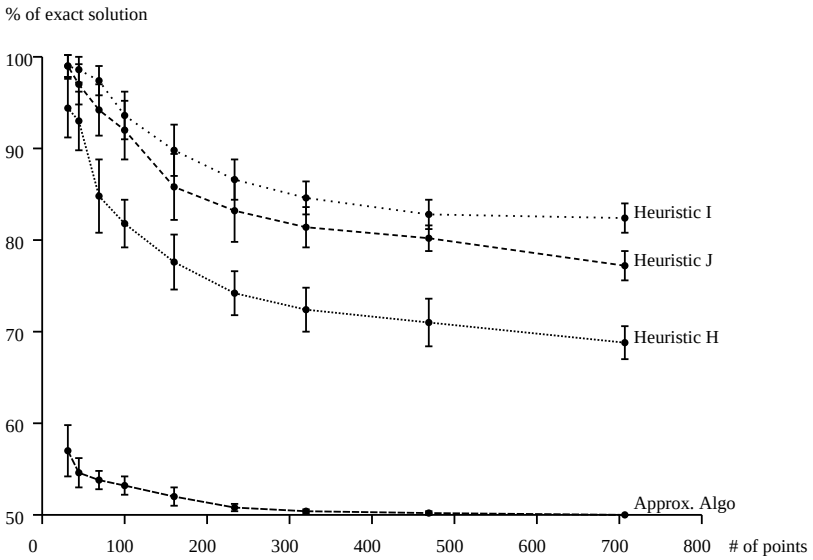

Figure 1.9: Quality of the algorithms on hard examples

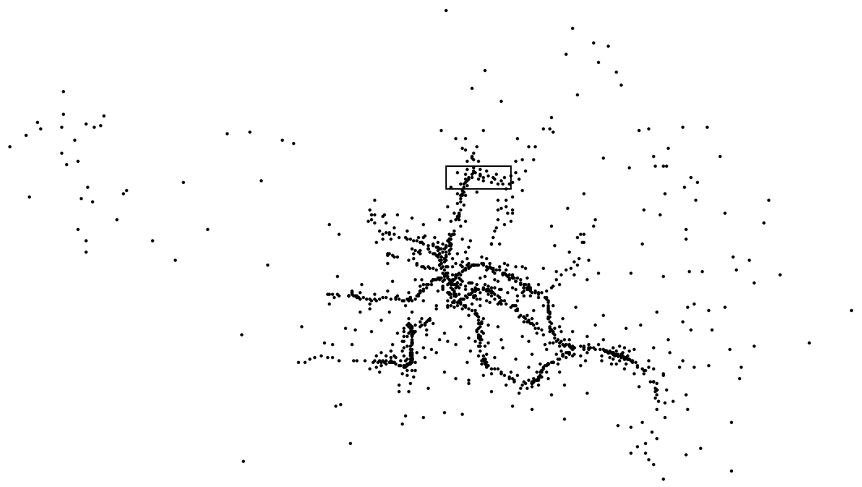

Figure 1.10: Map showing our sample data of approximately 1200 groundwater drillholes in Munich, and the section tested in Figures 1.11 and 1.12. There are no conflicts of interesting size between this section and the rest. The subway lines can be detected easily.

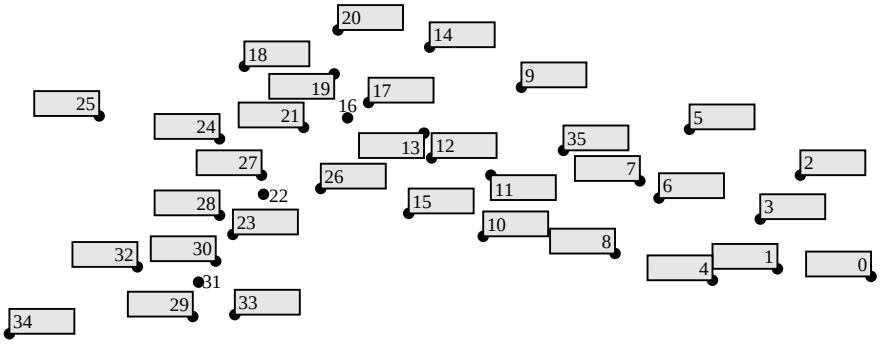

Figure 1.11: Solution of the program used by the authorities of the City of Munich before (label height 5000, 3 sites not labelled). It tries to maximize the number of sites labelled for a given size.

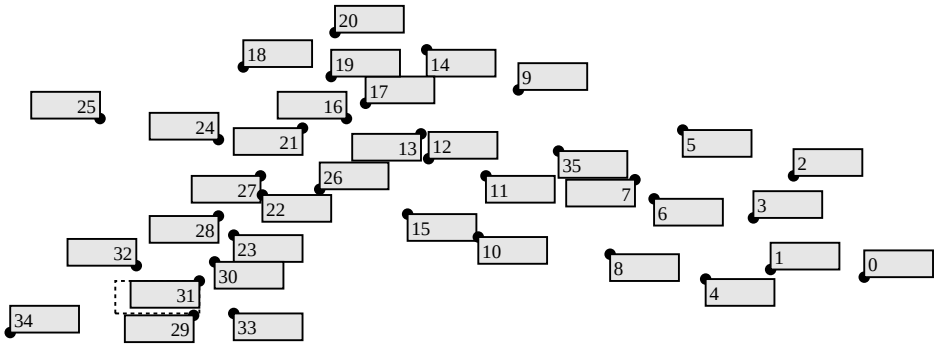

Figure 1.12: Solution produced by our heuristics (label height 5400, optimal).

Chapter 2

The Approximation Algorithm B

2.1 Structure

B differs from the hybrid algorithm suggested in Chapter 1 in the following way: It avoids to run A beforehand by finding the conflict sizes in Step 1 independently of A's result. In Phase II of Step 2, we introduce a new elimination rule that improves the performance substantially. The 2-SAT test of this phase maintains A's quality guarantee of 50 percent. Phase I and III are taken from Heuristic I which produced the best results in practice.

1. Find all important conflict sizes.
2. Do a binary search on these conflict sizes. Check for each size you look at, whether there is a solution or not, by going through the following three phases:

Phase I:	Preprocessing.
Phase II:	Eliminate candidates which cannot be part of the solution. Then do a 2-SAT test on a subset of those which have not been eliminated.
Phase III:	For those points which still have two or more candidates left, choose exactly two and check, whether this remaining problem is solvable by 2-SAT.

2.2 Finding Conflict Sizes

We show that it is sufficient to look at a constant number of closest neighbours of a site p in order to determine all conflict sizes which are not greater than the optimal label size. The reason for this strategy is that the k closest neighbours of all n sites p in any of the four quadrants relative to p can be found efficiently in $\mathcal{O}(kn \log n)$ time with an algorithm described in [F92].

Definition 6 *Let* $\mathrm{Quad}(p_i) = \infty p_i$*, that is the* i^{th} *quadrant relative to* p*,* $i \in \{1,2,3,4\}$*. Note that this includes the border except* p*.*
A conflict between a site s *and one of its eight closest neighbours in one of the four quadrants relative to* s *is called* important*. See Figure 2.1 for a counterexample. The size of such a conflict analogously is an important conflict size.*
A label *is the candidate of a site which has been chosen to be in the solution.*

Theorem 7 *All conflict sizes which are not important, are greater than* σ_{opt}*.*

We do not have to consider conflict sizes greater than σ_{opt}, because there cannot be a solution of that size. There is no need to check any other label size either, since the conflict graph of all

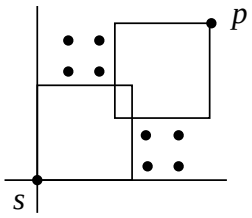

Figure 2.1: The conflict shown here between s_1 and p_3 is *not* important.

candidates does not change inbetween two consequetive conflict sizes. So it is sufficient to do the binary search in Step 2 of the algorithm on important conflict sizes.
It is obvious that the number of important conflict sizes is linear if we only have to look at a constant number of closest sites per candidate. These can be detected efficiently in $\mathcal{O}(n \log n)$ as mentioned above.

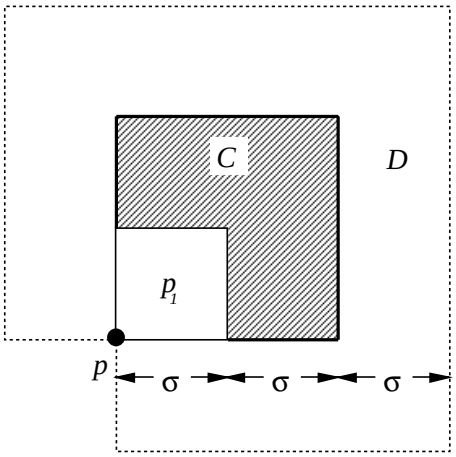

Figure 2.2:

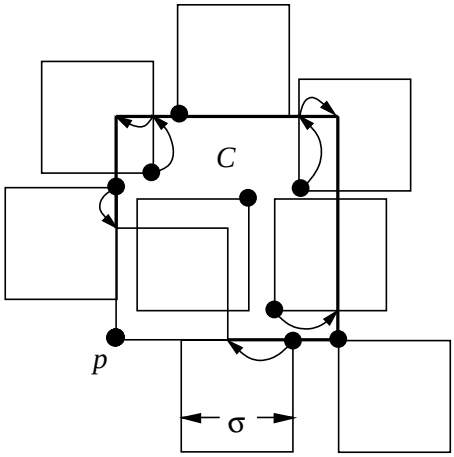

Figure 2.3:

Proof. For reasons of symmetry we can focus on conflicts in which some candidate p_1 is involved. First, consider only sites which lie in $\mathrm{Quad}(p_1)$. We want to show that the sizes of all conflicts between p_1 and candidates of these sites are greater σ_{opt}if they are not important. Suppose there is a conflict of size $\sigma \leq \sigma_{opt}$ between p_1 and one of the candidates of its k^{th} closest neighbour s^k. We show that there cannot be a partial solution of size σ for p and $s^1, \ldots, s^k$ if $k > 8$. This would be a contradiction to $\sigma \leq \sigma_{opt}$, because there is always a solution of size σ_{opt}, and it automatically is a solution for all smaller label sizes as well, and certainly for a subset of the sites.

It is clear that the distance between p and the sites with candidates in conflict with p_1 is bounded in the following way: $\sigma \leq \mathrm{d}(p, s^i) \leq 2\sigma \leq 2\sigma_{opt}$ for $i = 1, \ldots, k$, otherwise s^1 would lie in σp_1, or none of the candidates of s^k would have a conflict of size σ with p_1. The area C in which the

s^i can therefore lie, is shaded in Figure 2.2. Look at the area D which is where candidates of the sites in C can lie. The size of D is $15\sigma^2$, but the number of labels which can be placed in D, such that they are not in conflict with each other, is not 15 but 8, because the sites are restricted to C:

The idea is that we find out how many labels intersect the bold part of C's outline, and how many do not. We start with those sites whose labels contain a point of this line. We move them to a point of the outline which has either the same x- or y-coordinate — if the site is not already located on the outline. Since we move them "outwards", into a direction where no other label can lie, we avoid generating conflicts. We can then move sites along the line into its five corners. For those corners which are already contained by a label, take the site to which the label belongs, otherwise the closest site on either side of the outline. Now it is clear that apart from those in the corners not more than two sites can lie on the thick line, and another one in the interior of C, such that their labels do not overlap. Moving these labels on the one hand maximized the space in the interior of C, on the other it minimized the length of the intersection of each of the labels with the bold line. This means that in the original solution, too, not more than seven labels can have intersected the bold line, and only one can possibly have had an empty intersection.

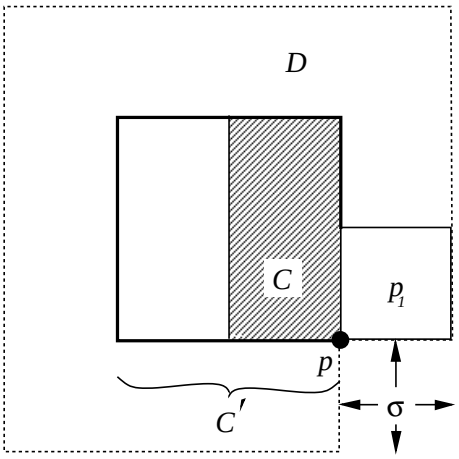

Figure 2.4:

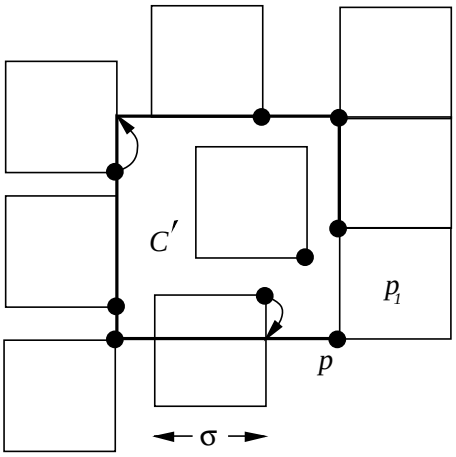

Figure 2.5:

Now consider all sites which lie in the second quadrant relative to p, and have candidates in conflict with p_1. Again, let C be the area in which such sites can lie. C is shaded in Figure 2.4. Look at the area C' in Quad(p_2) of all points which have the same distance to p as points in C.

Push all sites to the outline of C' as above. Three sites can then be moved into the corners of the outline, one on each of its four sides, and again there can be only one site such that its candidate does not contain a point of the outline, see Figure 2.5.

Exactly the same idea holds for sites in Quad(p_4), since it is symmetric to Quad(p_2) relative to p_1. No candidate of a site in Quad(p_3) can be in conflict with p_1, because it would then automatically contain p. Hence there cannot be a conflict of size $\sigma \leq \sigma_{opt}$ between p_1 and one of the candidates of its k^{th} closest neighbour for $k > 8$. □

2.3 Checking whether there is a solution for a fixed label size σ

Like in Chapter 1 we use the list of conflict sizes to do a binary search for the best possible solution. We do not have to consider any other label size, since the conflict graph does not change inbetween two consecutive interesting conflict sizes. As before, each step of the binary search consists of three phases. Phase I and III are the same as for Heuristic I which turned out to deliver the best results in practice, so these parts are omitted here. Instead, we focus on a new elimination rule which we apply in Phase II, and on how to keep A's approximation guarantee of 50 percent.

2.3.1 The New Elimination Rule for Phase II

We run once through all sites p. The heuristics look at the following three cases:

- If all candidates of p have been eliminated, we stop and return "no solution" to the program which does the binary search on the conflict list.
- If p has candidates free of intersections with other candidates, we choose an arbitrary one of them (say p_i), and eliminate all other candidates p_j of p. Before a candidate's deletion, we do the following updates: we delete its list of overlap information and the symmetric entries stored with all candidates which overlap it.
- If p has only one candidate p_i left, we do the same updates with all candidates q_j which overlap p_i, and then delete them.

Now we add a new rule. Refer to Figure 2.10 and 2.11 to see how this improves the performance of B as compared to Heuristic I. It can easily be shown that Corollary 5 holds for this new rule as well.

- If p has a candidate p_i which overlaps the last two candidates of another site q, then we update and eliminate p_i.

As before we maintain a stack while we run through the sites. On this stack we put all those candidates which now fulfill the same properties as p_i did before, i. e. do not intersect any other squares, or are the last candidates of their sites. Before we look at the next site p, we do all the decisions waiting for us on the stack. Since each of the cases listed above can be detected and handled in constant time, and the number of times they occur is bounded by the number of conflicts, Phase II takes us so far linear time.

At the end of this part of Phase II we are done if all sites have exactly one candidate left. Otherwise we know that candidates of sites with several candidates — call them *active* — never intersect with those that are "the last of their breed", i. e. belong to sites with exactly one square left, because then the former ones would have been eliminated. So it is enough to focus on active candidates from now on. The others are already chosen as part of the solution, and do not interfere with the active ones any more.
As a consequence of Corollary 5 we also know that we have not yet returned "no solution" if there is one of size σ. So we can find a solution with the help of 2-SAT as described in Section 1.2 if no site has more than two candidates left. Otherwise we do a test on a subset of the active candidates as follows.

2.3.2 Keeping the Approximation Guarantee

How can we make sure that B, too, keeps the approximation guarantee of 50 percent? Its main difference from the Approximation Algorithm A is that until the end of Phase II it does not destroy any candidate which might be necessary to construct a solution. A on the other hand already eliminates candidates if they contain a site at twice their size. We take advantage of this approach without risking to end up with a practical behaviour similarly bad as A's.

Definition 8 *A candidate p_i is called* σ–dead *if σp_i contains a site.*

Lemma 9 *Let σ be the current label size. Then after Phase II all sites have at most two candidates left which are not 2σ–dead.*

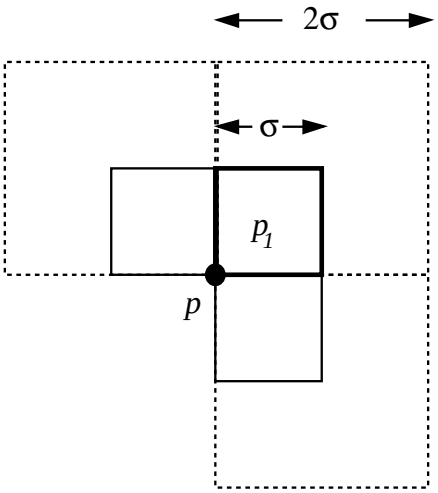

Figure 2.6:

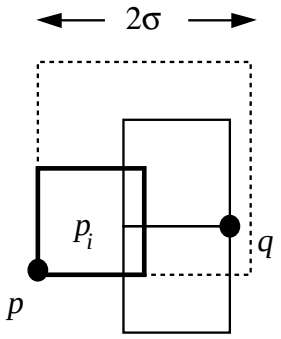

Figure 2.7:

Proof. Suppose p had three such candidates, w. l. o. g. p_1, p_2, and p_4 as indicated in Figure 2.6. If $2\sigma p_1$, $2\sigma p_2$, and $2\sigma p_4$ did not contain any other site, then there could clearly be no candidate in conflict with σp_1. This means that p_1 (or eventually p_2 or p_4) should have already been chosen as part of the solution in Phase II, and p's other candidates eliminated. □

Theorem 10 *We can efficiently construct a solution of size σ from the candidates left after Phase II — if $\sigma \leq \sigma_{opt}/2$.*

Proof. If $\sigma \leq \sigma_{opt}/2$, then there must be a solution π of size 2σ. π is also a solution of size σ, since simultanously shrinking all squares reduces the number of conflicts. Now consider all sites p which have more than one candidate left after Phase II. Let $i = \pi(p)$. We want to show that p_i has not yet been eliminated. To see that, we just have to check the conditions under which this could have happened:

- σp_i cannot contain any other site, because $2\sigma p_i$ does not.
- σp_i cannot overlap the last two candidates of some site q, since $2\sigma p_i$ would then contain q (see Figure 2.7).

- Suppose σp_i overlaps σq_j, and q_j is the last candidate of q (see Figure 2.8). Then $\pi(q) = k \neq j$, otherwise $2\sigma p_i$ and $2\sigma q_{\pi(q)}$ would overlap. The consequence of this is, that q_k also must have overlapped a candidate which is the last of its site, otherwise q_k and therefore p_i would not have been eliminated. Since the number of sites is finite, there must be a candidate r_m in this chain which overlaps a last candidate we have already looked at, say q_j. Then $2\sigma p_i$ and $2\sigma r_m$ must be disjoint, because they are part of solution π. At the same time σq_j must overlap σp_i and σr_m. This means that one corner of σq_j is contained in σp_i, and another one in σr_m. But then automatically the last two corners of σq_j must lie in $2\sigma p_i$ and $2\sigma r_m$ which are disjoint (see Figure 2.9). Obviously one of the corners of σq_j must be q. So q lies in either $2\sigma p_i$ or $2\sigma r_m$. This is a contradiction to the assumption that they are part of solution π.

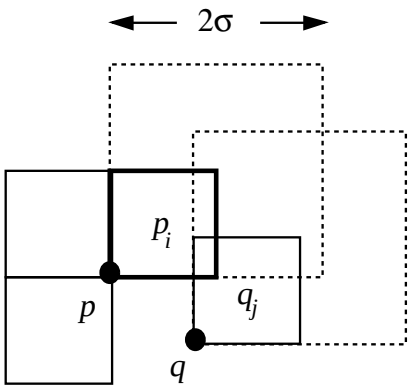

Figure 2.8:

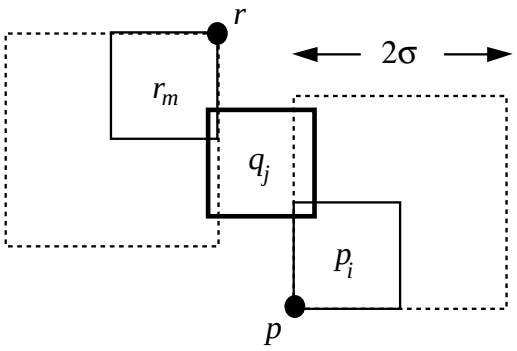

Figure 2.9:

We have just shown that if there is a solution of size 2σ, then all sites with active candidates have kept their candidate which is part of this solution all the way through Phase II. Lemma 9 tells us that after Phase II all sites have at most two squares left which are candidates for such a solution. So we can use 2-SAT on the set of active candidates which are not 2σ–dead. If 2-SAT finds a partial solution for those, we can just add all inactive candidates, and thus receive a solution for all sites, since the sets of active and inactive candidates do not overlap each other.

□

If however 2-SAT returns "no solution", then there cannot be any of size 2σ. That means that $\sigma > \sigma_{opt}/2$. In this case we try to find a solution in Phase III. This 2-SAT test will only be executed until it has returned a negative answer to a problem of size σ_0 to which the heuristics nevertheless found a solution afterwards. We can spare it then, because the binary search for a solution of maximal size continues only on label sizes σ with $\sigma > \sigma_0 > \sigma_{opt}/2$. That means that the test would keep returning negative answers.

Since the first part of Phase II and 2-SAT can be implemented in linear time [EIS76], Phase II has an overall running time of $\mathcal{O}(n)$.

2.4 Running Time Analysis

Phase I, II, and III can all be done in linear time, but since we have to look at $\mathcal{O}(\log n)$ conflict sizes during the binary search for the best solution, we need $\mathcal{O}(n \log n)$ time for Step 2. Finding conflicts in Step 1 takes the same time, so the whole algorithm is in $\mathcal{O}(n \log n)$.

2.5 Experimental Results

Here we focus on plots showing the approximation quality of our algorithms, since the time graphs of B look very similar to those of the heuristics in Figure 1.3 and 1.4, except that B is slower by a constant factor of about five percent which is needed to enforce the new elimination rule. The graphs displaying the number of certain operations as in Figure 1.5 look nearly identical for Approximation Algorithm B.
As in Chapter 1 the results are averaged only over those tests the exact solver managed within the time bound. The standard deviation is represented by the length of the vertical bars in each point of the result plots.

Approximation Quality

The approximation quality is only plotted for dense and hard examples (see Figure 2.10 and 2.11), because on random and real world problems the performance of both B and Heuristic I was almost always 100 percent, see Figure 1.6 and 1.7. On the other two example classes, the improvement of B as compared to the best Heuristic I of about three to five percent is quite evident. This is due to the new elimination rule introduced in Section 2.3.1.
The same remark on the examples for which X did not give a result within the time bound as in Section 1.7.4 should be made here: We did not include these examples in the calculation of the quality plots. But using the upper bound provided by the Approximation Algorithm A, and taking into consideration the typical quality of A, we found out that B's behaviour on those examples does not differ significantly from that on the other examples.

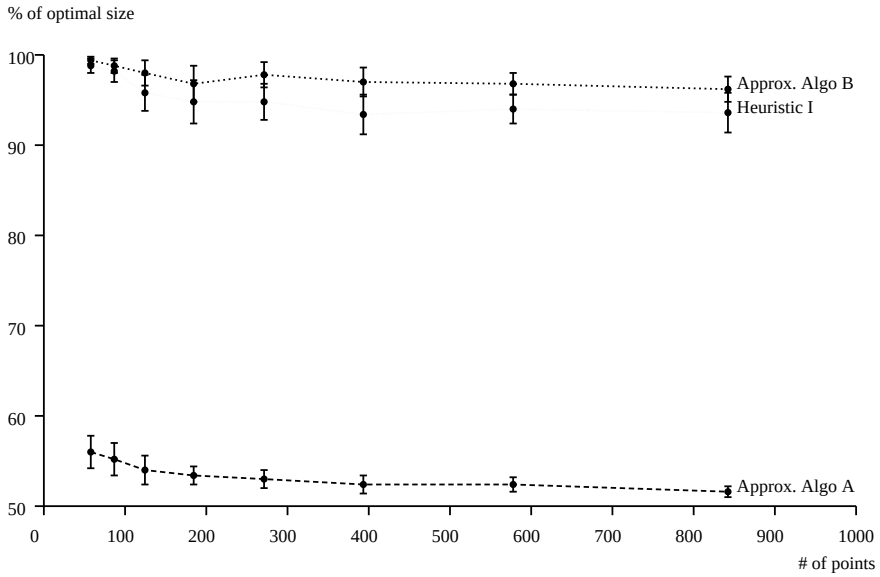

Figure 2.10: Quality of the algorithms on dense examples

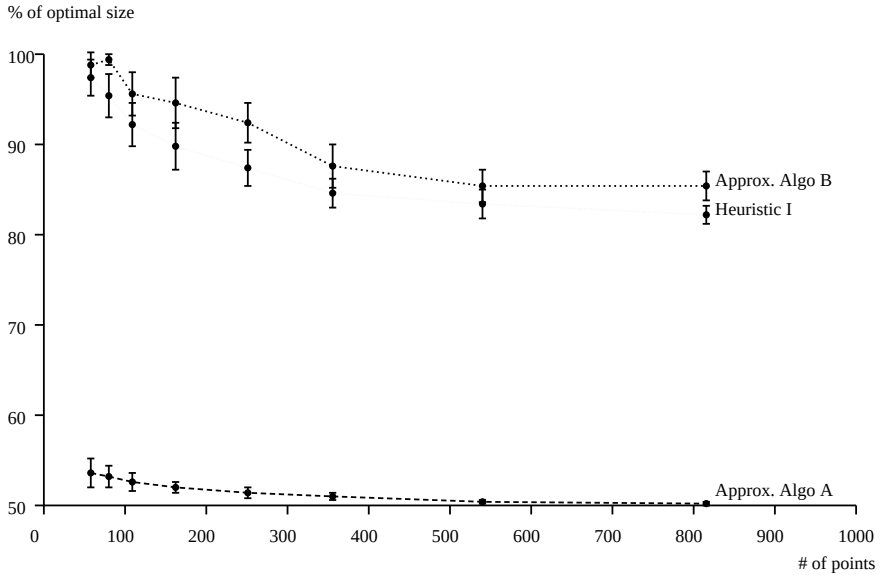

Figure 2.11: Quality of the algorithms on hard examples

Chapter 3

Implementation

The code was written in C++, and we strongly took advantage of data structures and algorithms provided by LEDA [MN95]. The possibilities LEDA offers, helped a great deal to shorten and simplify the code. All heuristics, approximation algorithms, exact solvers, and problem generators described here, can be tested on the World Wide Web under URL

`http://www.inf.fu-berlin.de/~awolff/html/labeling.html`

An example walk through this service is shown in Figures 3.3, 3.4, 3.5, and 3.6. The underlined words are hyperlinks to documents which provide detailed information about the corresponding topic. Most of the references are excerpts of this work transformed into hypertext.

Since implementing and testing was an essential part of the research, we want to give a short description of the problems which arose in that context.

3.1 Data Structures

The data structures we designed ourselves still mostly follow old C conventions, and did not take advantage of the **class** concept supported by C++.

```
const int MAXPOS = 4              // number of possible square positions per point

struct Location                   // coordinates of a point
{
  int x, y;
};
```

Representing coordinates as integers is not only handy, since it avoids rounding problems, but it also makes sense in applications. Experimental data normally has a fixed number of significant digits and can therefore easily be transformed into integers. Internally we double all coordinates to make sure that sizes of conflicts between sites with even and odd coordinates become integers as well. Still the use of integers did have some unexpected side effects — the number of conflict sizes did not grow linearly as expected for large dense and hard problem sets, but reached a maximum of about 200. So in our experiments all the algorithms seemed to have only linear time complexity, see 1.7.4.

```
struct Site                          // information belonging to a point
{
  int index;                         // its index in the input array
  int ord_index;                     // ordered index; used by Exact Solver S
  Location *l;                       // pointer to doubled coordinates
  list <struct Candidate *> *CDL;    // list of candidates not yet eliminated
  Candidate *CDA[MAXPOS];            // array of pointers to all candidates of a site
};
```

list is the LEDA version of doubly linked lists. It offers a wide variety of methods, including iterator stuctures like **forall**(*listelement, list*), but one has to be cautious not to delete the element to which the iterator *listelement* is pointing in a **forall**–loop. `CDL` is filled with all four candidates of a site `s` (stored permanently in `s->CDA[]`) every time before we check whether we can find a solution for a certain conflict size. Then, during Phase I, II, and III they are removed from this list progressively following the elimination rules described in Section 2.3.1. Using such a list proved to be an easy way of accessing exactly those candidates of a site which are still "around".

```
struct Eventpoint                    // for event point queue and sweep line status
{
  Site *s;
  seq_item status;                   // for deleting from sweep line status
  int key;                           // index according to y-coordinate
};
```

This structure is used to build up an event point queue for the line sweeps which have to be performed to get the k closest neighbours of a site in each of the four quadrants relative to it [F92]. The sweep line status uses sorted sequences which are implemented by (2, 4)-trees in LEDA [MN95]. It is essential that the keys are unique, which is a problem because the sites can not be expected to be in general position. So for each of the eight plane sweeps — one for every octant of the plane — the sites have to be ordered anew. This is also due to the fact that the sweep direction changes from sweep to sweep.

```
typedef float Areatype;              // type of the intersection area of two candidates
```

We offer the possibility to use **float** for the intersection area of two candidates, because it might need up to twice as many digits as the maximum coordinate. On the other hand rounding problems will not occur, since even when such areas are added and subtracted, they are never compared to zero. Intersection areas are used by Heuristic J as a means to decide which candidates to eliminate after Phase II. For this purpose all of them are inserted into a priority queue according to the sum of their intersection areas with other candidates, refer to Section 1.5.3. The standard LEDA priority queue is implemented by Fibonacci heaps. It allows insertions and minimum deletions in $\mathcal{O}(\log n)$; while decreasing a key costs only $\mathcal{O}(1)$ [MN95].

```
struct Olinfo                        // overlap information for a pair of candidates
{
  struct Candidate *home;            // one of the overlapping candidates
  Olinfo *overlap;                   // symmetrical info belonging to the other candidate
  Areatype olarea;                   // size of the intersection area
  list_item lhandle;                 // needed for deleting in O(1)
};
```

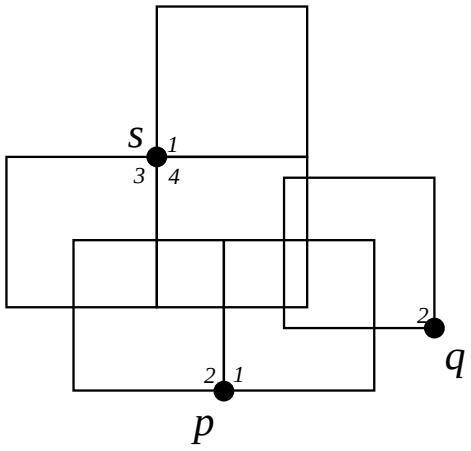

Figure 3.1: Note that a solution like this cannot occur at the end of Phase II. Since q_2 is the last candidate of q, all candidates overlapping it would have been eliminated.

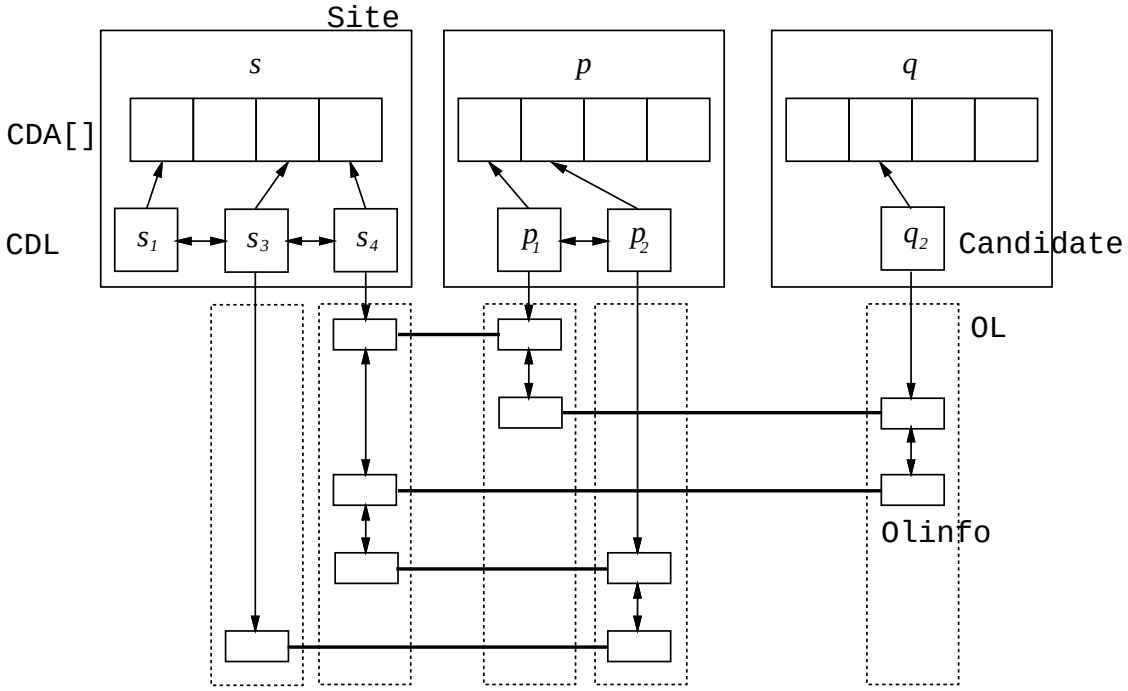

Figure 3.2: Administration of the conflict graph of the situation depicted above with the help of the structures `Site`, `Candidate`, and `Olinfo`. The pairs of instances of `Olinfo` which represent a conflict are linked by bold lines.

`Olinfo` is used to build up and then progressively destruct the conflict graph between overlapping candidates every time we check whether we can find a solution for a certain conflict size. An edge in this conflict graph is represented by a pair of symmetric instances of type `Olinfo` one of which is stored with each of the candidates causing the conflict. For an example, see Figures 3.1 and 3.2. It is here where the size of the intersection area between two overlapping candidates is stored for Heuristic J, while the sum of all these intersection areas of a candidate is stored directly in `Candidate`.

```
struct Candidate                  // 1 of MAXPOS squares belonging to each site
{
  short pos;                      // position ∈ {0, ..., MAXPOS-1}
  bool chosen;                    // ... to be part of the solution currently under construction
  Site *s;                        // site to which the candidate belongs
  int cl_dist;                    // distance of closest site (or 0)
  list <Site *> *Closest;         // closest k=8 sites
  list <Candidate *> *CFL;        // general conflict list
  list <Olinfo *> *OL;            // conflict list for current label size
  Areatype olarea;                // sum of all overlapping areas
  list_item lhandle;              // for deleting from s->CDL
  pq_item pqhandle;               // ... from the priority queue
};
```

This structure is used to bundle all information linked to a candidate. The list of closest sites is computed by `set_closest()`, the general conflict list by `get_conflicts()`, refer to Section 3.2. The list of overlap information `OL` however has to be redone for every step of the binary search for the best solution. We do this in Phase I for every candidate `c` by going through `c->CFL` which has constant length, and deciding which of the candidates listed there overlap `c` for the current label size.

What happens when we eliminate a candidate `c` in Phase I, II, or III? We must empty its list of overlap information `c->OL`. But before we delete one of its elements `ol`, we also get rid of its symmetric entry pointed to by `ol->overlap`, which is stored with the candidate `ol->overlap->home` overlapping `c`. We can quickly look up whether this candidate now fulfills any of the criteria which led to the deletion of `c`. Finally we remove `c` from the list of candidates `CDL` of its site `c->s`. Since the number of conflict partners of a candidate is constant, an elimination can be done in $\mathcal{O}(1)$ with the data structures described above.

3.2 Algorithm

The main funtion is identical for all heuristics and approximation algorithms — for reasons of simplicity we used the new method of detecting conflicts presented in Section 2.2 also for the heuristics.

```
float MAP-LABELING
(
  int which_heur,                          // determines algorithm
  Location L[],                            // locations of the sites
  int n,                                   // number of sites
  int sigma_upper,                         // optional, see Def. 2
  int solution[],                          // for returning the solution
  int & sigma_dead,                        // see Def. 2
  int & dead_point,                        // index of the corresponding site
  int & dead_pos                           // ... and of its candidate
)
{
  list <Site *> SL;                        // list of sites
  list <Eventpoint *> LOS;                 // list of ordered sites
  list <int > C;                           // list of conflict sizes (with multiples)
  int *cfl;                                // array of conflict sizes
  int cflnmb;                              // number of conflict sizes
  float sigma;                             // size of the solution
  int upper_bound;                         // ... for the optimal label size

  init_cand(L, n, SL, LOS, bug);           // O(n)

  set_closest(LOS, 8);                     // O(n log n)

  upper_bound = get_upper_bound(SL, sigma_upper,
        sigma_dead, dead_point, dead_pos); // O(n)

  get_conflicts(SL, upper_bound, C);       // O(n)

  cflnmb = sort_conflicts(C, &cfl);        // O(n log n)

  sigma = bin_search_solution(which_heur,  // O(n log n)
        cfl, cflnmb, SL, LOS, solution);   // (O(n log^2 n) for Heur. J)

  free_memory(cfl, SL, LOS);               // O(n)

  return(sigma);
};
```

In `init_cand()` memory is allocated for all sites, eventpoints, and candidates. Despite the fact that we properly deallocated all of it in `free_memory()`, we frequently ran into an `out of virtual memory` error when we did long test runs. The reason for this phenomenon was that all the little bits of storage we deallocated, were never used again, probably due to internal problems of the C++ gnu-compiler we used. Instead of writing our own memory management routines, we simply

link the test program with a different version of `init_cand()` where arrays of fixed size are used. Allocating only the storage needed for the entries of type `Olinfo` (because their number is harder to predict) did not cause the error.

`set_closest()` is an implementation of the algorithm which determines the k closest neighbours in each of the four quadrants relative to a site as described in [F92]. In Section 2.2 we showed that we need $k = 8$ to detect all interesting conflict sizes, which is done in `get_conflicts()`. This procedure generates a list `C` of all interesting conflict sizes while it figures out all the candidates which will overlap when being blown up continously until their side length reaches some upper bound of the optimal label size.

Just before, by calling `get_upper_bound()` the corresponding variable `upper_bound` is set to σ_{upper} if the parameter `sigma_upper` was handed over by the user, otherwise to σ_{dead}. Afterwards, `sort_conflicts()` sorts `C`, the list of conflict sizes, and puts a copy of each size into an array, thus getting rid of all multiples.

Finally, we do a binary search for the best solution. The parameter `which_heur` of the corresponding procedure `bin_search_solution()` determines which of the algorithms described in Chapter 1 and 2 is to be used.

We return the label size of the best solution found, and the array `solution[]` which holds the indices of the candidates to which the sites are mapped.

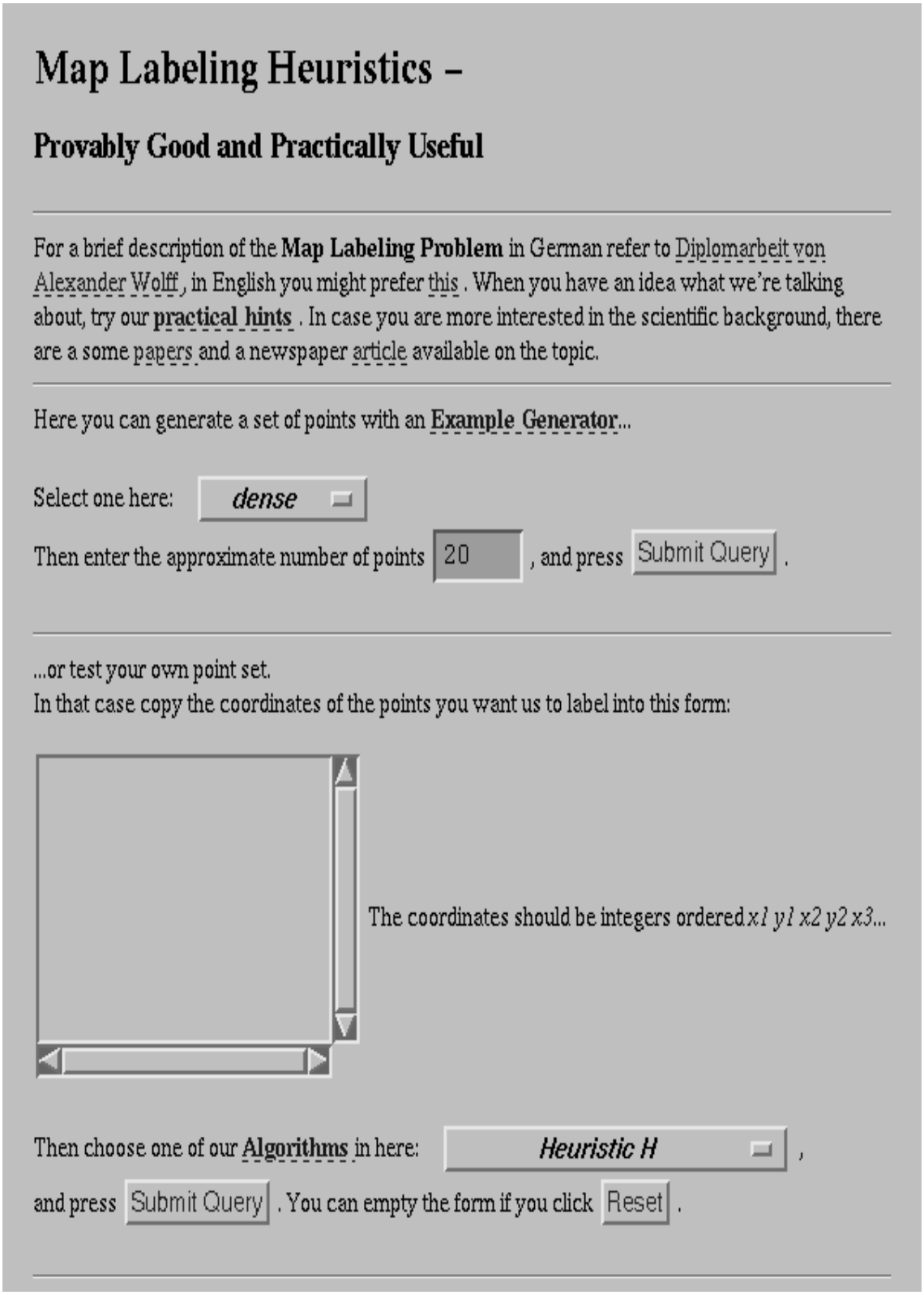

Figure 3.3: The labeling home page

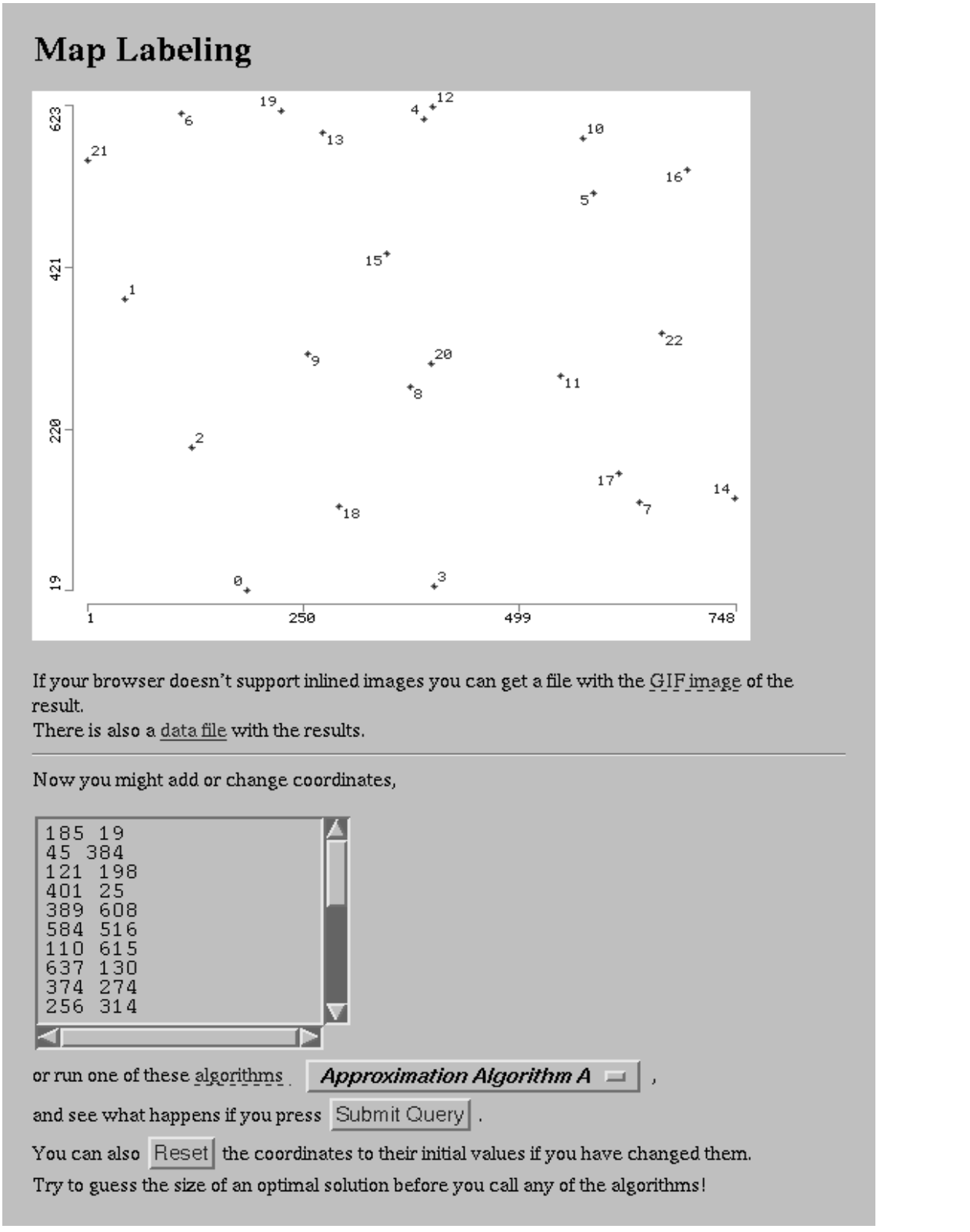

Figure 3.4: An example produced by problem generator dense

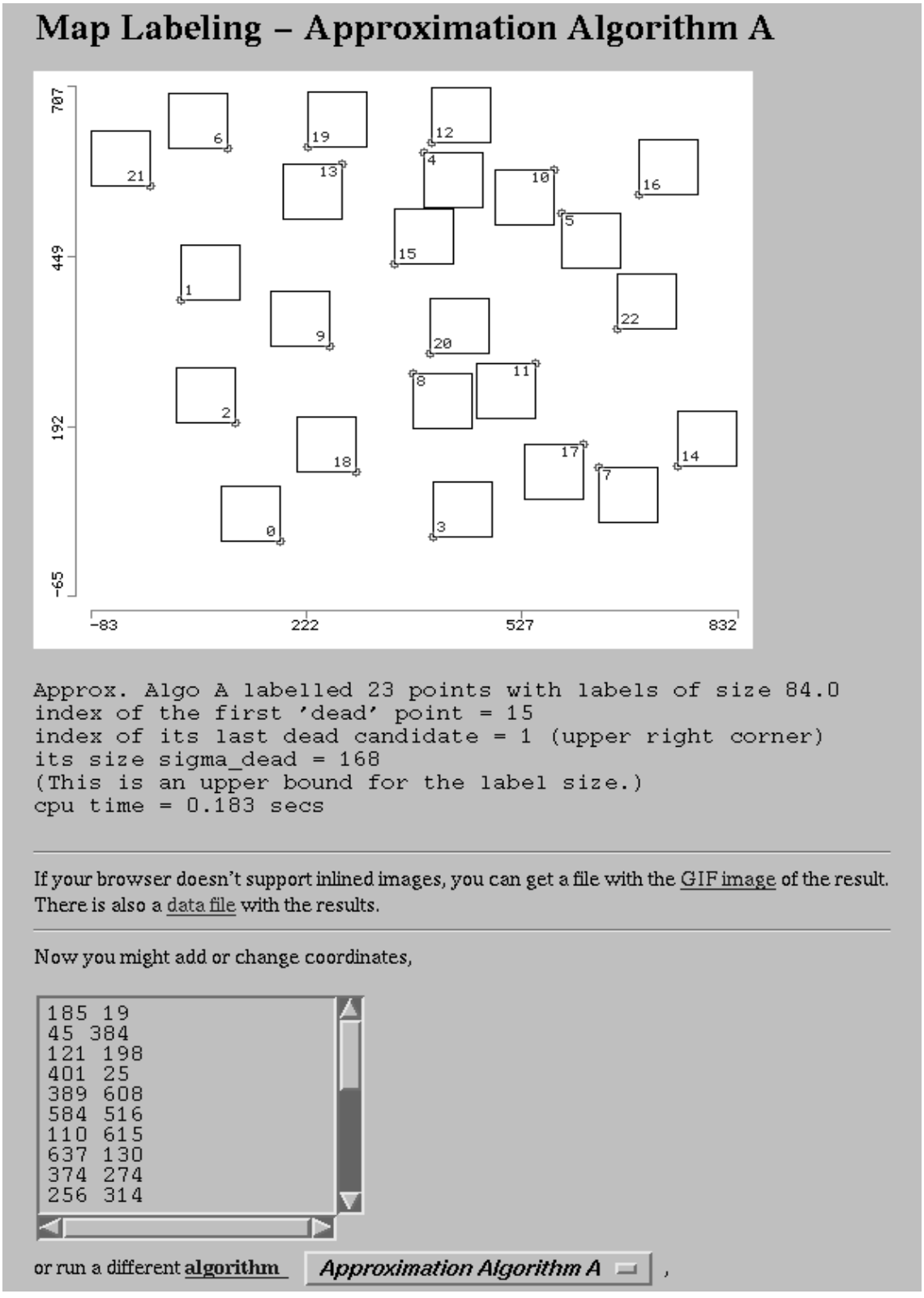

Figure 3.5: Approximation Algorithm A's solution of the example in Figure 3.4

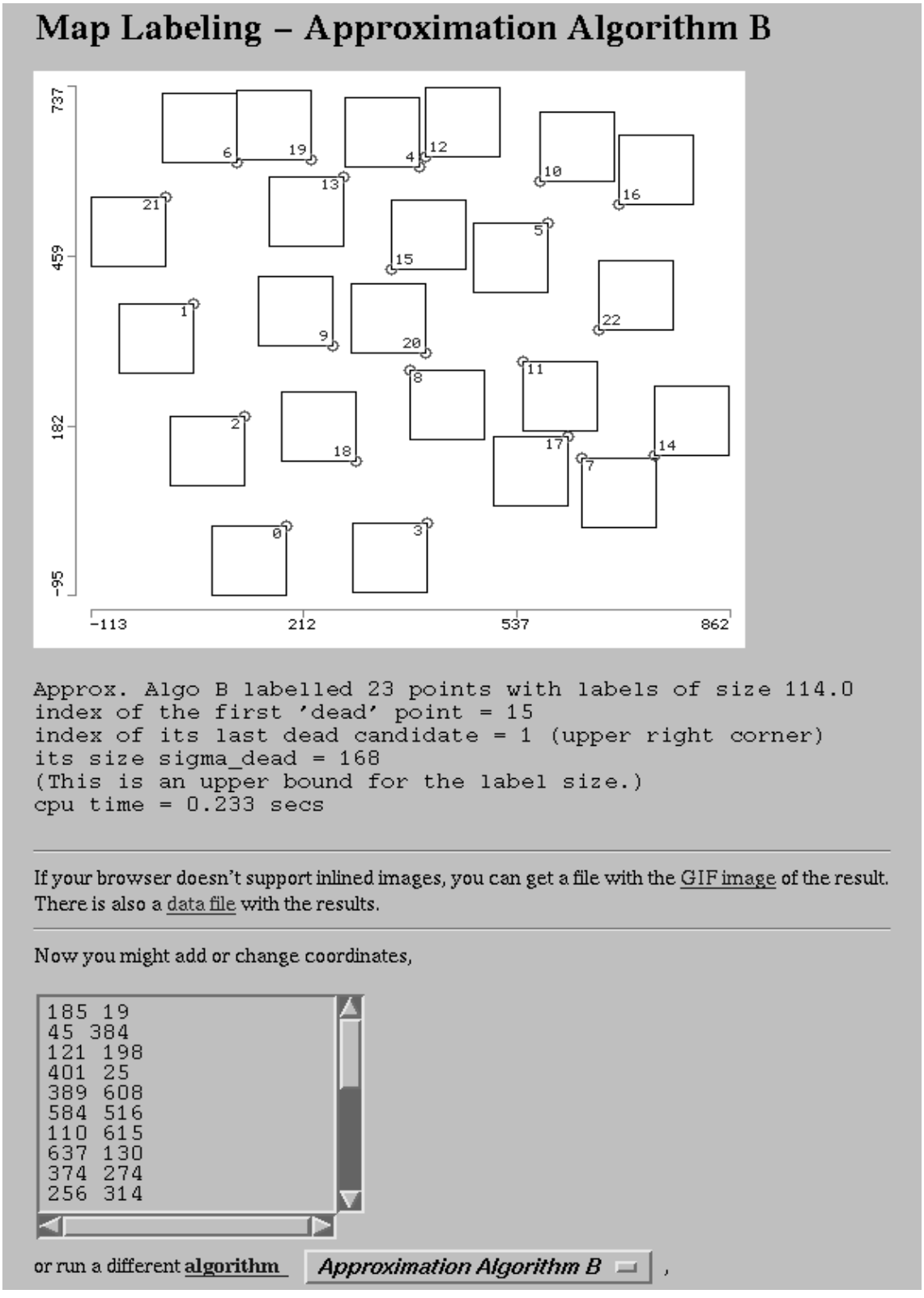

Figure 3.6: Approximation Algorithm B's solution of the example in Figure 3.4

Conclusion

Our experiences with the Map Labeling Problem and its solution can be summed up as follows: Formann and Wagner started with the purely mathematical formulation of the problem which was communicated to them by Kurt Mehlhorn from Saarbrücken, who received the problem from Rudi Krämer of the Amt für Informations- und Datenverarbeitung, Munich. They showed the $\mathcal{NP}$-hardness, and started developing an approximation algorithm when they heard of the practical relevance. They found one, analysed it, and showed its theoretical optimality. In theory the problem was solved perfectly.

Applied to real world data, the algorithm proved useless. Formann and Wagner used their insight into the problem structure gained during the design of A and into the reasons for its practical failure, to develop Heuristic H which produced satisfiably good results. Meanwhile Bettina Preis et. al. developed an Exact Algorithm S which could solve small problems of up to about 80 points, which enabled us to estimate the quality of this heuristic. We improved H to I, and to the even more sophisticated Heuristic J which turned out to be a little worse than our champion I. Erik Schwarzenecker used our heuristical concept to enable the Exact Algorithm X to solve larger problems in reasonable time. He also suggested the class of hard examples. Thus we were able to do a thorough experimental analysis of the quality of our heuristics [WW95a]. We also owe thanks to Stefan Lohrum who helped us to make our heuristics accessible on the World Wide Web.

The next step was the attempt to unite the advantages of the theoretically optimal Approximation Algorithm A with the strength of the heuristics — a much better practical performance. We came up with a new way of detecting conflicts which did not need A's result any more, and with a new rule for eliminating candidates not needed for constructing a solution. The result of these ideas was the Approximation Algorithm B which indeed shares A's theoretical optimality, but delivers even better solutions to our problem sets that the best heuristic did before [WW95b].

Our intense contacts with the practitioners were successful in two respects: We helped them to solve their problems, and they gave us the opportunity to get to know interesting related problems that come up in this context [S95], [WW95c].

Actually Heuristics I and J can be changed by dropping certain elimination rules such that they can be used to maximize the number of sites labelled with labels of a given size. An early version of Heuristic J is being tested by the Amt für Informations- und Datenverarbeitung, Munich, at the moment, by comparing it to the program used there so far (see Figures 1.11 and 1.12). For a more thorough experimental study an exact solver to this variant of the problem will have to be designed and implemented as well.

Bibliography

[AIK89] H. Aonuma, H. Imai, Y. Kambayashi, *A visual system of placing characters appropriatly in multimedia map databases*, Proceedings of the IFIP TC 2/WG 2.6 Working Conference on Visual Database Systems, North Holland (1989) 525–546

[EIS76] S. Even, A. Itai, A. Shamir, *On the complexity of Timetable and Multicommodity Flow Problems*, SIAM Journal on Computing **5** (1976) 691–703

[F92] M. Formann, *Algorithms for Geometric Packing and Scaling Problems*, Dissertation, Fachbereich Mathematik und Informatik, Freie Universität Berlin (1992)

[FW91] M. Formann, F. Wagner, *A Packing Problem with Applications to Lettering of Maps*, Proceedings of the 7th Annual ACM Symposium on Computational Geometry (1991) 281–288

[FW] M. Formann, F. Wagner, *An efficient solution to Knuth's METAFONT labeling problem*, Manuscript (1993)

[I75] E. Imhof, *Positioning Names on Maps*, The American Cartographer **2** (1975) 128-144

[KR92] D. E. Knuth and A. Raghunathan, *The Problem of Compatible Representatives*, SIAM Journal on Discrete Mathematics **5** (1992) 422–427

[MN95] K. Mehlhorn and S. Näher, *LEDA: a platform for combinatorial and geometric computing*, Communications of the ACM **38** (1995) 96–102

[IA86] H. Imai, T. Asano, *Efficient Algorithms for Geometric Graph Search Problems*, SIAM J. Comput. **15** (1986) 478–494

[KMPS93] L. Kučera, K. Mehlhorn, B. Preis, E. Schwarzenecker, *Exact Algorithms for a Geometric Packing Problem*, Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science **665** (1993) 317–322

[S95] V. A. Schmidt, *Reine Forschung, praktische Resultate*, DIE ZEIT (28.4.1995) 45

[W94] F. Wagner, *Approximate Map Labeling is in* $\Omega(n \log n)$, Information Processing Letters **52** (1994) 161–165

[WW95a] F. Wagner, A. Wolff, *Map Labeling Heuristics: Provably Good and Practically Useful*, to appear in: Proceedings of the 11th Annual ACM Symposium on Computational Geometry (1995)

[WW95b] F. Wagner, A. Wolff, *An Efficient and Effective Approximation Algorithm for the Map Labeling Problem*, to appear in: Proceedings of the 3rd Annual European Symposium on Algorithms (1995)

[WW95c] F. Wagner, A. Wolff, *Fast and Reliable Map Labeling*, to appear in: Proceedings of the 9th International Symposium on Computer Science for Environment Protection (1995)

[WKA94] G. Weber, L. Knipping, H. Alt, *An Application of Point Pattern Matching in Astronautics*, Journal of Symbolic Computation **17** (1994) 321–340