SERIE B — INFORMATIK

Map Labeling Heuristics: Provably Good and Practically Useful[◇]

Frank Wagner*
Alexander Wolff**

B 95–04
April 1995

Abstract

The lettering of maps is a classical problem of cartography that consists of placing names, symbols, or other data near to specified sites on a map. Certain design rules have to be obeyed. A practically interesting special case, the *Map Labeling Problem*, consists of placing axis parallel rectangular labels of common size so that one of its corners is the site, no two labels overlap, and the labels are of maximum size in order to have legible inscriptions.

The problem is $\mathcal{NP}$-hard; it is even $\mathcal{NP}$-hard to approximate the solution with quality guaranty better than 50 percent. There is an approximation algorithm A with a quality guaranty of 50 percent and running time $\mathcal{O}(n \log n)$. So A is the best possible algorithm from a theoretical point of view. This is even true for the running time, since there is a lower bound on the running time of any such approximation algorithm of $\Omega(n \log n)$.

Unfortunately A is useless in practice as it typically produces results that are intolerably far off the maximum size.

The main contribution of this paper is the presentation of a heuristical approach that has A's advantages while avoiding its disadvantages:

1. It uses A's result in order to guaranty the same optimal running time efficiency; a method which is new as far as we know.

2. Its practical results are close to the optimum.

The practical quality is analysed by comparing our results to the exact optimum, where this is known; and to lower and upper bounds on the optimum otherwise.

The sample data consists of 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 München.

[◇]This work was done at the Institut für Informatik, Fachbereich Mathematik und Informatik, Freie Universität Berlin, Takustraße 9, 14195 Berlin-Dahlem, Germany. It was supported by the ESPRIT BRA Project ALCOM II.

*wagner@math.fu-berlin.de
**awolff@inf.fu-berlin.de

1 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 [5] 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 München. They have a net of drill-holes spread over the city. The map has to contain the location of these holes and for every hole a block of measuring results such as the concentration of certain chemicals.

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 sizes. By rescaling one of the axes we can assume that the rectangles are squares. An adequate formalization is as follows:

Problem MAP LABELING

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 1 and 2.

Previously [4], we showed by reduction from 3-SAT that the corresponding decision problem is $\mathcal{NP}$-complete. 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, provided that $\mathcal{P} \neq \mathcal{NP}$, no polynomial time approximation algorithm with a quality guaranty better than 50 percent exists. Related results were reported in [1] and [8]. The running time of A is in $\mathcal{O}(n \log n)$. In [10] we showed that there is a matching lower bound on the running time of $\Omega(n \log n)$.

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 the square p was overlapping the 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 [2].

This works only because we make sure that no point has more than two squares left after the elimination phase. On the other hand, we often 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. So we developed a heuristical approach that uses strongly the ideas of A, maintains its quality and running time guaranty, and yields very convincing results. 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

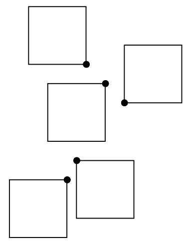

Figure 1: A valid labeling

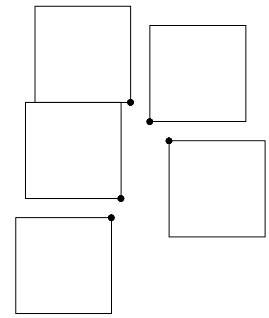

Figure 2: An optimal labeling for the example of Figure 1

phase. In order to handle this, we suggest three different heuristics to bring their number down to two.

The simplest of these heuristics is used by the City of München 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 [11] as a tool to label the output on the screen. With a very similar algorithmic approach we were able to solve the so-called METAFONT labeling problem posed by Knuth and Raghunathan [6].

2 Description of the Heuristics

2.1 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 [4].
$\sigma_{upper} \leq \sigma_{dead}$: 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.*

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 3, $Q := \{z \in \mathbb{R}^2 \mid p - s \preceq z \preceq p + 2s\}$ and

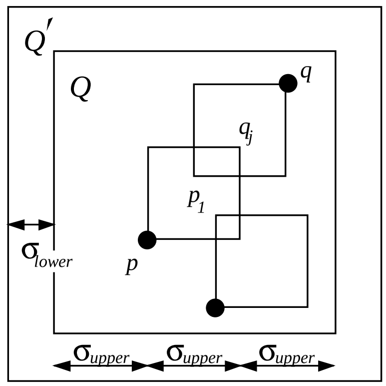

Figure 3:

$Q' := \{z \mid p - \frac{3}{2}s \preceq z \preceq p + \frac{5}{2}s\}$, 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. □

2.2 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}. This takes $\mathcal{O}(n \log n)$ time. What they 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: Make all decisions which do not destroy a possible solution.
   Phase III: For those points which still have two or more "active" candidates left, choose exactly two, and check whether this remaining problem is solvable by 2-SAT, as described in the introduction.

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

2.3 Finding interesting 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_∞–metric, have to be considered. Therefore, we can use a sweep line — or rather, sweep window, approach to determine these conflicts of interest. As usual, we need two data structures: firstly, an event point queue as horizontal structure. This is a queue which holds pointers to the lexicographically ordered sites in the window, that is to all sites of distance at most $2\sigma_{upper}$ left of the sweep line which moves to the right. 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 the y-coordinate.

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 [3].

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 candidates 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 q leaving it on the left side, we first of all 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 (see [7]), but accessing a successor or predecessor of p, or deleting p can then be done in constant time, computing the conflicts between its 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.

2.4 Check whether there is a solution for a fixed label size σ

2.4.1 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. 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 2.3.

2.4.2 Phase II: Making Decisions

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.

2.4.3 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 $\mathcal{O}(\text{n})$ 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.

Since we have to look at $\mathcal{O}(\log n)$ conflict sizes during the binary search for the best solution, these running times sum up to a total of $\mathcal{O}(n \log n)$ for Heuristic H and I, while J takes $\mathcal{O}(n \log^2 n)$ time.

3 Experiments

3.1 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 3.2) 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 [9]. 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.

3.2 Example Generators

Random. We just choose a given number of points uniformly distributed in a rectangle of given size.

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.

Hard. In principle we use the same method as for Dense, that is, trying to place as many squares as possible into a given 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 5.

The choice of these four example generators might be justified by the following considerations. The need for real world data for testing is obvious. Random and Dense are intuitively the first things one would come up with, and differ enough in their behaviour to make them worth looking at. Hard examples might serve as a reminder that we are looking at an $\mathcal{NP}$-complete problem, and that no heuristic can be proved to do better than 50 percent of the optimal solution [4].

3.3 Experimental Set-up

Since the problem generators Dense and Hard ask for a label size σ, while Random and Real World directly use the number of points as input, the problem sizes differ. We run the exact solver, the Approximation Algorithm A, and the heuristics on each of the examples. For every size we averaged the approximation quality and running time over 50 tests.

Actually we do not use σ_{upper} (that is twice the result of A) as an upper bound for the conflicts we have to look at in the heuristics, 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 in any case, especially not for large hard or dense examples where we have the highest number of conflicts per candidate.

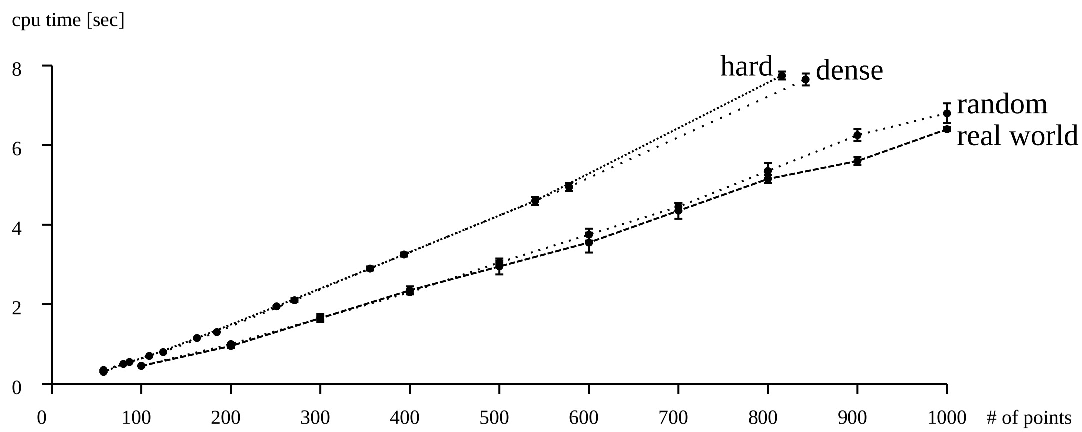

Figure 4: Running time of Heuristic J on different example classes

3.4 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. This on the other hand determines the number of crucial steps, namely finding all interesting conflicts once, and then extracting those valid for a certain σ in every step of the binary search.

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.

3.4.1 Running Time

In Figure 4 we plot the running times of the slowest of the three heuristics, namely J, on the different example sets. H and I are slightly faster. Above 300 points the plot shows a rather stable $\mathcal{O}(n)$-behaviour with very small standard deviation. So far we are neither able to analyse the running time for small dense and hard examples nor to support the empirically linear running time by a theoretical analysis.

3.4.2 Approximation Quality

In Figures 8, 9, 10, and 11, 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 6 and 7. On dense examples the differences between the heuristics become more clearly visible. Heuristic I is the best, yielding results of very high average quality with a slightly larger standard deviation. 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 the 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.

4 Implementation

The implementation of the heuristics follows the structure listed in 2.2. The code was written in C++, and we strongly took advantage of data structures and algorithms provided by LEDA [7]. The commands LEDA offers, helped a great deal to shorten and simplify the code. It was not optimized with respect to running time but rather kept "legible". All heuristics and problem generators can be tested on the WWW under `http://www.inf.fu-berlin.de/~awolff/html/labeling.html`.

5 Conclusion and Acknowledgements

Our experiences with the Map Labeling Problem and its solution can be summed up as follows: We started with the purely mathematical formulation of the problem which was communicated to us by Kurt Mehlhorn from Saarbrücken, who received the problem from Rudi Krämer of the Amt für Informations- und Datenverarbeitung in München. Quickly we showed the $\mathcal{NP}$-hardness, were surprised to hear of the practical relevance, and started developing an approximation algorithm. We found one, analysed it, and showed its theoretical optimality. The problem was solved perfectly—in theory!

Applied to real world data, the algorithm proved useless. We used the insight into the problem structure gained during the design of A and our insight 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 which could solve small problems up to about 80 points, which enabled us to estimate the quality of our 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 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. We also owe thanks to Stefan Lohrum who helped us to make our heuristics accessible on the WWW.

Our intense contacts with the practitioners were successful in two respects: We could solve their problems, and they gave us the opportunity to get to know interesting related problems that come up in this context. We are now adapting our heuristics to these variants of the original problem and hope to be able to solve them with similar success.

References

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

[2] S. Even, A. Itai, A. Shamir, *On the complexity of Timetable and Multicommodity Flow Problems*, SIAM J. Comput. **5** (1976) 691–703

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

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

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

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

[7] K. Mehlhorn, S. Näher, *LEDA, a Library of Efficient Data Types and Algorithms*, TR A 04/89, FB10, Universität des Saarlandes, Saarbrücken, 1989

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

[9] L. Kučera, K. Mehlhorn, B. Preis, E. Schwarzenecker, *Exact Algorithms for a Geometric Packing Problem (Extended Abstract)*, Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science (STACS 93), Lecture Notes in Computer Science **665** (1993) 317–322

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

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

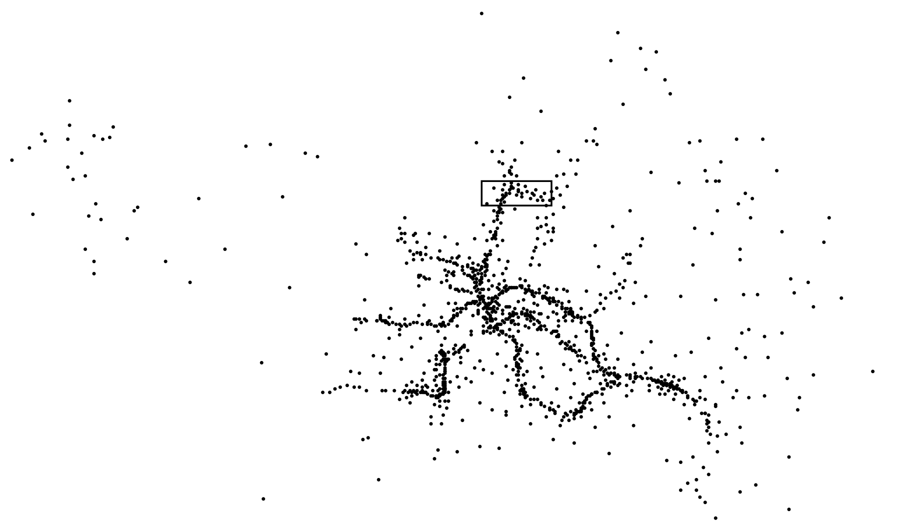

Figure 5: Map showing our sample data from Munich, and the section tested below. There are no conflicts between this section and the rest. The subway lines can be detected easily.

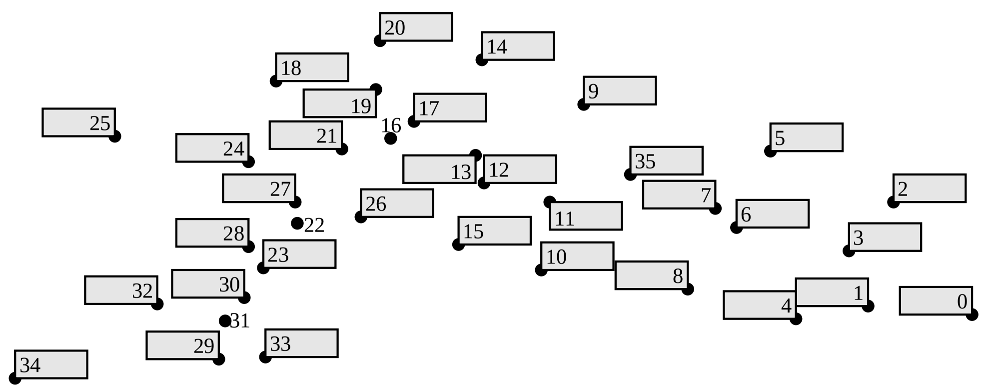

Figure 6: Solution of the program used by the authorities of the City of München before (label height 5000, 3 sites not labelled). It tries to maximize the number of sites labelled for a given size.

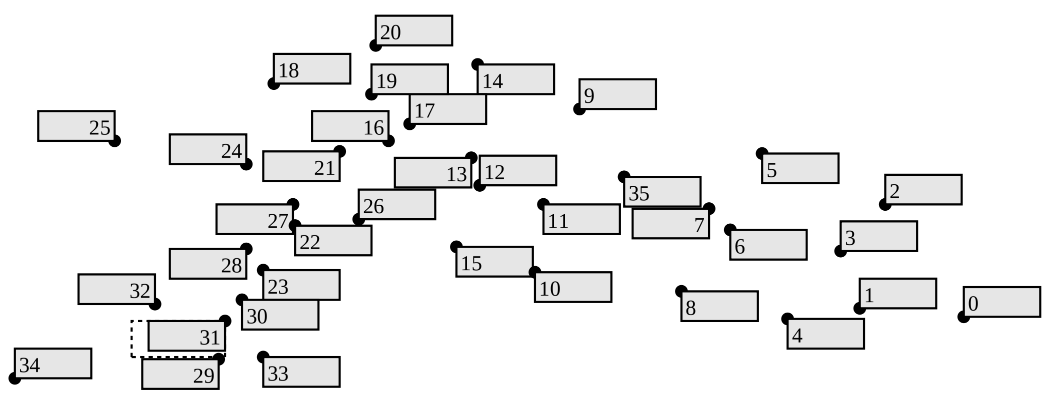

Figure 7: Solution produced by all of our heuristics (label height 5400, optimal). The dashed rectangle shows the candidate with label height $\sigma_{dead} = 6650$.

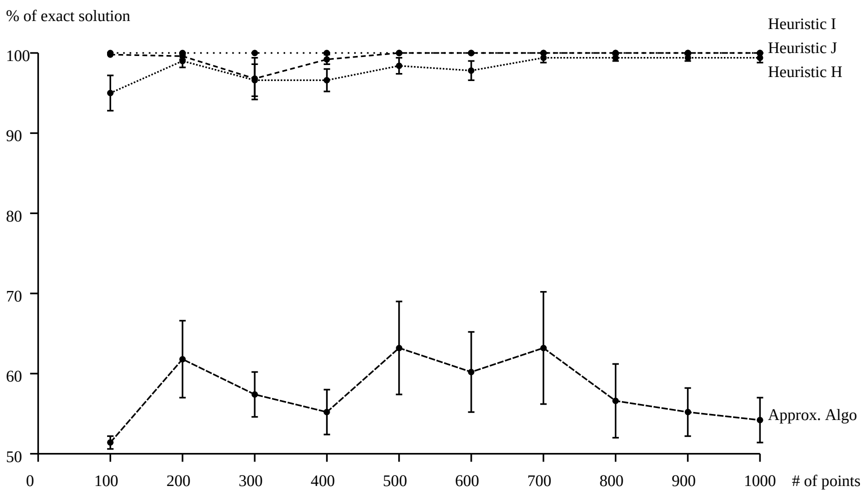

Figure 8: Quality of the heuristics on real world examples

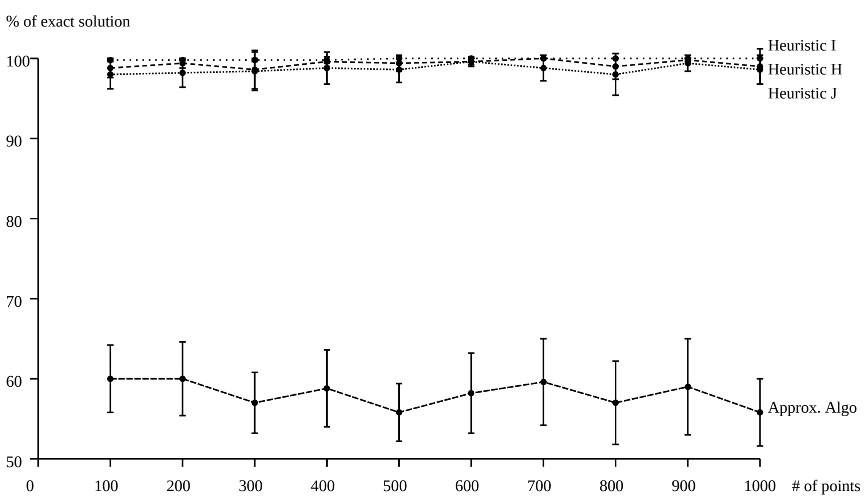

Figure 9: Quality of the heuristics on random examples

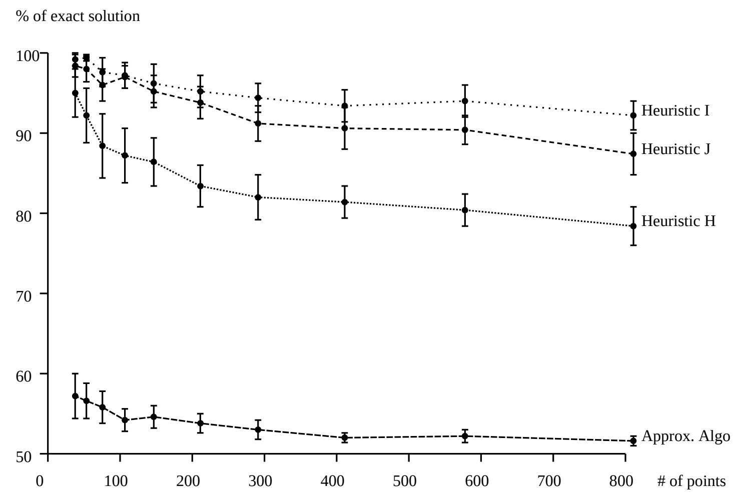

Figure 10: Quality of the heuristics on dense examples

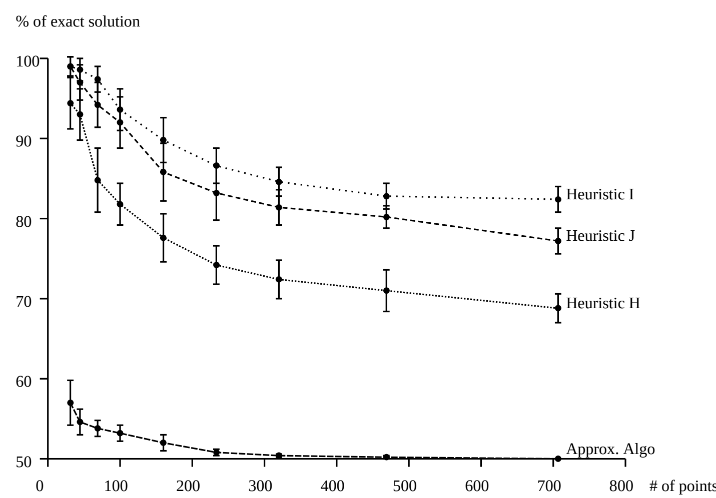

Figure 11: Quality of the heuristics on hard examples