Internal Report 2011–01 January 2011

Universiteit Leiden
Opleiding Informatica

Improving *Mapnik* With
Label Placement Algorithms

Bertram Bourdrez

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Improving *Mapnik* with Label Placement Algorithms

Bertram Bourdrez, Leiden University

January 27, 2011

Abstract

The *Mapnik* map renderer, used by the *OpenStreetMap* project, suffers from poor aesthetic qualities. One of the issues affecting the quality of the output is the placement of labels. Labels are too small, poorly positioned or missing altogether. As automatic label placement has been a subject of research in computer science, an obvious solution would be to apply one of the algorithms developed over the past two decades to the problem. This paper attempts to research, implement and evaluate such an algorithm that could be implemented within the flexibility and performance constraints of the *OpenStreetMap* project while providing accurate and visually pleasing output.

1 Introduction

With the rise of computer- and internet-based mapping applications, the desire has come — inspired by the *free software* model — to have access to a freely usable source of map data. The ability to embed commercially available map data into an application or website is seriously limited by a host of restrictive licensing conditions, and is often impossible due to technical and legal constraints. The success of Wikipedia and other user-contributed sources of data has led to the idea of user-contributed mapping data as obvious approach that could provide high-quality unencumbered data to non-commercial users and companies alike.

The *OpenStreetMap* project aims to create such a freely available world map, describing itself as "a project aimed squarely at creating and providing free geographic data such as street maps to anyone who wants them." The project provides an infrastructure for volunteers to provide and edit mapping data; many of these volunteers are ordinary people who explore and record their environment. Unlike commercially available sets of data which often only contain data relevant to a particular usage, *OpenStreetMap* aims to record any information which can be agreed on as relevant to users of its geographic data.

1.1 Problem description

For all its strengths, *OpenStreetMap* is let down by its map renderer, *Mapnik*, which generates maps which, while technically quite adequate, are not particularly visually pleasing. One of the issues with the *Mapnik*-generated maps is that they do not show enough labels for towns and regions, and the labels it does display are too small. Both of these faults can be traced back to a very poor labeling algorithm. In order to improve *Mapnik*, the labeling must be improved dramatically. This involves finding an algorithm to solve the Point Feature Label Placement problem. Thousands of modifications are made to the *OpenStreetMap* dataset every day, and new maps (at a wide range of zoom levels) have to be generated after every modification, making it imperative that the algorithm runs in as short a time as possible.

Therefore, our goal is to find an algorithm that adequately labels maps for the *OpenStreetMap* project and provides pleasing and legible results in a short runtime.

The theoretical background is discussed in Section 2, with a description of the problem some solutions in 3. The existing algorithm is discussed in Section 4. A possible solution is discussed in Section 5, with the three solving algorithms discussed in Sections 6, 7 and 8. The evaluation method is described in Section 9 and the conclusions are summarised in Section 10.

This paper is a bachelor thesis, under the supervision of Walter Kosters (Leiden University).

2 Label Placement

Computer-generated cartography was seriously discussed for the first time in the early 1960s, when guidelines [1] for mechanically-generated maps were drafted, despite the necessary equipment and software being unavailable available at the time. The real breakthrough came in the early 1980s, when the price of mechanical map generation was about to overtake manpower (cartographers spend most of their time on label placement, placing about 20 to 30 labels an hour [2]).

2.1 Types of labels

The cartographic conventions [1] set out in the 1960s specify three general kinds of labels to be placed in general-purpose maps:

1. **Point feature labels** are labels associated with a point feature such as mountain peaks, stations, libraries as well as — at low[1] zoom levels — towns, parks and lakes. These labels are best placed close to the object they denote.

2. **Area feature labels** are used to mark labels with clear boundaries, such as towns and lakes at higher zoom levels, and countries and seas at lower zoom levels. These labels are best placed inside the area in question.

3. **Line feature labels** are placed alongside line features such as rivers and roads. Whereas point and area features are generally placed horizontally for the sake of legibility, line feature labels follow the general shape of the line, and at high zoom levels are even positioned inside the outline of a road or river.

These three kinds of labels require different algorithms. Compared to point and area feature labels, line feature labels are obviously a completely different type of label, the placement of which has less to do with avoiding conflict with other labels and more with finding an ideal contour to follow that is both intuitive and easy to read. These labels are generally only rendered at the very highest zoom levels, at a zoom level where there are very few point feature labels. Rendering line feature labels is another interesting problem, but it lies outside the scope of this paper.

Point feature label placement focuses on trying to place the labels for each point feature in the most optimal place, without overlaying labels on each other or other features. Various strategies for placing area labels exist, although by their nature they tend to be easy to place. When used under the right conditions, area and point feature labels can be placed simultaneously with the same algorithm.

2.2 Desired properties

To prevent confusion, point feature labels are ideally placed to the immediate right or left of the point they label, with a slight preference to the right. The label should line up below or above the point, which many algorithms interpret to mean that the top or bottom of the label should line up with the point, which conveniently divides the area around the point into four quadrants in which a label can be placed (see Figure 1).

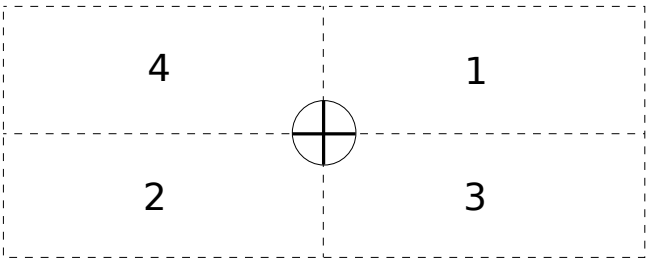

Figure 1: Possible positions of a label in one possible order of preference

[1] Note that *low* level of zoom means "not zoomed in much", i.e., zoomed out. This might seem contradictory to some.

In many cases this is too restrictive, however, and an additional four positions can be allowed as outlined in Figure 2. Note that for many algorithms, this increases the complexity of the problem by a factor of two or more.

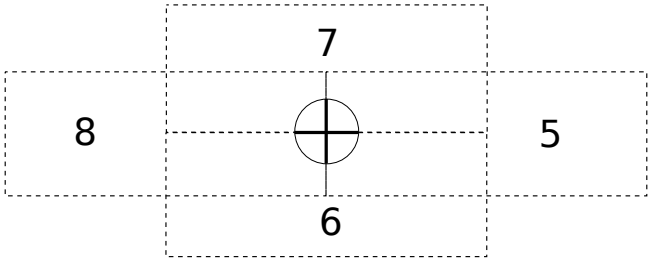

Figure 2: Another four possible label positions, to be combined with Figure 1

Other algorithms allow a label to be placed anywhere, as long as the point with which it is associated lies somewhere on the edge of the bounding rectangle.

3 PFLP algorithms

PFLP (*Point Feature Label Placement*) can be defined in a variety of ways, depending on the type of map and the demands placed on it. Most versions of the problem reduce it to a relatively narrowly constrained question, either to reduce the complexity or to make sure that the solution fits a specialist problem at hand.

At its most general, the problem is defined [3] as

> *A set of point features and a set of constraints (such as permissible amount of overlap) for placing labels. The goal of the problem is to label each feature so as to satisfy the constraints.*

The constraints used are the ones that make sense from a usability, aesthetics and complexity point of view. Commonly applied constraints include:

- Amount of overlap allowed, both between the labels themselves and between the labels and the rest of the map features.
- Positions of a label which are acceptable — either a discrete number based on what is known to be a satisfactory and unambiguously distinguishable placement, or any position that meets certain distance and relative position criteria.
- The angle at which a label may be placed. Most algorithms currently do not change the angle at which a label is rendered as it increases the complexity and is difficult to do in a way which conforms to cartographic standards.

In addition there is the *Point Selection* problem. If one accepts a solution where some labels do not appear, selecting which labels to show and which to hide is a separate problem [4] which must be solved at the same time. The importance of labels is often not given, or at least not with sufficient accuracy that would allow the algorithm to determine whether one or more labels can be dropped.

3.1 Satisfaction algorithms

Due to their clearly bounded complexity, satisfaction algorithms were among the first applied to the PFLP problem.

Lemma 3.1. PFLP *with no overlap, no selection and two acceptable placements per label can be reduced in polynomial time to* 2SAT.

Proof. For each point p, consider both possible label placement p_1 and p_2. Find all placements $p'_{i'}, p''_{i''}, p'''_{i'''}, \ldots$ for other labels that intersect with this placement. Construct logic statements of the form $\neg(p_i \wedge p'_{i'}) = \neg p_i \vee \neg p'_{i'}$ for each of those intersections. Finally, construct for each label p statements of the form $\neg(p_1 \wedge p_2) = \neg p_1 \vee \neg p_2$ to specify that at most one label per point is placed and statements of the form $p_1 \vee p_2$ to ensure that at least one label per point is placed. Each of these steps can be performed in $\mathcal{O}(n^2)$ time for n labels. The conjunction of all these statements can be computed using 2SAT. □

This means that PFLP problems with two possible placements are computable in real-world scenarios.

Lemma 3.2. PFLP *problems with no overlap, no selection and two acceptable placements per label can be computed in polynomial time and are* ***NL****-complete.*

Proof. Since the conjunction of disjunctions above is in *Conjunctive Normal Form*, 2SAT algorithms can be applied. These are known to run in polynomial time and nondeterministic logarithmic space [5]. □

Using a similar technique we can reduce the 3SAT problem, which is **NP**-complete, to the PFLP problem with more than two positions. Therefore, allowing more than two placements per label requires a more constrained version of the problem or a solution that is not guaranteed to be optimal.

3.2 Other deterministic algorithms

Greedy algorithms perform surprisingly well on the PFLP problem. They will make decisions based on what provides the greatest immediate improvement at any point during the algorithm — which generally means placing a label in an unoccupied position if possible. When no such position exists, such an algorithm backtracks to the last position where an alternate choice was available. These algorithms perform remarkably well on simple data sets, but suffer from variable run times and struggle with more complicated data sets [6].

Many algorithms restrict the size or shape of a label to either specific shapes (circles [3] and squares [5] provide good results) or to a single uniform rectangle size. These algorithms use geometric properties of the labels to exclude impossible placements and therefore reduce the search space considerably. Of course, the size constraint on the labels means that this solution can only apply to certain highly specific domains.

3.3 Non-deterministic algorithms

Selecting a placement for n labels gives n^q possible solutions when choosing between q possible label placements per label. For all but the lowest values of n and q the complexity of performing anything approaching an exhaustive search through the solution space becomes very time-consuming. Therefore we turn to non-deterministic algorithms like Simulated Annealing and Evolutionary Algorithms.

Annealing is a metallurgical process which is used to change properties such as hardness and strength. To this end the metal is heated until it is white hot, and then slowly cooled down. During this process, the heat imparts energy onto metal atoms, causing them to oscillate more wildly. During the cooling down process, atoms will find new — and hopefully more — positions to settle down. While creating a computer model of this cooling process, chemical engineers came up with a model [7] where during each step, atoms were given a random displacement. If the total amount of energy E in the system was decreased by this motion, the displacement was accepted and the atom was moved. However, if the energy went up, the movement would be accepted with a likelihood of $P = \exp(-\Delta E/k_B T)$, where ΔE is the increase in energy, k_B is Boltzmann's constant (a certain amount of energy per degree of temperature) and T is the temperature. As temperatures go down during the cooling process, the likelihood of displacements that increase energy will go down steadily.

Simulated Annealing is an algorithm which mirrors the process of metallurgical annealing: a process in which randomness is introduced in order to bring about small, iterative improvements. Intriguingly, it also closely mirrors the algorithm that simulates the annealing process by iteratively

accepting new states that are either better than the current state or — with ever decreasing likelihood — somewhat worse than the current one [8]. In this "simulation of simulated annealing" one can even use the same probability function $P = \exp(-\Delta E/k_B T)$ or a variant thereof.

Evolutionary Algorithms [9, 10] treat candidate solutions to a problem as individuals in a population and uses well-known evolutionary biology concepts such as crossover, mutation and recombination to create new solutions which are then added to the population. A selection mechanism picks the individuals which are to survive to the next iteration of the algorithm based on their suitability as a solution to the problem, as calculated by an *objective* or *fitness* function. There are various different kinds of Evolutionary Algorithms which differ, for example, in the way in which they represent the solution as a genome, and the operations they perform on them during every generation.

Evolutionary Algorithms and Simulated Annealing are both non-deterministic algorithms which use a random element to attempt to avoid getting stuck in a local optimum; Simulated Annealing will randomly accept a worsening iteration, and Evolutionary Algorithms use selection algorithms which keep a balance between locally optimal as well as clearly non-optimal individuals in the gene pool.

Both algorithms need an *objective* or *fitness* function which evaluates a solution and represents the degree to which it fits the constraints as a number.

4 Mapnik

Mapnik is one of the map rendering tools used by the *OpenStreetMap* project. The only popular alternative, *osmarender*, is considered to be a dead end in development as it is written in XSLT, which puts serious restrictions on performance and maintainability.

Mapnik is therefore the main way of representing *OpenStreetMap* data. It generates the tiles for the main *slippy map*[2] on the `http://www.openstreetmap.org/` front page and most of the other graphic representations generated for the project.

4.1 Design

Mapnik works by parsing an XML configuration file called a *style* file, which contains information about which features should be rendered. These features are organised in layers, which the renderer outputs from bottom to top — i.e., the first layer defined in the style file is rendered first, and subsequent layers are rendered over it, possibly obscuring elements previously rendered. Each layer specified in the *style* file has a list of geographical features which should be represented on that layer, and style properties such as colour and width for those features. Feature types include (using the names of the *Mapnik* classes):

- Point, such as towns and mountain peaks.
- Line, including roads and rivers.
- Polygon, used for the outlines of objects such as built-up areas and lakes.
- Text
  Unfortunately, both line features and point features are handled by almost exactly the same code path and data structures, distinguished only by a `label_placement` flag:
  - `point` features are always aligned horizontally, and need to be placed near their associated point.
  - `line` features should match as closely as possible with the line (road, waterway) they label.
- Shield, for marking motorways with the road number on a coloured background (see Figure 3).

[2] A *slippy map* is a map that can be dragged around, allowing the user to explore an arbitrarily large area, and change between zoom levels, without reloading the web page.

- Building.
- Marker, for icons that denote things like mail boxes and railway stations (see Figure 4).

Various other types are defined for future expansion.

Figure 3: Example of motorway shields (S112 and A10)

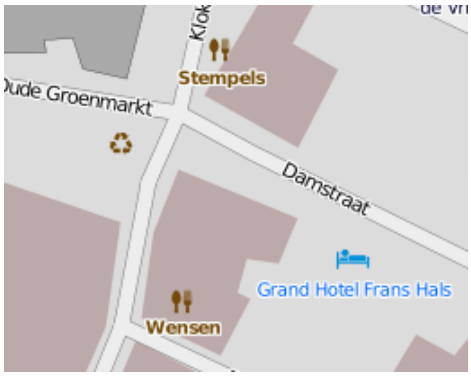

Figure 4: Example of markers (restaurants, recycling point, hotel)

4.2 Design deficiencies

Due to the layered design, *Mapnik* has some issues which make proper label placement more difficult. Information gathered in the process of generating one layer is not propagated to the next, thus requiring duplicate work or loss of information. Problems include:

- Text is rendered on various different layers, making it hard to track what is already on the map, and making it impossible to undo unfortunate placements in later layers.
- Information about geographical features that might conflict with a label placement is lost when a layer is concluded. Therefore, text is often rendered on top of important landmarks. Only motorway shields and bitmap markers are exempt from this problem, because they are rendered last.
- Text is supplied to the text rendering layer with only a set of coordinates as indication as to where it should be rendered. Whether or not it is acceptable to render *onto* those coordinates, or if the label should be *next* to the coordinates, is not expressed.
- Labels are supplied to the renderer in order of importance, but without a measure of exactly *how* important a label is. This makes it impossible to determine whether a poorly-fitting label should be rendered at all.
- The outlines used to calculate potential collisions between labels are the bounding boxes of the individual letters instead of the bounding boxes of the whole label. While this allows — in theory — for more elegant and precise fitting of labels, it does increase the complexity of the algorithm by an order of magnitude.

However, the most obvious problem with the text rendering in *Mapnik* is the lack of an actual label placement algorithm. When the text rendering layer is asked to render a label, it attempts to place it in the exact position specified. If this is not possible due to a label already being in that position, the label is ignored and processing moves on to the next label. This pseudo-algorithm explains why the default font size in *Mapnik* is so small: bigger fonts would cause too many features to disappear from the map. If the legibility of *Mapnik*-generated maps is to be improved, font sizes will have to be increased and therefore a higher quality label placement algorithm is required.

5 Algorithm design

Given the size and complex nature of the search space, it is clear that it is unlikely that the single best solution can be found in a reasonable time. Therefore we must define an *objective* function that will approximate the quality of a solution. Finding a solution which largely conforms to the following quality guidelines will suffice — indeed, manual label placement generally involves finding such an imperfect solution.

Because of the inconsistent use of terminology in literature on the subject, we begin by defining the following:

Definition 1. *A **point** is an (x, y) coordinate on a map which should be labeled.*

Definition 2. ***Labels** are the text which denote what a* point *represents, plus the rectangular bounding box around the text.*

Definition 3. *Each* point *has one or more **placements**, a finite number of possible positions where a* label *could be placed.*

5.1 Data sources

Label placement is one of the last stages of the map rendering process. By the time the label rendering layer is invoked, the map should already contain all the geographical features that are to appear on the finished version. Section 4.2 describes the layered approach to the rendering process, and our labeling algorithm is one of the last layers to be applied.

Consequently, input to the algorithm consists of the map as rendered up to that point, plus information of what the label placement algorithm has to do, in the form of a set of points. Each point is provided with the following information:

Field	**Type**
Font size	Integer
Colour	RGB value
x coordinate	Floating point
y coordinate	Floating point
Letter bounding boxes	List of rectangles

Table 1: The metadata associated with labels in *Mapnik*

More fields are provided, but those are either not relevant to the placement, or would only be relevant when doing more advanced placement (like rotation and wrapping).

Obtaining the relative importance of town labels from this *OpenStreetMap* data alone is difficult: *OpenStreetMap* only supplies four levels of detail for town labels, which make up the majority of the rendered information at lower zoom levels (see Table 2).

`hamlet`	$<$ 1,000
`village`	1,000–10,000
`town`	10,000–100,000
`city`	$>$ 100,000

Table 2: Populations in *OpenStreetMap*

To counter this problem, official population statistics from Dutch government sources [11] were used as an additional input. Even though it is possible to create a usable map from the *OpenStreetMap* data alone, the government numbers provide a more challenging problem as well as more pleasing output. Additionally, *OpenStreetMap* does provide a mechanism for recording a town's population, and therefore it is not inconceivable that user-supplied population data will make it into the *OpenStreetMap* database in the future.

Using these sources of information, a model is created in which each point is stored along with the metadata and information necessary for subsequent calculations. For each label, four or eight possible placements are created (see Figures 1 and 2). These placements are then checked for intersections with any of the other placements so that future collision detection calculations can be sped up by being performed on only the relevant neighbouring placements.

One piece of information of note is the *importance* which is stored per point. The importance is a value which indicates how important a point is. Ideally this value is very high for points which should never be hidden, and very low or even negative for unimportant small towns.

5.2 Desired properties

In order to rank different candidate solutions we need an objective measure of quality. Such an evaluation function will be based on cartographic convention and aesthetic value — or at least a reasonable approximation thereof. We will attempt to list the qualities that can be evaluated in a reasonable time by a computer algorithm.

5.2.1 Overlap

The most obvious and easiest to calculate criterion is overlap, specifically the number of overlaps between pairs of placements. By keeping a list of which placements overlap it is easy to check if both of those places are actually selected to be used, and if so, a collision is recorded. The overlap will be calculated based on the bounding rectangles of the labels — a map in which the bounding boxes overlap but the label text does not will still look very crowded and undesirable, and calculating collisions between rectangles is an order of magnitude faster than anything else.

5.2.2 Label omissions

In most circumstances, the objective is to place as many of the available points on the map as possible. Therefore the algorithm will be supplied with a surplus of points, with many of the labels not making it into the final map representation. The algorithm will be rated on the number and importance of the points that have been omitted. Leaving out a capital town is inexcusable, whereas leaving out insignificant objects is perfectly acceptable, since they are only supposed to be rendered if the space is not occupied by anything else.

5.2.3 Preferred position

Since Western languages are read from left to right and from top to bottom, our preferred (Western-language) map will have the labels to the right (or possibly below) the point they label. This means that not all possible placements are treated equally, as seen in Figures 1 and 2.

5.2.4 Distance between labels

In parts of the map with relatively low amounts of conflict between, various solutions might be possible. To find a preferred solution from the various possible ones, the distance between labels is taken into consideration. Where possible, placing labels *away* from each other is preferred to reduce visual clutter.

5.2.5 Overlap with other map features

Map design guidelines [1, 12] suggest that labels should not be placed on top of important map features, where they would hinder the user's view of potentially important information. It is one of the most important quality measures for cartographers, but one of the least considered ones in the field of automatic label placement due to the difficulty in quantifying the amount of information obscured.

5.3 Objective function

Armed with a model, we will now attempt to create algorithms that return an acceptable solution. How acceptable a solution is will have to be determined according to the previously mentioned quality characteristics. For all these values, it goes that lower is better.

Each point can be in two states, *shown* or *hidden*, and for each point, one particular placement is considered as "picked." All calculations are done as if all the *hidden* points will not be shown, and the label will be rendered in the "picked" placement.

For each point, the following is calculated:

1. **Overlap** (see Section 5.2.1) The objective function for a label is defined by calculating which other labels it collides with. For each of these collisions, the point with the *lowest* importance (see above) will have the difference in importance between the two points added to its objective value.

2. The objective function associated with each **hidden** (see Section 5.2.2) label is simply the importance of that label.

3. For the **preferred position** (see Section 5.2.3) of a label according to convention, simple constants are used, with a higher objective value for less desirable positions such as top left.

4. **Distance between labels** (see Section 5.2.4) is measured by simply checking which nearby (but non-overlapping) placements are in use. Labels which are closer together cause a higher objective value.

5. Working out the amount of **obscured information** (see Section 5.2.5) is quite a difficult problem, as the relevant geographical data needed to determine whether a placement overlaps with anything significant is discarded in earlier steps. Even if this information was available, it would still be quite computationally intensive to do anything with it.

   Therefore we will do something simpler and cheaper: we calculate the difference in the colour of the pixels obscured by the letters. For each such pixel we measure the Euclidean distance in the RGB colour space between it and its neighbours.[3]

   For each placement, we calculate the following:

Algorithm 1: Algorithm for calculating the information of a rectangle

```
Sum = 0 ;
foreach pixel p do
    if p obscured by text then
        foreach neighbour n do
            Sum = Sum + (p.red − n.red)² + (p.green − n.green)² + (p.blue − n.blue)² ;
        end
    end
end
```

   Note that this calculates the square of the distance, but as we are only looking for *relative* information values, we can forego the expensive square root. We then order the placements by this value and apply penalties when the placements that obscure more information are selected.

5.4 Implementation

The original approach to the problem was to modify the *Mapnik* renderer to hold off on placing the labels until they were all processed, and then perform a labeling pass. However, the quality of the *Mapnik* code leaves something to be desired, and adding functionality would involve a significant rewrite.

[3] There are more sophisticated ways of calculating colour difference, such as *CIEDE2000* [13], which involves more than 15 variables and dozens of floating point calculations including trigonometry and square roots.

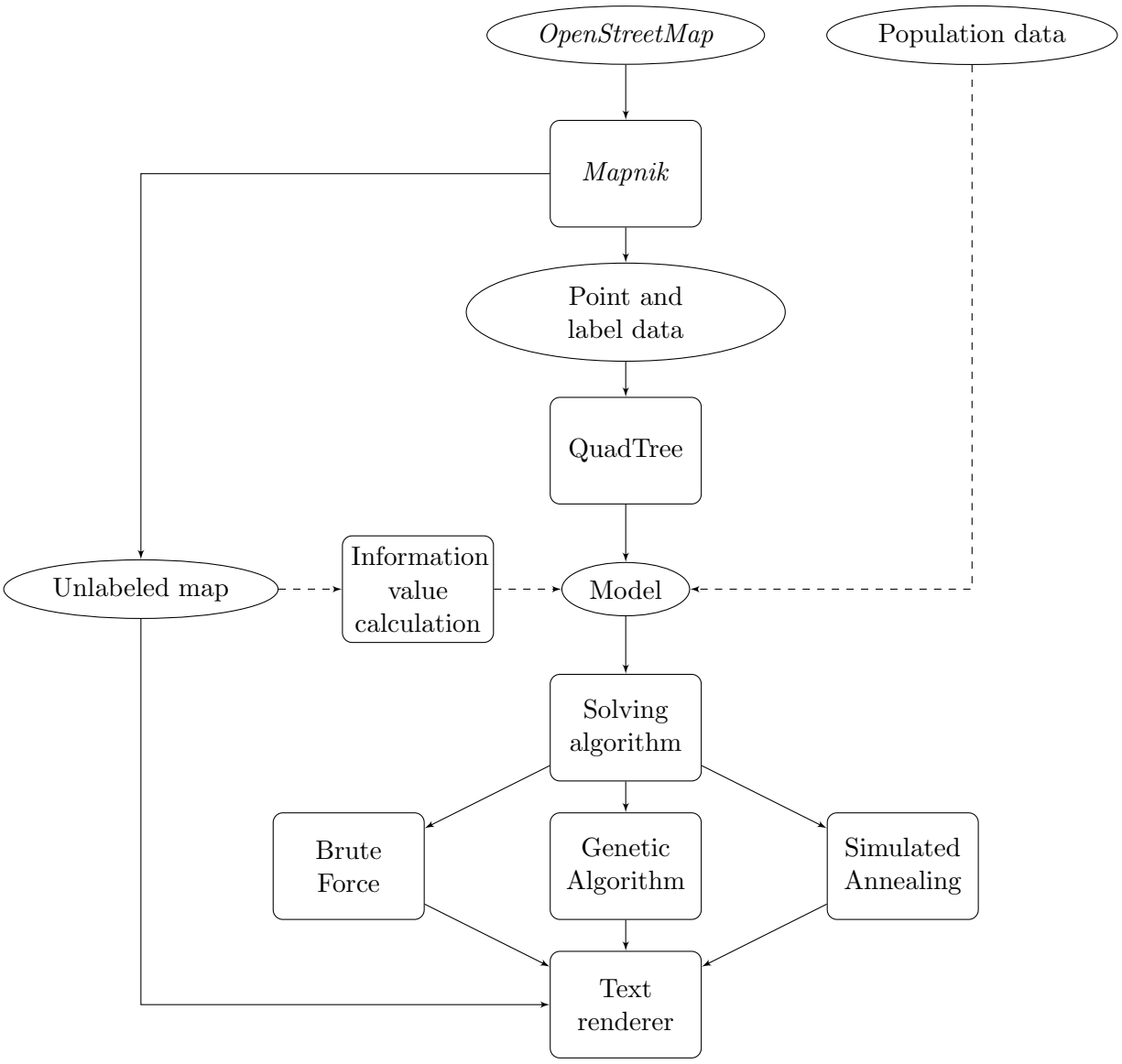

Figure 5: Diagram of the data flow in the rendering process. Rectangles are algorithms, ellipses are data. Dashed lines are optional.

Therefore it was decided to modify *Mapnik* to create a map without any labels on it, accompanied by a list of the labels and their metadata (see Table 5.1).

With this data we can create a model and solver that is not encumbered by any of *Mapnik*'s design flaws. We can choose a language that is more suitable to rapid prototyping and frequent design changes. Additionally, since the map is already rendered, we can repeatedly run the map labeler without having to go through all the other stages of the map creation process.

Figure 5 shows the steps involved in creating a labeled map.

5.5 Collision optimisation using Quad Trees

To determine whether the different positions for labels collide, one has to check whether each of four (see Figure 1) or eight (see Figure 2) placements collide with every possible position of every other label, meaning that for n labels one has to do $(8 \times n) \times (8 \times (n-1))$ rectangle intersections, each of which take four coordinate comparisons, for a total of $256n^2 - 256n$ comparisons.

This relatively unnecessary work slowed down the startup of the rendering process significantly, especially with moderately high values of n. To speed up the process of working out which placements conflict with each other, the placements are stored in a *Quad Tree* [14, 15], a data structure designed to efficiently find points or rectangles lying in a certain area. With a *Quad Tree*, all rectangles that

intersect with a given rectangle can be calculated in $\mathcal{O}(\log n)$, with a similar cost for inserting the placement into the *Quad Tree*. In all, the complexity was reduced by a factor of roughly $8n$.

6 The Genetic Algorithm

Since we now have a objective function, one basic requirement for creating a *Genetic Algorithm* is fulfilled. To create a working algorithm, we need a representation for candidate solutions, and three basic operators: selection, combination and mutation.

6.1 Representation

Genetic Algorithms generally use bit strings to represent candidate solutions, and we can store all the information for a single point with q possible placements in $1+\log_2 q$ bits; one bit for whether the point is hidden and the rest to represent the currently chosen placement. In the four or eight position case, this would result in three or four bits per point. However, to accommodate more sophisticated crossover and mutation operators, each point was represented by a bit indicating whether the point was hidden, and a number that represented which particular placement had been selected. This approach was faster (as converting from bit representation to selected placement was not necessary) and produced better results.

6.2 Selection operator

The preferred method for selection is often *proportional selection* where an individual gets chosen with a chance which is inversely proportional to its objective value,[4] but since the lowest possible objective value for some problems is often quite high (in particular in very crowded areas where a lot of labels will either collide or be hidden), the difference in objective value between individuals is often only a fraction of the total objective value of these individuals. Therefore we apply the simpler method of rank-based selection, where an individual is selected with a probability function that heavily biases the individuals with the lower objective value.

6.3 Crossover operator

Crossover operators vary wildly in effectiveness, so four completely different operators were tried:

One-Point and **Two-Point Crossover**.

Multi-Point Crossover, where each point is selected as a crossover position with a probability p_c. This means that *a priori* it is not clear how many crossover points there will be.

Uniform Crossover. Note that this is merely the special case of Multi-Point Crossover with $p_m = 0.5$.

A domain-specific crossover (an implementation of a **Bias Crossover** operator) is one that is more likely to select the breaks between clusters as a crossover point. That way, solutions for parts of the map that are internally optimal are more likely to be preserved.

6.3.1 Clustering

Using a modified [16] version of Kruskal's *Minimum Spanning Tree* algorithm, all points can be divided into clusters of related points. Kruskal's algorithm [17] works by merging trees until a single spanning tree is created. In this modified version, when merging would create a tree exceeding a certain size, the trees are not merged. This results in a forest of trees, all below a certain size. These trees then form the clusters we are looking for. The ordering of points in our representation is done by cluster, and the position of the breaks between clusters is stored.

[4] Remember, lower objective values are better.

The Bias Crossover operator discussed above will select breaks between clusters with a somewhat higher probability than other points, thus making it more likely that clusters will make it from one generation to the next unmodified. However, even if the crossover operator picks a different crossover point, at most one cluster per crossover will be altered. Any remaining clusters will be preserved unmodified from one of the parents.

6.4 Mutation operator

After some experimentation with the standard mutation operator, which flips each bit with a certain probability (say, $p_m \approx 0.01$), it was found that this does not provide enough mutation to get the algorithm out of the many local minima in the search space. If a single label is moved, all the labels around it must often be moved as well if any progress is to be made. In extreme cases four or five labels would need to be moved to improve a labeling, but each individual move would make the result worse.

A relatively simple domain-aware mutation operator does not just move a label with probability p_m, but when it does, it also moves all the labels that intersect with both the old and new placement with a (significantly higher) probability p_n. This causes every mutation to cause an "earthquake" in its immediate neighbourhood, thus increasing the chances that the radical change that would be necessary to improve the most "stuck" areas of the map would be affected.

A similar approach is to select a particular cluster of points, and then mutate all points within that cluster with probability p_n.

7 Simulated Annealing

The Simulated Annealing algorithm only operates on a single solution, improving it by changing one label at a time. This makes a single iteration significantly faster (most of the time in both algorithms is spent calculating the objective function, which only has to be run once per iteration).

The algorithm creates a starting position by selecting a random placement for each label, with all labels being visible. The temperature is initialised at 1.0 and k is set to a value between 0.1 and 1 (depending on how long the algorithm is allowed to run). The algorithm then proceeds to perform the steps of Algorithm 2 until the temperature drops below 0.003.

Aside from the speed improvement which results from only having one solution to work with, the evaluation function can be optimised as only label ℓ can have changed between invocations of the evaluation function. Therefore the evaluation function only has to consider ℓ and any labels ℓ might intersect with. For other labels we can use cached information, as the five properties we calculate are local and do not depend on any information other than properties about the label itself and any intersecting nearby labels.

The downside of the algorithm is that it has to run several thousands of iterations before the temperature has dropped sufficiently.

8 Other algorithms

Other algorithms were considered as potential solutions to the PFLP problem. Two notable ones will be discussed briefly.

8.1 Memetic Algorithms

A type of algorithm that has become quite popular is the Memetic or Hybrid Algorithm [18] which combines the random approach of Evolutionary Algorithms and Simulated Annealing with search algorithms that rely on domain knowledge.

Due to the speed constraints on our solution method it would be advisable to incorporate a more direct search method. This can make it harder to escape from a local optimum but it considerably

Algorithm 2: Simulated Annealing algorithm

```
counter ← 0
while temperature < 0.003 do
    O_old ← ObjFunction()
    ℓ ← a random label
    if rand() < 0.2 then
        if ℓ is hidden then ℓ is unhidden
        else if ℓ is unhidden then ℓ is hidden
    else
        label ℓ is moved to another randomly selected position
    end
    O_new ← ObjFunction()
    if O_new > O_old then
        if rand() < e^a, where a = ((O_old − O_new)/O_new) / (k·temperature) then
            do nothing
        else
            revert changes to label ℓ
        end
    end
    counter ← counter+1
    if counter % 5 = 0 then
        temperature ← temperature·0.7
    end
end
```

reduces the run time of the algorithm, and it guarantees that each solution conforms to a minimum quality level.

One solution which was incorporated into both the Evolutionary Algorithm and the Simulated Annealing methods was a 'fix-up' pass that was run after a certain number of iterations of the algorithm, that checked whether any points needed hiding or unhiding. Any point with a low importance value that collided with various other (more important) points was hidden, and any point with a high importance value that could be shown without colliding with anything important would be shown.

8.2 Brute force algorithm

Even though brute force algorithms tend to fare badly at this kind of problem, one is included in the comparison for three reasons. Firstly, it provides a decent baseline comparison for the other algorithms. Secondly, if it does turn out to provide a solution that is in any way useful, it will do so in an order of magnitude less time than the other algorithms, as it does not explore large parts of the search space. Lastly, it can be used as a hill-climbing algorithm during or after the run of the other algorithms.

The brute force algorithm is exceedingly naïve. Going over all points in a loop, it first tries to move each label to a more suitable position. In the second and third passes it tries to hide and unhide labels, respectively. Each operation is only performed if it improves the objective value. This allows the algorithm to be run repeatedly until it stops improving without the fear of creating an infinite loop: if the objective value improves each time the algorithm is run there is no way the algorithm can run forever as either the objective value will hit 0 or it will stop improving.

9 Evaluation

As stated in Section 1, we aim to find an algorithm which 'provides pleasing and legible results'. In order to find out whether which of the algorithms succeeded in that, we need a measure of quality for the labeling.

To provide a point of reference, the maps generated by the Evolutionary Algorithm (`evolve`), the Simulated Annealing algorithm (`anneal`) and the brute force algorithm (`brute`) are compared to the base OSM renderer (`original`), *Yahoo!*'s Static Map rendering service (`yahoo`) and the Google Maps API (`google`). Maps were generated at various zoom levels corresponding to the zoom levels 8 through 11 (inclusive) on Google's map API, which roughly corresponds to levels 10 through 7 (*Yahoo!* uses the reverse order of zoom levels to other online mapping systems [19]).

At first, these commercial mapping solutions were used as points of reference to fine-tune the evaluation function. Google Maps in particular is widely considered [20] to be a leader in digital map labeling for consumer applications, so taking their work as an example seems like a good — if ambitious — goal. The other major company involved in the consumer map website market is Microsoft with its Bing offering, but the Bing website does not offer a service which could be used in this comparison, and a recent Bing redesign has severely reduced map label legibility.

Examples of the three solutions (Figures 7, 9 and 11) are shown here for reference, with the commercial map of the same area shown in Figures 6 and 8 and the current state of *OpenStreetMap* in Figure 10.

Figure 6: Google's rendering

Figure 7: Simulated Annealing algorithm

9.1 Website

Side-by-side comparisons of the output of all three algorithms next to web references showed promising results. However, since we ourselves set the standards by which the objective function judges the results, the results were always going to conform to our own ideals. These ideals do not necessarily agree with the map legibility ideals of the general map-reading public. Clearly, to evaluate the result, another method would be required.

To this end, a website was created, which asked the user to rate a series of maps. On each page, the following was presented (see Figure 12):

- A pair of maps, one chosen from the set { `google`, `yahoo`, `original` } and one from the set { `evolve`, `anneal`, `brute` }.

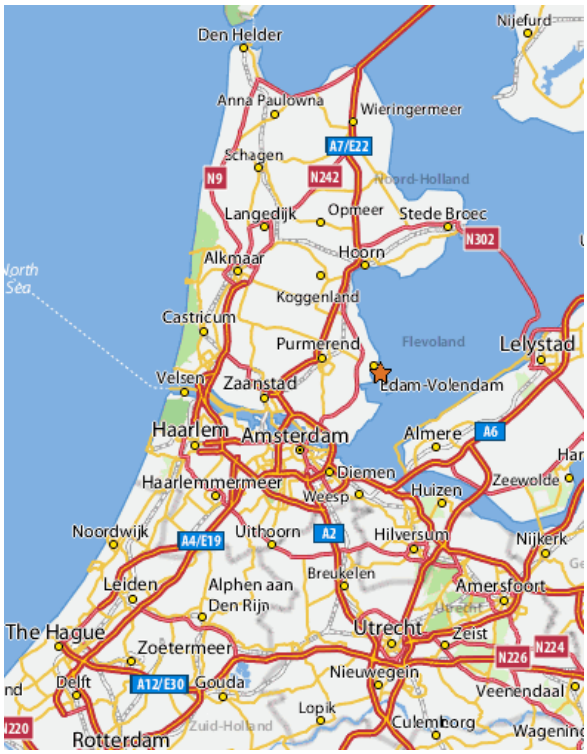

Figure 8: Yahoo! rendering

Figure 9: Genetic algorithm

Figure 10: Unmodified *Mapnik* (OSM)

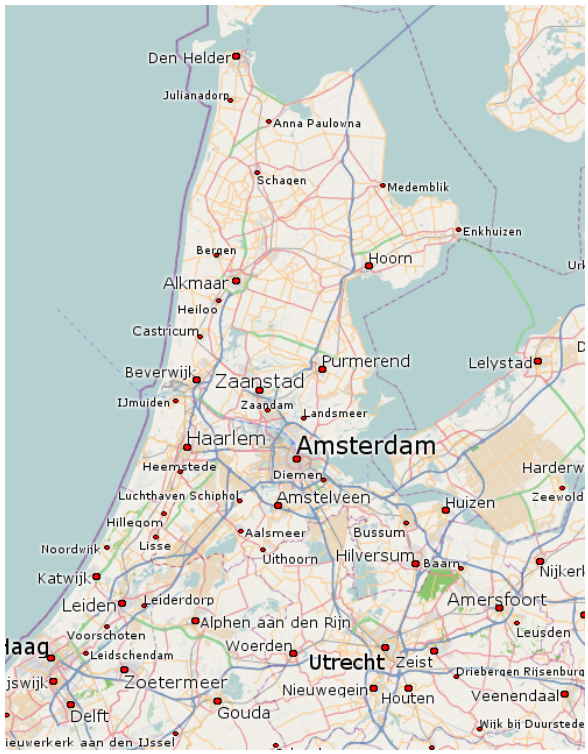

Figure 11: Brute force

- In some cases these maps would be in the original colour, in other cases they would be converted to black and white to make sure the subjects were not being distracted by the colours present in the map.
- The order in which the maps were presented was random. The reference map was placed on the left as often as it was on the right.

- Subjects were asked to rank maps on four criteria:
  - Amount of information (`information`)
  - Aesthetics (`pleasing`)
  - Possibility for confusion (`confusion`)
  - Legibility (`legible`)

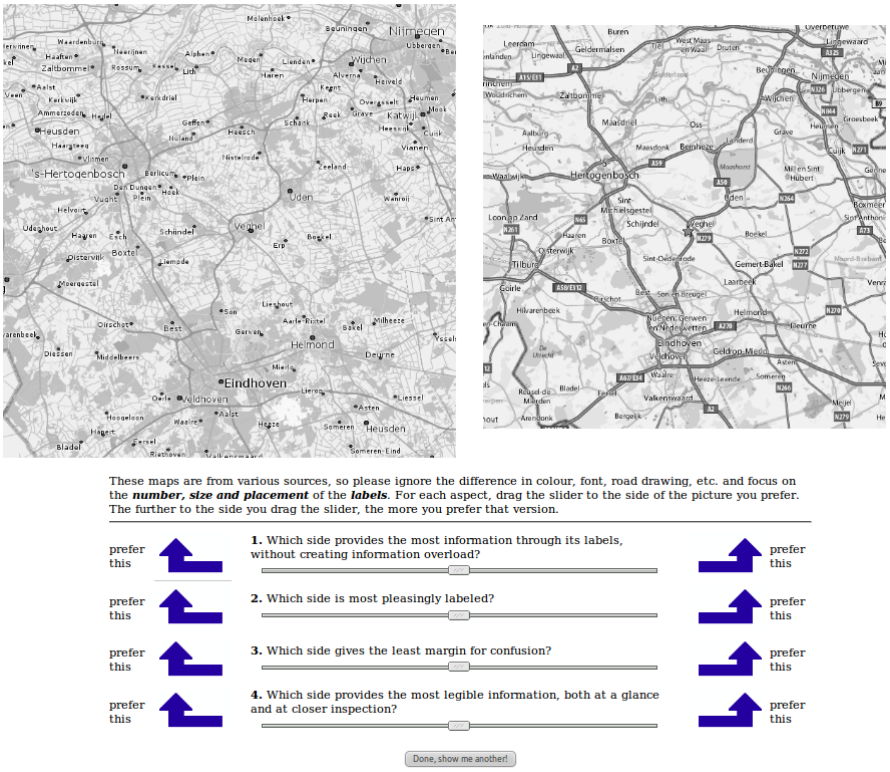

Figure 12: Evaluation website

Because it is very hard to rate maps on an absolute scale without a frame of reference, the site produces numbers which represent how much a map is preferred over another. These numbers rank from −50 (fully prefer the original) to +50 (fully prefer our version), with 0 expressing ambivalence.

The website was brought to the attention of several dozen subjects — friends, relatives, co-workers, fellow students — who were asked to rate as many maps as possible. A total of 42 random maps were rendered in all rendering engines in both colour and monochrome, although no subject rated all 42. Of the people contacted to participate in this survey, twenty-five responded, providing over 400 map comparisons.

9.2 Results

Figure 13 shows the results of the survey, showing the performance of each algorithm relative to each of the three reference labelings. Each bar represents the average of a certain set of ratings. These averages, again between −50 and +50, are separated out by both our algorithm and the reference map, so it's possible to analyse the performance of each of our algorithms against each of the reference ones.

The results are quite remarkable. As expected our maps fare worst against Google, given Google's experience [20] with mapping. Also unsurprising is the performance against *Yahoo!* and the original OSM rendering, since both have various obvious flaws that our solutions do not have, like a relatively low number of labels, a small and hard to read font and fairly naive placement. Several things do surprise, however. The `brute` solution (the exceedingly naïve brute force algorithm) fared extremely well — especially in avoiding confusion, for which it outranks every other algorithm.

It is also somewhat surprising that our algorithms outscore even Google in the 'legible' and 'confusion' metrics, although that is probably caused by focusing on just the labeling, where Google must juggle various user demands which we — without loss of generality — do not consider in this paper.

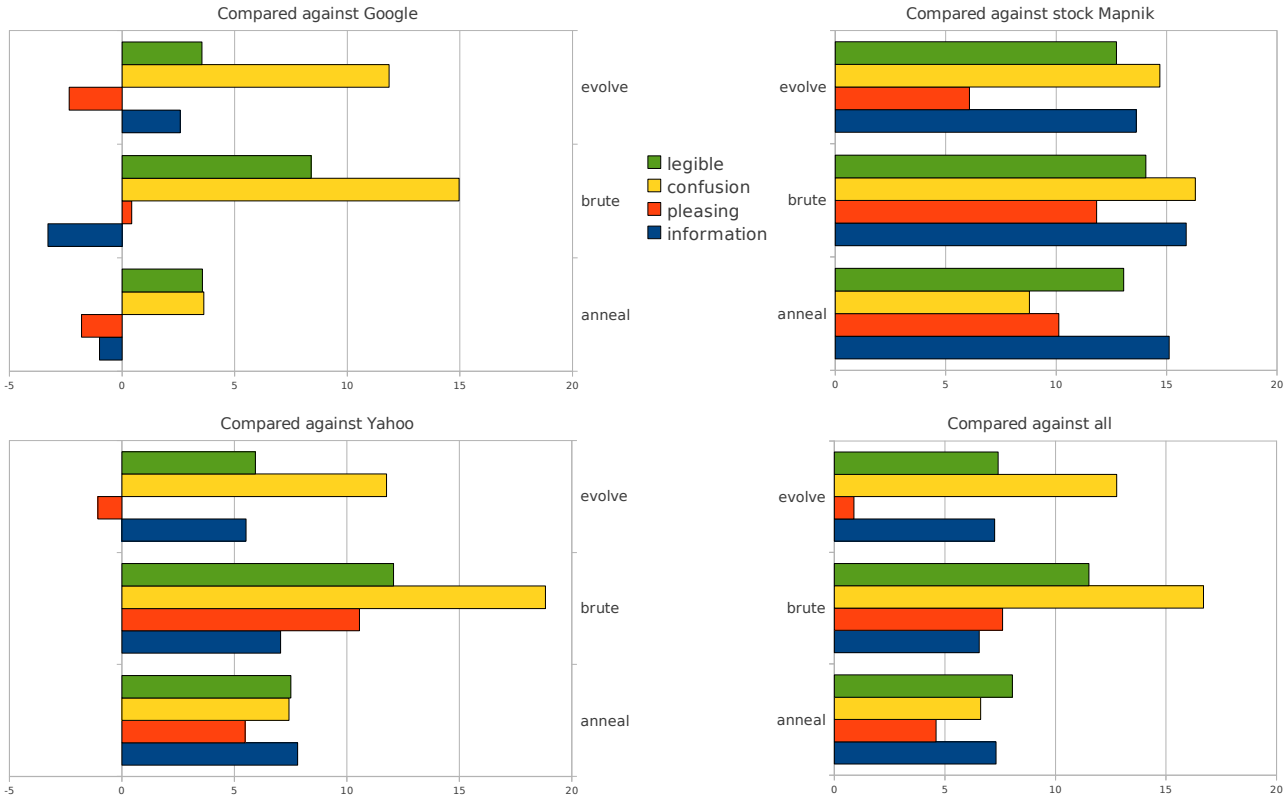

Figure 13: Rating website results. Each bar represents the average of all the ratings in that particular head-to-head matchup.

10 Conclusions

The evaluation in Figure 13 shows that the brute force algorithm performs at least as well as if not better than the nondeterministic algorithms. While the solutions it produces are less sophisticated, some of its very deficiencies could explain why it is rated so highly. For example, it discards labels it cannot place far more quickly than the other algorithms. This might be seen as data loss, but it also makes the map clearer. Additionally, since it does not try solving complex arrangements of labels, and it will not hide major cities, there is an increased likelihood that small towns near big cities will not be shown. There is evidence to suggest [20] that not showing any labels around a major urban centre will make it stand out more, thus conforming to the user's perception of the city.

Given the enormous performance advantages (the brute force algorithm runs several orders of magnitude faster than the other two algorithms) it is clear that the brute force algorithm is a suitable way of rendering labels for an organisation like *OpenStreetMap*; it is fast, easily implemented and gives a high degree of user satisfaction. Even if it had performed slightly worse than the other algorithms, it would probably still have been preferable due to the shorter (and more predictable) runtime.

10.1 Further research

Further avenues of investigation can be identified which run in two very different directions. On the one hand, the label rendering in *Mapnik* can be improved in other fields than the PFLP solver. For example, the line feature label rendering needs work too, and it is probably advisable to create a separate algorithm which selects which labels to render.

On the other hand there are further improvements to be made to the PFLP solving algorithm. As indicated in Figure 13, all of the algorithms still lag behind Google in creating a pleasing and informative map. We tested only *if* our solution corresponded with what people expected of a map, and not the ways in which it *didn't*. By identifying what people feel needs to be improved, the algorithm can create maps that are even more readable. Some work [21] has been done in this area already.

References

[1] E. Imhof. Die Anordnung der Namen in der Karte. *Internationales Jahrbuch für Kartographie*, 2:93–129, 1962.

[2] A. C. Cook and C. B. Jones. A Prolog rule-based system for cartographic name placement. *Comput. Graph. Forum*, 9(2):109–126, 1990.

[3] S. Doddi, M. V. Marathe, A. Mirzaian, B. M. E. Moret, and B. Zhu. Map labeling and its generalizations. In *SODA '97: Proceedings of the Eighth Annual ACM-SIAM symposium on Discrete algorithms*, pages 148–157, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[4] L. R. Ebinger and A. M. Goulette. A constructive genetic approach to point-feature cartographic label placement. In T. Ibaraki, K. Nonobe, and M. Yagiura, editors, *Metaheuristics: Progress as Real Problem Solvers*, pages 205–214. Springer, NY, 2005.

[5] M. Formann and F. Wagner. A packing problem with applications to lettering of maps. In *SCG '91: Proceedings of the seventh Aannual Symposium on Computational geometry*, pages 281–288, New York, NY, USA, 1991. ACM.

[6] J. Christensen, J. Marks, and S. Shieber. An empirical study of algorithms for point-feature label placement. *ACM Trans. Graph.*, 14(3):203–232, 1995.

[7] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[8] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[9] S. van Dijk. Genetic algorithms for map labeling, 2001. PhD thesis, Department of Computer Science, Utrecht University.

[10] F. Hoffmeister and T. Bäck. Genetic algorithms and evolution strategies: Similarities and differences. In *Parallel Problem Solving from Nature*, volume 496 of *Lecture Notes in Computer Science*, pages 455–469. Springer Berlin / Heidelberg, 1991.

[11] Centraal Bureau voor de Statistiek. Bevolking – Cijfers, retrieved May 2010. `http://www.cbs.nl/nl-NL/menu/themas/bevolking/cijfers/default.htm`.

[12] G. N. Peterson. *GIS Cartography: A Guide to Effective Map Design.* CRC Press, Boca Raton, 2009.

[13] G. Sharma, W. Wu, and E. N. Dalal. The CIEDE2000 color-difference formula: Implementation notes, supplementary test data, and mathematical observations. *Color Research and Application*, 30, 2005.

[14] R. A. Finkel and J. L. Bentley. Quad Trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.

[15] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second edition, 2000.

[16] A. Vathy-Fogarassy, B. Feil, and J. Abonyi. Minimal spanning tree based fuzzy clustering. *World Academy of Science, Engineering and Technology*, 8, 2005.

[17] R. L. Graham and P. Hell. On the history of the minimum spanning tree problem. *IEEE Ann. Hist. Comput.*, 7(1):43–57, 1985.

[18] P. Moscato and C. Cotta. Memetic algorithms. In *Optimization Techniques in Engineering*, pages 53–85. Springer-Verlag, 2004.

[19] J. deVilla. Universal zoom levels for Google Maps, Live Search Maps and Yahoo! Maps, retrieved November 2010. `http://www.globalnerdy.com/2008/11/09/universal-zoom-levels-for-google-maps-live-search-maps-and-yahoo-maps/`.

[20] J. O'Beirne. Google Maps & label readability – why do Google Maps's city labels seem much more readable than those of its competitors?, retrieved November 2010. `http://www.41latitude.com/post/2072504768/google-maps-label-readability`.

[21] A. Kling. Label placement survey, retrieved December 2010. `http://www.youthencounter.eu/label-placement/survey-english.php`.