Three Rules Suffice for Good Label Placement[*]

Frank Wagner[§] Alexander Wolff[‡] Vikas Kapoor[§]
Tycho Strijk[**]

Abstract

The general label-placement problem consists in labeling a set of features (points, lines, regions) given a set of candidates (rectangles, circles, ellipses, irregularly shaped labels) for each feature. The problem arises when annotating classical cartographical maps, diagrams, or graph drawings. The size of a labeling is the number of features that receive pairwise nonintersecting candidates. Finding an optimal solution, i.e. a labeling of maximum size, is NP-hard.

We present an approach to attack the problem in its full generality. The key idea is to separate the geometric part from the combinatorial part of the problem. The latter is captured by the conflict graph of the candidates. We present a set of rules that simplify the conflict graph without reducing the size of an optimal solution. Combining the application of these rules with a simple heuristic yields near-optimal solutions.

We study competing algorithms and do a thorough empirical comparison on point-labeling data. The new algorithm we suggest is fast, simple, and effective.

1 Introduction

Label placement is a general problem in information visualization. In cartography the problem is known as the map-labeling problem. Since the first attempts of automating map production, an abundance of approaches has been applied to this problem: expert systems [6], 0–1-integer programming [19], and simulated annealing [2] to name only a few. Map labeling is usually divided into point, line, and area labeling, but the problem can be formulated independently of the features to be labeled. Two interesting subproblems have been studied. In both cases, an instance consists of a set of features and a set of label candidates for each feature.

[*]This work was supported by the Deutsche Forschungsgemeinschaft (DFG) under grants Wa 1066/1-1, 3-1, and 3-2, by the Dutch Organisation for Scientific Research (N.W.O.), and by the ESPRIT IV LTR Project No. 21957 (CGAL).

[§]Institut für Informatik, Fachbereich Mathematik und Informatik, Freie Universität Berlin, Takustraße 9, D-14195 Berlin, Germany. Email {`wagner`|`kapoor`}`@inf.fu-berlin.de`

[‡]Institut für Mathematik und Informatik, Ernst-Moritz-Arndt-Universität Greifswald, Jahnstraße 15a, D-17487 Greifswald, Germany. Email: `awolff@mail.uni-greifswald.de`

[**]Dept. of Computer Science, Utrecht University, The Netherlands. Email: `tycho@cs.uu.nl`

The Label-Size Maximization Problem: Find the maximum factor σ such that each feature gets a label stretched by this factor and no two labels overlap. Compute the corresponding *complete* label placement.

The Label-Number Maximization Problem: Find a maximum subset of the features, and for each of these features a label from its set of candidates, such that no two labels overlap.

The decision versions of both problems are NP-hard in general [5, 4].

In recent years, especially the point labeling problem has achieved some attention in the algorithms' community. Formann and Wagner proposed an approximation algorithm that maximizes the size of uniform axis-parallel square labels. Their algorithm can be applied to technical maps where labels are often numbers or abbreviations of fixed length. It is optimal in respect to both its approximation factor of $\frac{1}{2}$ and its running time of $O(n \log n)$ [4, 14]. Here n refers to the number of points, each of which has four label candidates. For the same problem, there is an algorithm that keeps the theoretical optimality of the approximation algorithm, but performs close to optimal in practice [15].

For square labels of arbitrary orientation and for circular labels there are approximation algorithms for maximising the label size, again under the restriction that all labels are uniform, i.e. of equal size [3]. There is also a bicriteria algorithm that mediates between the two problems mentioned above [3].

The complexity of the label-*number* maximization problem is quite different. Agarwal et al. gave a divide-and-conquer algorithm for placing axis-parallel rectangular labels of arbitrary height and width [1]. The approximation ratio of that algorithm is just $1/O(\log n)$, but the authors also gave an algorithm based on line stabbing that approximates the problem by a factor of $\frac{1}{2}$ in $O(n \log n)$ time if the label height (or width) is fixed. In addition, they presented a polynomial time approximation scheme (PTAS) for the same setting.

If fixed-height rectangular labels are allowed to touch their points anywhere on the rectangle boundary, there is still a PTAS and an $O(n \log n)$ algorithm that guarantees to label at least half the number of points labeled in an optimal solution [13].

Kakoulis and Tollis suggested a more general approach to labeling [9]. They compute the candidate conflict graph and its connected components. Then they use a heuristic similar to the greedy algorithm for maximum independent set to split these components into cliques. Finally they construct a bipartite "matching graph" whose nodes are the cliques of the previous step and the features of the instance. In this graph, a feature and a clique are joined by an edge if the clique contains a candidate of the feature. A maximum-cardinality matching yields the labeling. Depending on how the clique check and the matching algorithm is implemented, their algorithm takes $O(k\sqrt{n})$ or even $O(kn)$ time, where k refers to the number of conflicts. The authors do not give any time bounds.

The algorithm for label-number maximization we present in this paper is the first that combines all of the following characteristics that make it especially suitable for fast internet applications. Our algorithm

- does not depend on the shape of labels,
- can be applied to point, line, or area labeling (even simultaneously) if a finite set of label candidates has been precomputed for each feature,
- is easy to implement,
- runs fast, and
- returns good results in practice.

The input to our algorithm is the conflict graph of the label candidates. The algorithm is divided into two phases similar to the first two phases of the algorithm for label-size maximization described in [15]. In phase I, we apply a set of rules to all features in order to label as many of them as possible and to reduce the number of label candidates of the others. These rules do not destroy a possible optimal placement. Then, in phase II, we heuristically reduce the number of label candidates of each feature to at most one.

While we treat the label-placement problem essentially as a maximum independent set problem, it can also be viewed as a *constraint satisfaction* problem [16], a problem class discussed in the artificial intelligence community, or as a problem of *compatible representatives*, which was independently introduced into the discrete mathematics community by Knuth and Raghunathan [10]. For a broader discussion of the links between constraint satisfaction and label placement, see [17].

This paper is structured as follows. In Section 2 we give the details of our two-phase algorithm. In Section 3 we describe the set-up and in Section 4 we give the results of our experiments. We compare our algorithm on point-labeling data to five other methods, namely simulated annealing, a greedy algorithm, two variants of the matching heuristic of Kakoulis and Tollis, and a hybrid algorithm that combines our rules with their clique matching.

Part of the examples, on which we do the comparison, are benchmarks that were already used in [15]. We added examples for placing rectangular labels of varying size, both randomly generated and from real world data. Our samples come from a variety of sources; they include the location of some 19,400 groundwater drill holes in Munich, 373 German railway stations, and 357 shops. The latter are marked on a tourist map of Berlin that is labeled on-line by our algorithm. The algorithm is also used by the city authorities of Munich to label drill-hole maps. All synthetic and real world data is available via the Internet.[1] Our tests differ from experiments performed by other researchers [7, 2, 13, 9] in that we included synthetic instances which are very dense but still have a complete labeling.

2 Algorithm

Our algorithm consists of two phases. In phase I we apply a set of rules to all given features in order to label as many of them as possible and to reduce the

[1] `http://www.math-inf.uni-greifswald.de/map-labeling/general`

number of label candidates of the others. These rules do not reduce the size of an optimal labeling, i.e. the maximum number of features that can be labeled simultaneously. Then, in phase II, we heuristically reduce the number of label candidates of each feature to at most one.

2.1 Phase I

In the first phase, we apply all of the following rules to each of the features. Let p_i be the i-th label candidate of feature p. For each of the rules we supply a sketch of a typical situation in the context of point labeling. We use four rectangular, axis-parallel label candidates per point, namely those where one of the label's corners is identical to the point. We shaded the candidates that are chosen to label their point, and we used dashed edges to mark candidates that are eliminated after a rule's application.

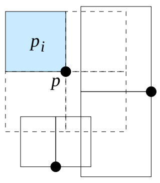


Figure 1: Rule L1.

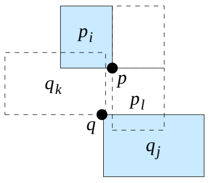


Figure 2: Rule L2.

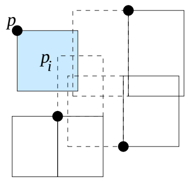


Figure 3: Rule L3.

(L1) If p has a candidate p_i without any conflicts, declare p_i to be part of the solution, and eliminate all other candidates of p, see Figure 1.

(L2) If p has a candidate p_i that is only in conflict with some q_k, and q has a candidate q_j $(j \neq k)$ that is only overlapped by p_l $(l \neq i)$, then add p_i and q_j to the solution and eliminate all other candidates of p and q, see Figure 2.

(L3) If p has only one candidate p_i left, and the candidates overlapping p_i form a clique, then declare p_i to be part of the solution and eliminate all candidates that overlap p_i, see Figure 3.

We want to make sure that these rules are applied exhaustively. Therefore, after eliminating a candidate p_i, we check recursively whether the rules can be applied in the neighborhood of p_i, i.e. to p or to the features of the conflict partners of p_i. As stated above, our rules are conservative in the following sense.

Lemma 1 *If there is an optimal solution of size t for the given instance before applying any of rules L1 to L3, then there is still an optimal solution of size t after applying one of these rules.*

Proof. Assume to the contrary that the size of the optimal solution decreases after we remove a candidate p_m of the feature p. Then every optimal solution π

before the elimination must have assigned p_m to p. Consider the circumstances under which p_m can be removed:

- There is a candidate $p_i \neq p_m$ of p that does not intersect any other candidate (see rule L1). Then we could replace p_m by p_i in π.
- There is a candidate $p_i \neq p_m$ of p and a feature q with a candidate q_j, and p_i and q_j behave as stated in rule L2. Then we could replace p_m and $\pi(q)$ by p_i and q_j since these do not intersect candidates of features other than p and q.
- p_m intersects a candidate q_i who is the last candidate of the feature q, and the candidates intersecting q_i (including p_m) form a clique (see rule L3). Then π does not label q at all. Instead of labeling p with p_m we could also label q with q_i since all candidates intersected by q_i are also intersected by p_m.

In each case we would get a solution of the same size as π. Contradiction. ◻

2.2 Phase II

If we have not managed to eliminate all conflicts and to reduce the number of candidates to at most one per feature in the first phase, then we must do so in phase II. Since phase II is a heuristic, we are no longer able to guarantee optimality. The heuristic is conceptionally simple and makes the algorithm work well in practice, see Section 3. The intuition is to start eliminating troublemakers where we have the greatest choice. Spoken algorithmically, we go through all features p that still have the maximum number of candidates . For each p we delete the candidate with the maximum number of conflicts among the candidates of p. This process is repeated until each feature has at most one candidate left and no two candidates intersect. These candidates then form the solution.

As in phase I, after eliminating a candidate, we check recursively whether our rules can be applied in its neighborhood. For the pseudocode of the complete algorithm, refer to Figure 4.

2.3 Analysis

In order to simplify the analysis of the running time, we assume that the number of candidates per feature is constant and that we are given the candidate conflict graph in the usual adjacency-list representation. Then it is easy to see that in phase I, rules L1 and L2 can be checked in constant time for each feature. We use a stack to make sure that our rules are applied exhaustively. After we have applied a rule successfully and eliminated a candidate, we put all features in its neighborhood on the stack and apply the rules to these features. Since a feature is only put on the stack if one of its candidates was deleted or lost a conflict partner, this part of phase I sums up to $O(n + k)$ time, where n is the number of candidates and k is the number of pairs of intersecting candidates

```
RULES(features $F$, candidates $C_f$ for each $f \in F$, cand. conflict graph $G$)

// phase I
apply rules L1 to L3 exhaustively to all features

// phase II
while there are intersecting candidates do
    $f \leftarrow$ feature with maximum number of candidates in $F$
    delete candidate $c$ of $f$ with maximum number of conflicts in $C_f$
    apply rules L1 to L3 exhaustively, starting in the neighborhood of $c$
end
```

Figure 4: Pseudocode for our general label-placement algorithm.

in the instance, i.e. the number of edges in the candidate conflict graph. For rule L3, we have to check whether a candidate is intersected by a clique. In general, this takes time quadratic in the number of conflict partners. Falling back on geometry, however, can help to cut this down. In the case of axis-parallel rectangular labels for instance, a clique can be detected in linear time by testing whether the intersection of all conflicting rectangles is not empty. A simple charging argument then yields $O(k^2)$ time for checking rule L3.

We can check rule L3 even in constant time if we apply it only to candidates with less than a constant number of conflicts. This makes sense since it is not very likely that the neighborhood of a candidate with many conflicts is in fact a clique. In this case, phase I can be done in $O(n + k)$ time.

In phase II we can afford simply to go through all features sequentially and check for each feature f whether f has the current maximum number of candidates. If so, we go through the candidates of f and determine the one with the maximum number of conflicts. The amount of time needed to delete this candidate and apply our rules has already been taken into account in phase I. Thus phase II needs only linear extra time.

Putting things together, we get an $O(n + k^2)$ algorithm if rule L3 can be checked in linear time, and an $O(n + k)$ algorithm if we allow only constant effort for checking rule L3. In our experiments, we have not bounded this effort, yet this part of the algorithm showed a linear-time behavior.

For axis-parallel rectangular labels, the conflict graph can be determined in $O(k + n \log n)$ time. Thus applied to point labeling, our algorithm can label n points in $O(k + n \log n)$ total time, given a constant number of axis-parallel rectangular label candidates per point and constant effort for checking rule L3.

2.4 Priorities

An important topic we have neglected so far are priorities. Both features and their label candidates can be given priorities. Priorities for label candidates usually express aesthetical preferences. For example, when labeling points, cartographers prefer the label position to the upper right of a point to all other positions [8, 18].

Both kinds of priorities can be incorporated into our algorithm. When using feature priorities, we only have to restrict rule L3. Rule L3 can only be applied if feature p has the highest priority among the features whose candidates form a clique in the candidate conflict graph.

When using candidate priorities, rules L1 and L3 must be restricted such that only those candidates are eliminated whose priority does not exceed that of p_i. Restricting rule L2 depends on which function f of the priorities is to be maximized. Usually f will simply be the sum of the priorities of all label candidates in the solution. When applying rule L2, only those candidates of p and q can be eliminated that do not allow a higher value of the restriction of f to the points p and q compared with the candidate pair p_i and q_j.

Restricting our rules as above may reduce the number of features that are labeled successfully in phase I and thus leave more work to the heuristic in phase II. At least for the case of feature priorities, an indication that this is not as much of a problem as one might suspect can be found in Chapter 3 of [17]. There our algorithm is compared with two variants, EI+L3 and EI-1*, one with and one without rule L3. Only for very dense examples does the variant without rule L3 perform significantly worse than its competitors.

2.5 A Hybrid Algorithm

Since the decisions our algorithm makes in phase II are only based on local properties of the candidate conflict graph, these decisions can be made very efficiently. Using more global information is time-costly, but might also improve the quality of the results. Therefore we thought that it would be interesting to combine our set of rules with the global aspect of the matching heuristic of Kakoulis and Tollis [9]. The resulting hybrid algorithm proceeds as follows.

As before, we compute the candidate conflict graph. In phase I, we again apply our set of rules exhaustively. In the new phase II, however, we use the heuristic proposed by Kakoulis and Tollis to break up the connected components of the conflict graph into cliques. Recall that in every connected component, they determine the candidate with the highest degree, eliminate it, and repeat this process recursively until each component is a clique. The choice of the candidate that is to be eliminated has the following exception. If the candidate belongs to a feature that has “very few” candidates left, then the candidate with the second highest degree in the current connected component is eliminated.

Like in the old phase II, after each deletion, we apply our rules in the neighborhood of the eliminated candidate in order to propagate the effect of our heuristical decision.

As soon as all connected components are cliques, we use a maximum-cardinality bipartite-matching algorithm to match as many cliques with features as possible.

The new phase II can be implemented by an extended breadth-first search (BFS). First, we compute all connected components of the conflict graph by common BFS. At the same time, we store the candidate with the highest, second-highest and the lowest degree for each component. To decide whether a component C is a clique, we simply check whether the vertex with minimum degree in C matches the number of vertices in C minus one. If this is not the case, we delete the vertex v_1 with the highest degree. There is one exception. Let a vertex be *important* if it corresponds to a candidate that is the last candidate of a feature. If v_1 is important and the vertex v_2 with the second highest degree in C is not important, then we delete v_2 instead of v_1.

Now let v be the vertex we deleted. We apply our rules in the neighborhood of v and then apply the extended BFS recursively to all vertices that were adjacent to v just before its deletion. In each level of the recursion at least one vertex is deleted, and each edge is considered at most twice by BFS. Thus, if each of the n features has a constant number of candidates, we have at most $O(n)$ recursion levels and each takes at most $O(k)$ time. This results in $O(nk)$ time for the new phase II compared with $O(n+k)$ for the previous version. Computing a maximum-cardinality bipartite matching takes $O(k\sqrt{n})$ in practice.

3 Experiments

We compare our algorithm **Rules** with the following five other methods on point-labeling data. We implemented all algorithms in C++.
Annealing is a simulated-annealing algorithm based on the experiments by Christensen et al.. We follow their suggestions for the initial configuration, the objective function, a method for generating configuration changes, and the annealing schedule [2]. In order to save time, we allowed only 30 instead of the proposed 50 temperature stages in the annealing schedule. This did not seem to influence the quality of the results.
Greedy repeatedly picks the leftmost label (i.e. the label whose right edge is leftmost), and discards all candidates that intersect the chosen label. This simple algorithm has an approximation factor of $1/(H+1)$, where H is the ratio of the greatest and the smallest label height [12]. Greedy can be implemented to run in $O(n \log n)$ time. However, our $O(n^2)$-implementation is simply based on lists and uses brute force to find the next leftmost label candidate.
Matching refers to the "pure" matching heuristic of Kakoulis and Tollis [9]. Our implementation uses the recursive extended BFS of Section 2.5 and the maximum-cardinality bipartite-matching algorithm supplied by LEDA [11]. It runs in $O(kn)$ time. We did not apply any of our rules and did not do any pre- or post-processing. The post-processing sketched in [9] consists of moving labels locally or allowing partial overlap of labels with features or other labels. This would contradict our four-position labeling model and thus make a comparison with the other algorithms impossible.
Matching+L1 is a variant of their algorithm, also proposed in [9]. Here rule L1 is applied exhaustively before the heuristic that reduces all connected components to cliques. This does not change the asymptotic runtime behavior.
Hybrid refers to the algorithm sketched in Section 2.5. It combines the heuristic

by Kakoulis and Tollis that reduces connected components to cliques with our rules. Again, our implementation uses LEDA's $O(kn)$-time matching algorithm.

We run the algorithm described above on the following example classes. In Appendix B an instance of each class is depicted. The numbers in parentheses refer to the number of points that have been labeled by Rules.

RandomRect. We choose n points uniformly distributed in a square of size $25n \times 25n$. To determine the label size for each point, we choose the length of both edges independently under normal distribution, take its absolute value and add 1 to avoid nonpositive values. Finally we multiply both label dimensions by 10.

DenseRect. Here we try to place as many rectangles as possible on an area of size $\alpha_1\sqrt{n} \times \alpha_1\sqrt{n}$. The factor α_1 is chosen such that the number of successfully placed rectangles is approximately n, the number of points asked for. We do this by randomly selecting the label size as above and then trying to place the label up to 50 times. If we do not manage to place the label without intersection, we select a new label size and repeat the procedure. If none of 20 different sized labels could be placed, we assume that the area is well covered, and stop. For each rectangle we placed successfully, we return its height and width and a corner picked at random. It is clear that all points obtained this way can be labeled by a rectangle of the given size without overlap.

RandomMap and **DenseMap** try to imitate a real map using the same point placement methods as RandomRect and DenseRect, but more realistic label sizes. We assume a distribution of 1:5:25 of cities, towns, and villages. After randomly choosing one of these three classes according to the assumed distribution, we set the label height to 12, 10 or 8 points accordingly. The length of the label text then follows the distribution of the German railway station names (see below). We assume a typewriter font and set the label length to the number of characters times the font size times $\frac{2}{3}$. The multiplicative factor reflects the average ratio of character width to height.

VariableDensity. This example class is used in the experimental paper by Christensen et al. [2]. There the points are distributed uniformly on a rectangle of size 792×612. All labels are of equal size, namely 30×7. We included this benchmark for reasons of comparability.

HardGrid. In principle we use the same method as for Dense, that is, trying to place as many labels as possible into a given area. In order to do so, we use a grid of $\lfloor \alpha_2\sqrt{n} \rfloor \times \lceil \alpha_2\sqrt{n} \rceil$ square cells of edge lengths n. Again, α_2 is a factor chosen such that the number of successfully placed squares is approximately n. In a random order, we try to place a square of edge length n into each of the cells. This is done by randomly choosing a point within the cell and putting the lower left corner of the square on it. If it overlaps any of the squares placed before, we repeat at most 10 times before we turn to the next cell.

RegularGrid. We use a grid of $\lfloor \sqrt{n} \rfloor \times \lceil \sqrt{n} \rceil$ squares. For each cell, we randomly choose a corner and place a point with a small constant offset near the chosen corner. Then we know that we can label all points with square labels of the size of a grid cell minus the offset.

MunichDrillholes. The municipal authorities of Munich provided us with the coordinates of roughly 19,400 ground-water drill holes within a 10×10 kilometer square centered approximately on the city center. From these points, we randomly pick a center point and then extract a given number of points closest to the center point according to the L_∞-norm. Thus we get a rectangular section of the map. Its size depends on the number of points asked for. The drill-hole labels are abbreviations of fixed length. By scaling the x-coordinates, we make the labels into squares and subsequently apply an exact solver for label-size maximization. This gives us an instance with a maximal number of conflicts which can just be labeled completely.

In addition to these example classes, we tested the algorithms on the maps depicted in Figures 12 and 13, see Appendix C.

4 Results

We used examples of 250, 500, ..., 3000 points. For each of the example classes and each of the example sizes, we generated 30 files. Then we labeled the points in each file with axis-parallel rectangular labels. We used four label candidates per point, namely those where one of the label's corners is identical to the point. We allowed labels to touch each other but not to obstruct other features.

4.1 Quality

The graphs in Appendix A (Figures 8 to 11) show the performance of the six algorithms. The average example size is shown on the x-axis, the average percentage of labeled points is depicted on the y-axis. Note that we varied the scale on the y-axis from graph to graph in order to show more details. To improve legibility, we give two graphs for each example class; on the left the results of Rules, Annealing, and Hybrid are depicted, while those of Greedy and the two variants of Matching are shown in the graph on the right of each figure. The results on a single example size, namely approximately 2500 points, are shown in Figure 5. There, the worst and the best performance of the algorithms are indicated by the lower and upper endpoints of the vertical bars.

Due to lack of space, we only present graphs of half of our example classes. The results on the other classes can be found on our web site. Here we summarize them briefly. On RandomMap, all algorithms performed about 12% worse than on RandomRect. The results on DenseRect were nearly identical to those on DenseMap. On RegularGrid and on MunichDrillholes, all algorithms labeled between 99 and 100% of the input points.

The example classes are divided into two groups; those that have a complete labeling and those that do not. For the former group, the percentage of labeled points expresses directly the performance ratio of an algorithm. For examples of the latter group, which consists of RandomRect, RandomMap and VariableDensity, there is only a very weak upper bound for the size of an optimal solution, namely the number of labels needed to fill the area of the bounding box of the instance completely. Thus for VariableDensity at most 2539 points can possibly

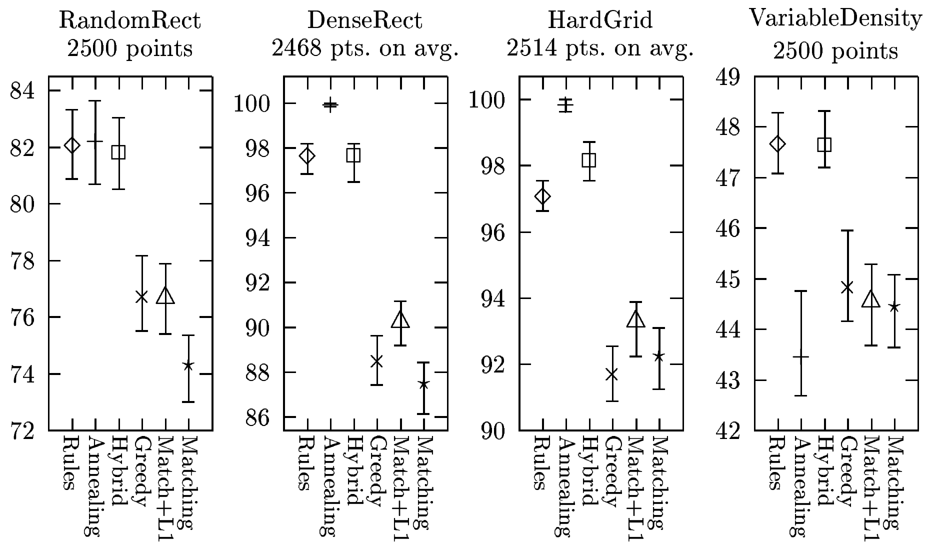


Figure 5: Average performance on examples of approx. 2500 points

be labeled. Experiments we performed with an exact solver on examples of up to 200 points showed that on an average about 85% of the points in an instance of RandomRect and usually less than 80% in the case of RandomMap can be labeled. Other than VariableDensity, these classes are designed to keep their properties with increasing point number. This is reflected by the fact that the algorithms' performance was nearly constant on these examples. We used the same set of rules as in phase I of our algorithm to speed up the exact solver.

In terms of performance the algorithms can be divided into two groups. The first group consists of simulated annealing, our rule-based algorithm, and the new hybrid algorithm; the second group is represented by the greedy method and the two variants of the matching heuristic. The first group outperforms the second group clearly in all but one example class. On RegularGrid data, the second group and Hybrid achieve 100%, followed very closely (99.2%) by Rules and Annealing. For all example classes (except RegularGrid and MunichDrill-holes, where all algorithms performed extremely well), there is a 5–10% gap between the results of the two groups.

For all examples that have a complete labeling, Rules and Hybrid label between 95 and 100% of the points. Experiments with the above-mentioned exact solver on small examples hint that the same holds for RandomRect and RandomMap examples. For some of the example classes, simulated annealing outperforms our algorithms by 1–2%. However, in order to achieve similarly good results, simulated annealing needs much longer (see below), in spite of the fact that it uses the same fast $O(n \log n)$-time algorithm for detecting rectangle intersections (based on an interval tree). We were astonished to see that Hybrid and Rules yield practically identical results in spite of their different approaches. Only on HardGrid and RegularGrid data sets Hybrid was better than Rules — by merely 1%. The similarity of their results suggests that it is the rules that do most of the work.

In the second group, the greedy algorithm performed well given that it makes its decisions only based on local information. Surprisingly, its results were practically always better that of the "pure" Kakoulis–Tollis heuristic that relies on a global matching step. Adding rule L1 as a pre-processing step improved the result of the matching heuristic by up to 3%. This variant performed better than the greedy algorithm in most example classes, but was still clearly worse than simulated annealing and our algorithms except on RegularGrid.

4.2 Time

In Figures 6 and 7 we present the running times of our implementations in CPU seconds on a Sun UltraSparc. We used the SUN-Pro compiler with optimizer flag `-fast`.

We show the two example classes where simulated annealing performed most slowly and fastest. All other graphs resemble that of MunichDrillholes, except HardGrid where the running times are distributed similarly to VariableDensity.

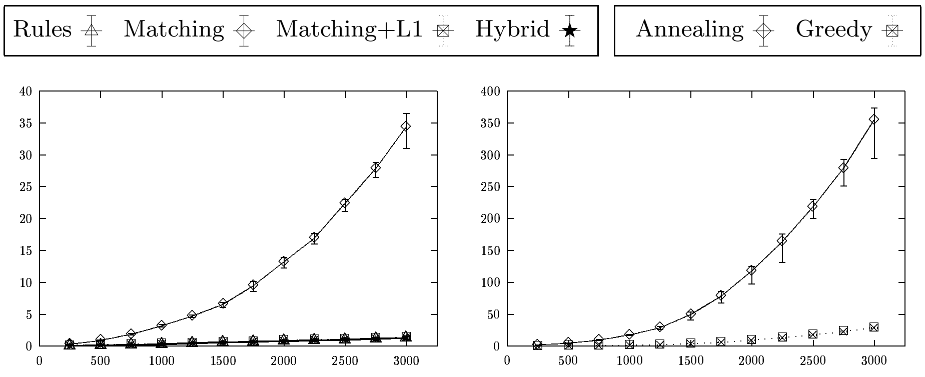


Figure 6: MunichDrillholes: point number versus running time

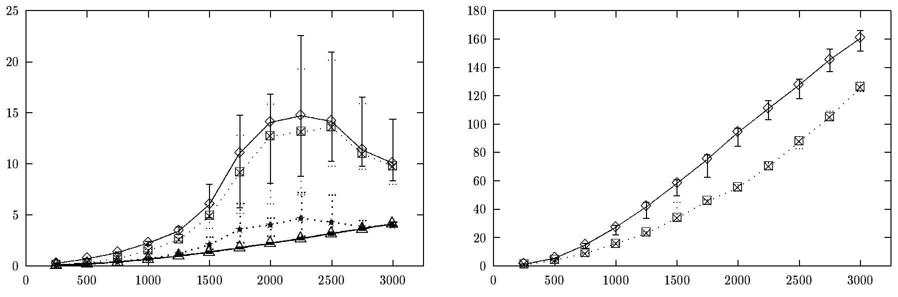


Figure 7: VariableDensity: point number versus running time

Given heaps and priority search trees, the greedy algorithm would run comparably fast to Rules. Our implementation of simulated annealing seems to be slower by a factor of 2–3 than that of Christiansen et al. [2]. This difference in running time may be due to the machines on which the times were measured.

On large examples, our algorithm was faster by a factor of 2–10 as compared to the matching heuristic, and by a factor of 30–100 with respect to simulated annealing. Applying rule L1 as a pre-processing step speeds up the matching algorithm, slightly for VariableDensity but considerably for MunichDrillholes.

Conclusion

We have presented a simple and fast heuristic for a very general version of the label-placement problem. Given a set of features and a set of label candidates for each feature, our algorithm finds a large subset of the features, and for each of these features a label from its set of candidates, such that no two labels overlap. Due to this generality, we could not expect to achieve any approximation guarantee such as algorithms focusing on special label shapes. Still, our technique works very well for point labeling, a classical problem of cartography.

The results are similar to those of simulated annealing, but obtained much faster. Our algorithm Rules outperformed the matching heuristic of Kakoulis and Tollis [9], not only in terms of time, but also very clearly in terms of quality. Our new hybrid algorithm yields nearly identical results as Rules. Hybrid shows that our simple set of rules also helps to improve the heuristic of Kakoulis and Tollis. However, compared with Rules, it is questionable whether the additional effort for applying BFS repeatedly and using an $O(k\sqrt{n})$ bipartite-matching subroutine really pays.

For fast Internet applications, such as on-line mapping, our experiments strongly suggest that Rules is the method of choice. For high-quality label placement, however, further experiments are needed in order to investigate how well our algorithm handles priorities, different feature and label shapes, and a greater number of label candidates per feature.

Acknowledgments. We thank Christian Knauer for technical support, Lars Knipping for implementing the example generators, Alexander Pikovsky for his interesting experiments with simulated annealing, and Rudi Krämer, Frank Schumacher, and Karsten Weihe for supplying us with real world data.

References

[1] Pankaj K. Agarwal, Marc van Kreveld, and Subhash Suri. Label placement by maximum independent set in rectangles. In *Proceedings of the 9th Canadian Conference on Computational Geometry (CCCG'97)*, pages 233–238, 1997.

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

[3] Srinivas Doddi, Madhav V. Marathe, Andy Mirzaian, Bernard M.E. Moret, and Binhai Zhu. Map labeling and its generalizations. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*, pages 148–157, New Orleans, LA, 4–7 January 1997.

[4] Michael Formann and Frank Wagner. A packing problem with applications to lettering of maps. In *Proc. 7th Annu. ACM Sympos. Comput. Geom. (SoCG'91)*, pages 281–288, 1991.

[5] Robert J. Fowler, Michael S. Paterson, and Steven L. Tanimoto. Optimal packing and covering in the plane are NP-complete. *Information Processing Letters*, 12(3):133–137, 1981.

[6] Herbert Freeman. An expert system for the automatic placement of names on a geographic map. *Information Sciences*, 45:367–378, 1988.

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

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

[9] Konstantinos G. Kakoulis and Ionnis G. Tollis. A unified approach to labeling graphical features. In *Proc. 14th Annu. ACM Sympos. Comput. Geom. (SoCG'98)*, pages 347–356, June 1998.

[10] Donald E. Knuth and Arvind Raghunathan. The problem of compatible representatives. *SIAM J. Discr. Math.*, 5(3):422–427, 1992.

[11] Stefan Näher and Kurt Mehlhorn. LEDA: A library of efficient data types and algorithms. In *Proc. Internat. Colloq. Automata Lang. Program.*, pages 1–5, 1990.

[12] Tycho Strijk and Marc van Kreveld. Practical extensions of point labeling in the slider model. In *Proc. 7th ACM Symposium on Advances in Geographic Information Systems*, pages 47–52, 5–6 November 1999.

[13] Marc van Kreveld, Tycho Strijk, and Alexander Wolff. Point labeling with sliding labels. *Computational Geometry: Theory and Applications*, 13:21–47, 1999.

[14] Frank Wagner. Approximate map labeling is in $\Omega(n \log n)$. *Information Processing Letters*, 52(3):161–165, 1994.

[15] Frank Wagner and Alexander Wolff. A practical map labeling algorithm. *Computational Geometry: Theory and Applications*, 7:387–404, 1997.

[16] Frank Wagner and Alexander Wolff. A combinatorial framework for map labeling. In Sue H. Whitesides, editor, *Proceedings of the Symposium on Graph Drawing (GD'98)*, volume 1547 of *Lecture Notes in Computer Science*, pages 316–331. Springer-Verlag, 13–15 August 1998.

[17] Alexander Wolff. *Label Placement in Theory and Practice*. PhD thesis, Fachbereich Mathematik und Informatik, Freie Universität Berlin, May 1999.

[18] Chyan Victor Wu and Barbara Pfeil Buttenfield. Reconsidering rules for point-feature name placement. *Cartographica*, 28(1):10–27, 1991.

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

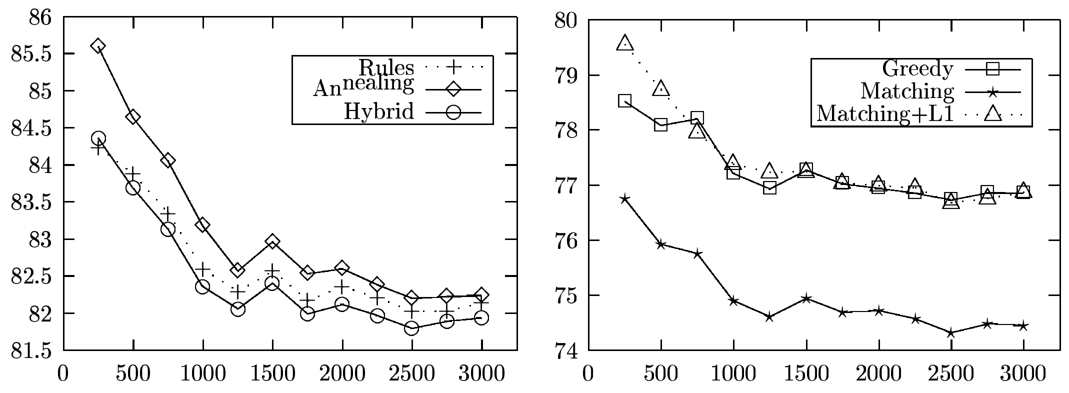


Figure 8: RandomRect

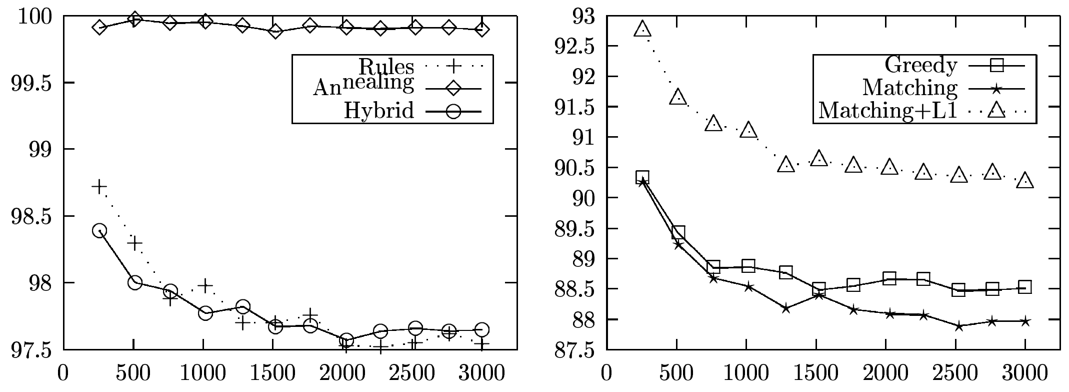


Figure 9: DenseMap

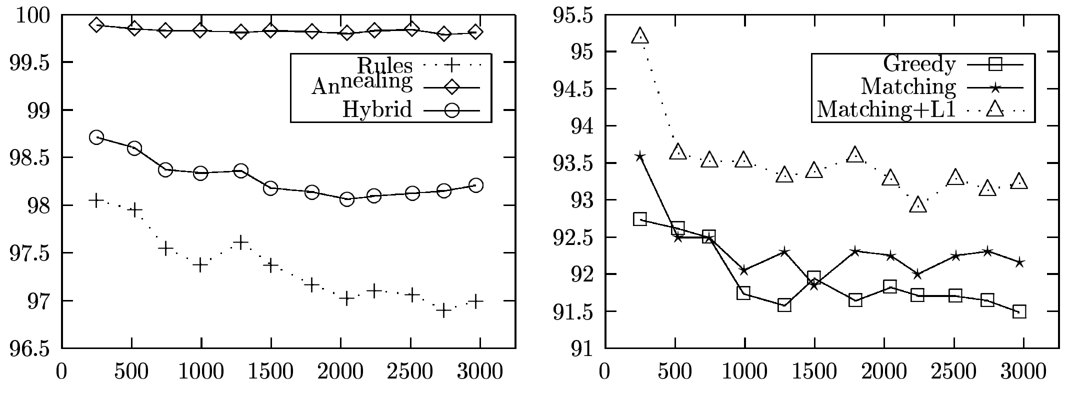


Figure 10: HardGrid

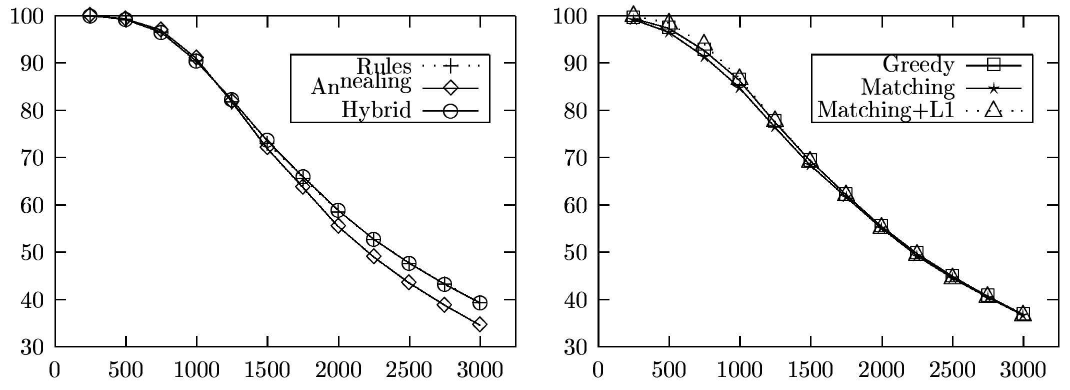
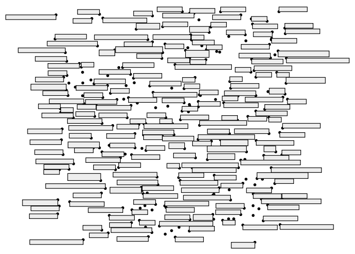
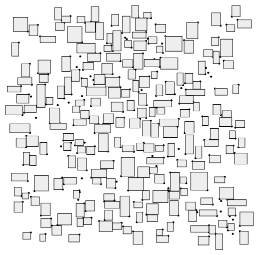
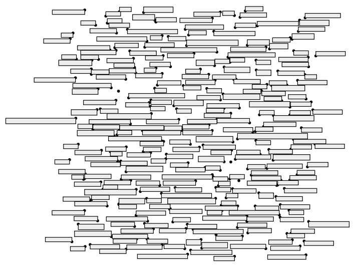


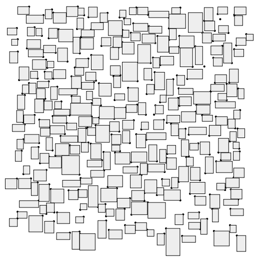
Figure 11: VariableDensity

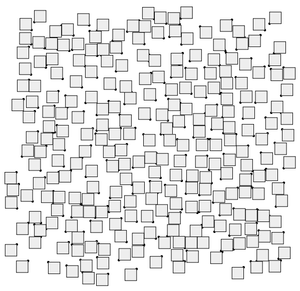
Appendix B: Instances of our Example Classes

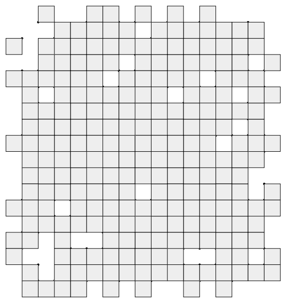

RandomMap: 250 (193) points

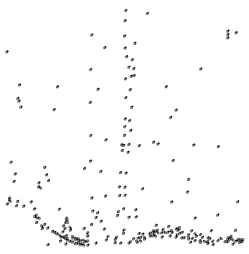

RandomRect: 250 (212) points

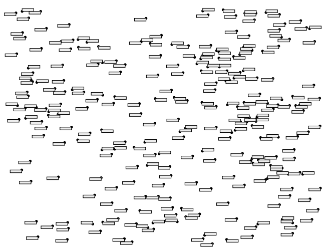

DenseMap: 253 (249) points


DenseRect: 261 (258) points


HardGrid: 253 (252) points


RegularGrid: 240 (240) points


MunichDrillholes: 250 (250) points


VariableDensity: 250 (250) points

Appendix C: Example Maps

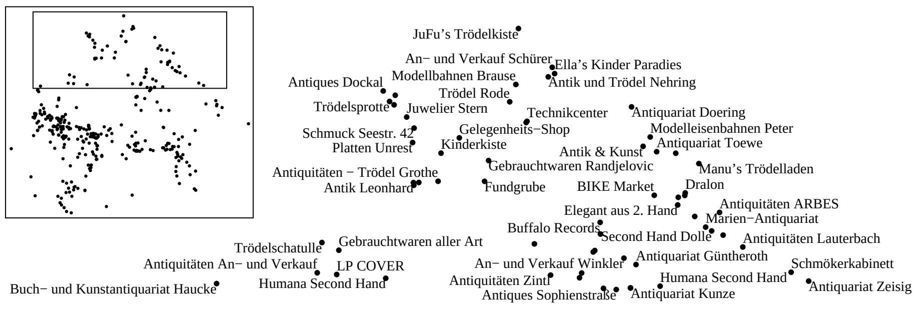


Figure 12: left: 357 tourist shops in Berlin, right: 45 of 63 labeled.

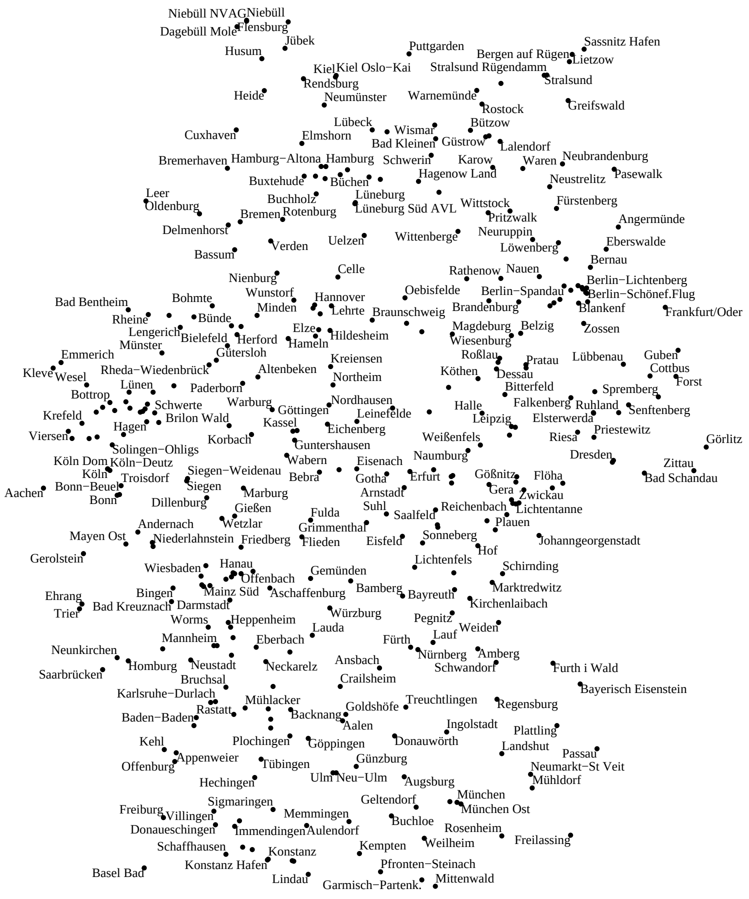


Figure 13: 373 German railway stations, 270 labeled.