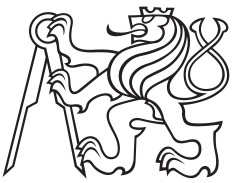
Master's Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Algorithms for Automatic Label Placement

Tomáš Chamra

Open Informatics
Artificial Intelligence

May 2017
Supervisor: Ing. Petr Pošík, Ph.D.

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science

DIPLOMA THESIS AGREEMENT

Student: Tomáš Chamra

Study programme: Open Informatics
Specialisation: Artificial Intelligence

Title of Diploma Thesis: Algorithms for Automatic Label Placement

Guidelines:

1) Familiarize yourself with the Automated label placement problem
2) Investigate different approaches to solve this problem
3) Design and implement at least two different non-trivial algorithms which solve this problem
4) Define a metric for scoring label placement algorithms and use it to compare algorithms implemented in the previous task

Bibliography/Sources:

[1] IMHOF, Eduard. Positioning names on maps. The American Cartographer, 1975, 2.2: 128-144.
[2] KOBR, Aleš. Automatické rozmísťování popisků na mapě. 2013.
[3] WOLFF, Alexander. The Map-Labeling Bibliography.

Diploma Thesis Supervisor: Ing. Petr Pošík Ph.D.

Valid until the end of the winter semester of academic year 2017/2018

L.S.

Head of Department

Dean

Prague, September 8, 2016

Acknowledgement / Declaration

First of all, I would like to thank my supervisor Ing. Petr Pošík, Ph.D. for a huge amount of patience and being very nice and helpful during the preparation of this thesis and during our frequent meetings. Without him, this work probably would not be finished ever.

Very special thanks goes to my family, which supported me and believed in me during my whole studies, and especially during my last year at the university, which was quite challenging for all of us.

Huge credits for helping me during preparation of this thesis goes to all my friends and colleagues, especially those at the university and at FREQUENTIS Czech Republic.

Computational resources were provided by the CESNET LM2015042 and the CERIT Scientific Cloud LM2015085, provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures".

I hereby declare that I worked out the presented thesis independently and I quoted all used sources of information in accord with Methodical instructions about ethical principles for writing an academic thesis.

Prague, May 25th, 2017

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 25. května 2017

.....

Abstrakt / Abstract

Práce popisuje problém automatického umísťování popisků do mapy. Jednotlivé bodové, čárové a plošné objekty v mapě je třeba označit odpovídajícími textovými či obrázkovými popisky. Tyto popisky je nutné rozmístit tak, aby se vzájemně nepřekrývaly a zároveň byly jasně přiřaditelné k odpovídajícím objektům. O problému je známo, že je NP-těžký a nalezení optimálního rozmístění všech popisků je výpočetně velmi náročné i pro nejjednodušší mapy.

Pozornost je věnována umísťování popisků označujících bodové a čárové objekty, včetně prvního kroku obnášejícího přípravu možných pozic pro umístění těchto popisků, při dodržení běžných kartografických pravidel pro rozmísťování popisků. Následně jsou na problém aplikovány tři různé druhy algoritmů – greedy („hladové“) algoritmy v kombinaci s lokálním prohledáváním, matematická optimalizace (v podobě 0-1 celočíselného programování) a genetické algoritmy.

Popsané algoritmy jsou v softwarové části práce implementovány a na závěr porovnány na několika různých datových sadách, vycházejících z reálných geografických podkladů a z náhodně vygenerovaných map. Závěrečné srovnání se zaměřuje na kvalitu výsledného rozmístění (dle metrik definovaných v práci), času potřebnému k nalezení řešení a také na determinističnost daných algoritmů.

Klíčová slova: umísťování popisků do mapy, hladové algoritmy, genetické algoritmy, matematická optimalizace

Překlad titulu: Algoritmy pro automatické umísťování popisků

Thesis describes the problem of automatic map label placement. Various point, line or area features in maps must be marked with matching text or graphic labels. These labels have to be placed so they do not overlap with each other and they are clearly associable with corresponding map features. The problem is known to be NP-hard and finding optimal positions of all map labels is highly computationally expensive, even for the simplest maps.

Focus is given to the placement of labels describing point and line map features, including the initial phase of enumerating possible label positions, respecting the basic cartographic rules common for those labels. Afterwards, three different algorithm types are applied to the problem itself – greedy algorithms (in combination with local search optimization), mathematical optimization (0-1 integer programming) and genetic algorithms.

Ultimately, the described algorithms are implemented in the software part of the work and compared on various data sets, based on both real world geographical data and randomly generated maps. The final comparison focuses especially on the quality of the result (scored by the metrics defined in the thesis), time needed to find the solution and determinism of the given algorithms.

Keywords: map label placement, greedy algorithms, genetic algorithms, mathematical optimization

Contents

Tables / Figures

Chapter 1
Introduction

Cartographers all around the world have to face many difficulties and barriers during preparation and publication of maps. Apart from mapping the terrain data itself, they also have to mark and label all important features using various symbols and labels. For each of these symbols or labels, cartographers have to decide about the best position to place them, in order to maintain readability and usability of the whole map.

Labels can have various forms – they might be represented by texts, graphical symbols, or even holes drilled in a metal, in some very special cases. Also, labels can have miscellaneous shapes, colors or sizes. They represent different map objects, which can be usually categorized as points, lines or areas.

According to various reports [1], cartographers can spend even more than a half of the map preparation time just by placing labels on the map – and that is a lot of time! Because of this, any kind of help that would simplify this process is obviously highly appreciated. It is quite surprising to find out how many people have been working on this problem (even long time ago, when computers were much less powerful than they're today) and how many companies struggle with this problem even now, in the 21st century.

There are two basic types of maps [2], which quite differ in the process of their preparation and the form how they are presented to users:

- **Static maps** are mostly intended for viewing and printing. During preparation, all desired features (e.g. cities, rivers or protected areas) and parameters (map size, scale, etc.) of these maps are already known since the beginning, so they can be hand-designed by the cartographers or generated by a computer and then either printed or distributed as images or PDF documents.
- **Dynamic maps** on the other hand allow more interactivity, like enabling/disabling data layers or changing the map scale or size. Changing these parameters could dramatically affect which objects (either map features or labels) should be visible and how they should be rendered (e.g. all cities should be marked in large maps with higher scales, while smaller maps with small scales should only display the largest cities). This category of maps is represented by various online map services and nowadays becomes more and more popular.

This thesis focuses on the first case, i.e. static maps, as they can be completely precalculated and rendered during the preparation time. Creating dynamic maps and rendering them in real time is a quite different task, requiring use of different other technologies and algorithms, which are usually commercially developed and they are not subject of this work.

Unfortunately, even the most simplified variation of the label placement task (labeling only point features and choosing just from four available positions for each label) has been proven to be NP-hard [3]. Knowing this, computer scientists have to look for some better heuristics than just relying on the brute force approach, which has unacceptable complexity for any reasonable number of labels.

Originally, this thesis was inspired by problems solved during preparation of aeronautical charts, which are maintained and regularly published by civil aviation agencies all around the world. However, the thesis itself is targeted on general label placement problem, instead of focusing purely on a single class of maps. This work is also highly influenced by the Master's thesis of Aleš Kobr [4], who has applied simulated annealing to label placement on aeronautical charts.

1.1 Thesis objectives

The goal of this work is to investigate and compare different available approaches to the solution of the label placement problem, and to find out which of the analyzed algorithms fits the best to this problem. For the comparison, there are three algorithms chosen for the deeper analysis:

- **Greedy algorithms** can often provide good results in a very short time, but they have a high chance of getting stuck in a local minima.
- **Mathematical optimization** methods can provide optimal solutions after performing exhaustive and long-running calculations.
- **Genetic algorithms** on the other hand have a chance to find quite good results in not so long time, but they are not deterministic and finding optimal solution is not guaranteed at all.

In order to compare the algorithms, it is necessary to define a metric determining which placement is nice and which is not. This metric should be used by the algorithms as an optimization criterion.

1.2 Thesis structure

First part of the thesis introduces and provides basic description of the map label placement problem and the basic rules which should be followed during the map preparation and label placement. It also contains a brief summary of existing works and approaches.

The following part describes metrics used to score different label positions and to choose the best ones, as well as metrics used to evaluate the whole map configuration, considering all labels together.

Third part deeply describes all three algorithm classes mentioned in this work – greedy algorithms, mathematical optimization and genetic algorithms.

Ultimately, the last part contains description of data sets used to compare the described algorithms and shows some sample maps labeled by these algorithms. It also contains detailed description of software implemented as a part of this work (including all previously described algorithms) and the conclusion of the thesis.

In the appendix of this thesis, there is a brief description of the computational geometry problems faced to properly calculate label positions and all necessary metrics. Appendix also contains contents of the attached CD disc and the list of abbreviations used in the thesis.

Chapter 2
State of the art

Over the past fifty years, authors from all over the world have published hundreds of articles about label placement, and especially its automated variant. German scientist Alexander Wolff (who is also author of many publications on this topic) has gathered The Map-Labeling Bibliography [5], containing most of the articles related to this topic. However, the list was not updated since 2009 and it is obviously not complete.

First article describing the problem was written in the 1960s by Swiss cartographer Eduard Imhof [6–7], who has thoroughly described placement rules of different labels for points, lines and areas. Imhof's work is considered as the base for all following works on label placement, as he briefly described which label positions are good and which are not.

Another important work was published ten years later by Israeli cartographer Pinhas Yoeli [8], who described his computer program for label placement and (probably more importantly) defined recommended positions for placing labels around points and described their priorities – which positions are better than the others.

2.1 Search spaces

One of the most important differences between various works done on this topic is a choice of a search space. There are two basic approaches to address this problem – either "discretizing" the search space by enumerating all possible label positions for each of the map features, or searching in continuous space [9].

Both of these options have their advantages, but also require different techniques to solve the problem – and this is not only question of the optimizer itself, but there is also a need for specialized algorithms used in analytical and computational geometry.

The first mentioned "discretization" approach is much more common and it is usually realized by enumerating possible positions according to some pattern (e.g. basic four or eight positions around a point object on a map, as proposed in already mentioned Yoeli's work [8]) and then choosing the best combination of these positions. These solutions are obviously restricted by the selected generation or enumeration pattern, but it is possible to avoid this issue by randomizing some positions over the map.

On the other hand, it is possible to consider labels all over the map (not just restricted by some patterns). This approach can offer many solutions that the previously mentioned one would not even consider, but this approach may bring the problem complexity even higher. Authors following this way sometimes use specific ways of simplifying the problem, and also use various discretization techniques, e.g. working with pixels that are going to be rendered on a computer screen [10].

2.2 Optimization goals

Within these hundreds of articles, researchers were facing many different variants of the label placement problem. Many of them focused just on labels related to point features on the map, while others labeled lines and areas as well.

Even more important decision is whether one wishes to find only "perfect" solution having zero collisions with other labels and/or map features (this definition is well suitable for methods of constraint satisfaction programming). Other possibilities are to remove labels that would cause conflicts in the map (and minimize number of the labels being removed), or allow collisions but penalize them using a fitness function.

Interesting subproblem might be the label size maximization [11], which focuses on finding maximal size of labels, either directly or by searching for some label size scaling coefficient.

2.3 Mathematical programming

As described in works of Robert Cromley [12] and Steven Zoraster [13–14], the label placement problem could be also handled by methods of mathematical programming. Cromley formulated the problem as a generic linear program, while Zoraster applied 0–1 integer linear programming. Both solutions were implemented in Fortran and Zoraster's solution was (or maybe still is?) used by many American companies active in the petroleum industry.

Zoraster used Lagrangian relaxation and subgradient optimization methods to solve the optimization problem, but probably the most interesting part is how he worked with constraints. In the first iteration, only the constraints related to initial solution were considered. Then, over time, additional constraints related to partial solutions were added. Thanks to this relaxation, the solver was presented with much less constraints then it would be with all constraints given in the beginning.

2.4 Stochastic methods

Due to overwhelming size of the search space, it might be a very good idea to consider applying some of the stochastic methods, like simulated annealing or evolutionary algorithms.

Jon Christensen et al. [9] described the use of simulated annealing algorithm for point feature label placement and according to their benchmarks, the algorithm outperformed other solutions like Zoraster's 0–1 ILP. Simulated annealing was also used by Aleš Kobr [4] for preparation of aeronautical charts. Kobr's work is also interesting as it uses Octree [15] for storing and manipulating large amounts of geometric data, making collision detection much easier and faster.

In 2001, Steven van Dijk used genetic algorithm in his thesis [16]. He implemented the algorithm for labeling point and line features, compared different crossover functions and proposed various improvements that may lead to better GA solutions, particularly inspired by heuristics proposed by Verner et al. [17] and Raidl [18].

Another implementation of genetic algorithm was presented by Karolína Burešová [10], whose work is also outstanding for solving the problem in a continuous space, i.e. not just generating pattern-defined positions and choosing from them. Apart from just changing the label position, she considered different label variants (e.g. with different shape or size) as well.

Multicriterial approach was investigated by Bradstreet et al. [19], optimizing three different criteria (maximal font-size, minimum number of conflicts and maximal clarity) and resulting multiple solutions. Final stage of the method involved manual intervention, which was required to choose the finest solution.

2.5 Other approaches

Among hundreds of other publications, one of the proposed solutions is reformulation of the label placement problem as relatively easily solvable 2-SAT problem [20–21]. However, this approach works only for very restricted variations of the problem.

Another way to approach the problem is application of various greedy and local search heuristics, or their combinations like GRASP (Greedy Randomized Adaptive Search Procedure). Cravo et al. [22] applied GRASP to point feature label placement problem in 2008 with quite good results, considering the short running time of the algorithm.

2.6 Commercial solutions

It is really nice to study and design algorithms for academic use, but sometimes the usability requirements in academic sphere are quite different from the ones in the commercial sphere. As already mentioned, some of the listed algorithms (e.g. Zoraster's algorithm) are used commercially, but many of them are only usable under very specific circumstances.

Unfortunately, users are not always willing to wait for hours (or even days) until the best algorithm potentially finds the best solution, so the developers often have to choose sub-optimal algorithms providing good solutions in minutes or even fractions of seconds, depending on the product type.

Another important aspect which has to be taken into account is a determinism of the solution. Stochastic algorithms bring a chance to find a really good solution, but this is generally not guaranteed and the performance in a single run might be poor. Also, users might be quite uncomfortable with computer programs producing various outputs for the same input data.

There are two major areas which could make use of label placement – GIS software suites used for creating maps, and various web applications displaying and using these maps. Both categories are represented by many different computer programs, but due to obvious reasons the algorithms used internally in these software suites are usually not very well documented.

GIS systems

Highest commercial potential for use of label placement algorithms is probably in the area of geographical information systems (GIS), that are used for various manipulations with geographical and spatial data, including preparation of maps. GIS systems often support both static and dynamic maps, according to classification described in the introduction.

Examples of these systems are **GeoMedia**[1] or **ArcGIS**[2], which are both widely used and among other features they both contain integrated modules for label placement. Apart from integrated algorithms, most GIS systems support third party plugins so they can be extended by even more complex and configurable solutions – for example, some maps might require use of special label shapes, which may not be supported by integrated label placement tools.

[1] `http://www.hexagongeospatial.com/products/power-portfolio/geomedia`
[2] `https://www.arcgis.com/features/index.html`

Online maps

Second area where label placement could be heavily utilized is represented by web applications like **Google Maps**[1] or **Mapy.cz**[2], which provide online access to maps to the general public. These maps are usually dynamic, i.e. the content displayed in them changes when user zooms the map in or out.

Obviously, when the visible content changes, the labels have to change as well in order to be usable and easily readable. For example, when user observes some city in detail, he sees all streets with their names and possibly even with house numbers. While the user zooms out, the streets disappear and he sees the city name instead. Then, when he zooms out even more, he may still see the city name, but in smaller letters (if the city is large), or it might disappear completely and it can be just replaced by the name of the region or the country.

Technically, these labels could be placed (and even determined whether they should be visible or not) either in real time, or they can be precalculated for different map scales – but due to performance reasons, in most web services it is probably the latter option, as the user wants to browse the map in real time, without waiting for some label placement process to be finished.

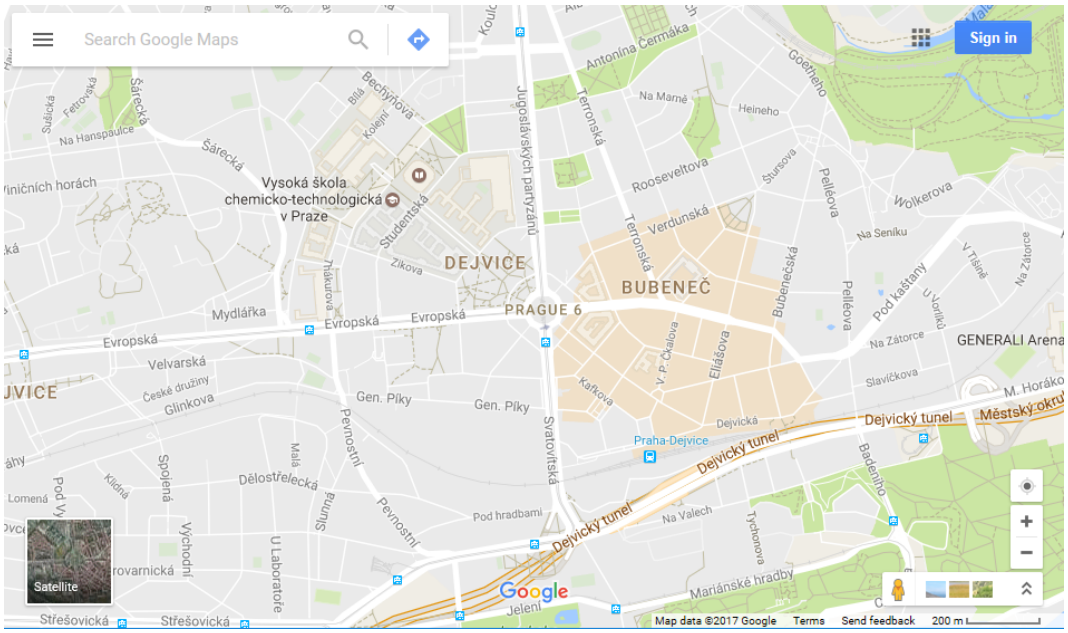


Figure 2.1. Example of maps provided by Google, containing many labels with different levels of importance

Also, many of these map applications use the same data sources in the background, like the data sets publicly provided by **Open Street Map**[3] (which are often based on cadastral maps published by the local governments), or other free or proprietary map providers. Therefore, most differences between online map platforms are not in the map content itself, but in their presentation, readability (this is the part related to this thesis) and additional features like journey planning or navigation.

[1] `https://www.google.cz/maps`
[2] `https://mapy.cz/zakladni`
[3] `https://www.openstreetmap.org`

Chapter 3
Basics of label placement

This section little bit more formally introduces the label placement problem and related definitions interesting for the purpose of this thesis. It summarizes basic rules for positioning labels – both general ones, applicable to all labels, and specific ones for different kind of labels.

3.1 Features and labels

First of all, the very basic and essential part of the problem is a **map**. Map is a drawing containing various map **features**, which can be generally divided into three basic classes:

- **Point features** represent small objects or points in the map. They may describe various points of interest, and with decreasing map scale they might describe even large objects (e.g. city is pretty huge area feature on a large scale map, but just a little point feature on a small scale map). In the computer, point features are usually stored as a point coordinates.
- **Line features** represent objects that can be stored and described by a line or curve. Examples of a line features are streets, rivers and similar objects. In the computer, line features are stored as lines, polylines, curves, arcs and similar structures.
- **Area features** covers all other features that cannot be represented by neither points nor lines, because they cover large area of the map (and therefore the point representation would not fully describe their size and shape) and they do not have linear shape. Example of these area features could be countries or regions, larger cities, lakes, forests and many other map features. In the computer, area features are stored as polygons (or sets of polygons).

Along with map features, there are also **labels** describing those features. Labels could have a form of text label or a small picture, and they are intended to describe the meaning of the features they are attached to. Content and visual appearance of the labels highly depends on the type and purpose of the map (road maps, tourist maps, aeronautical charts, etc.) as the labels can have various meanings in different maps.

Text labels have many parameters – the most important is definitely their content, but they also can use different fonts combined together with different styles (bold, italic, etc.) and sizes. And obviously, the important parameter for this work is the label position. Label, with all these parameters, define the meaning of related map features, their type, importance and determine how the reader will perceive and understand the whole map.

Image labels (pictograms) are usually based on images from a predefined set (which is also domain specific, as aeronautical charts will likely use different symbols than maps intended for tourists), and can represent various objects in a real world. For those symbols, there are two basic properties which have to be set – size and position.

In this thesis, the type, content and meaning of the label is not really important. It is enough to know the size and shape of the label (which is rectangular in most cases),

and the feature the label belongs to. This said, the algorithms presented further in this thesis expect to receive a list of labels, with their sizes and shapes, prepared in advance. The task of these algorithms is to find the best position for all labels – nothing less, nothing more.

3.2 Problem definition

The problem is simple – given a set of map features and a set of labels belonging to these features (with known sizes and shapes of the labels), find the best possible position for each label.

More specifically, each label has its bounding box, represented by a convex polygon (or a non-convex polygon, which can be converted to a convex one using one of the algorithms allowing to find a convex hull, as described in the Appendix A). The task is to find a position for each label, determined by a set of coordinates (the exact form is dependent on the coordinate system used) and angle. In most cases the labels are placed horizontally, but they do not necessarily have to be (non-horizontal labels are typical for example for description of linear features).

3.3 Solving the problem

As already mentioned in the previous section, there are two basic approaches to solve this problem – labels can be either moved in a continuous space, allowing many possible positions, or the solution space can be discretized by enumerating a set of possible candidate positions for each label and consequently assigning the best available candidate position for each label.

In this thesis, the latter mentioned approach is utilized. However, this basically divides the whole problem into two subproblems:

- **Candidate generation** is an essential preprocessing phase, which handles generation of candidate position set for each label. Positions are generated according to the rules described in the following sections, and they should respect various configurable parameters (e.g. the requirement might be to place a label along a line, preferably near to the left end of that line). This process is exactly the same for all of the placement algorithms used in the next step, and does not have to be repeated unless some placement rules have been changed or reconfigured).
- **Label placement** phase is the main topic of this thesis. Described algorithms take a set of candidates available for each label, together with other information about the map and some metric for measuring the quality of the solution, and they propose the best positions of all labels.

3.4 Labeling rules

The basic rules for placing labels were first described long time ago by Eduard Imhof [6–7] and even though they are quite old, they're still applicable and represent a good starting point for rule definitions.

First, Imhof described basic rules for all kinds of map features and those will be followed in this thesis as well (however, it is quite different task to just describe them on a paper, compared to incorporating them into a computer program). Consequently, he

described rules for various cartographic phenomena (dealing with rivers or mountains, various shapes of labels, etc.), but these are not so important for the purpose of this work – which is meant to be quite general, and not deep focusing into specific cartographic topics.

Legibility

As the most important property, labels in the map should be legible. That means that they should be easily readable, discriminable and visible. Legibility depends on the properties of used font (typeface, style, size, color, etc.), as well as on the visual arrangement of other labels and map features [7].

Clear graphic associations

It is extremely important to be able to quickly distinguish associations between map features and labels. This property is also tightly connected with previously mentioned legibility, since the visual look and contents of labels are also determined by the type and importance of the specific map features.

Sometimes, when the association between the label and the map feature it belongs to is not so clear, it might be a good idea to use a leader line. This line can simply visualize which label belongs to which feature, but on the other hand it can also make the whole map even more messy, because adding leader lines increases the number of elements visible in the map.

Avoiding overlaps

Third important factor during map labeling is avoidance (or minimization) of overlaps. These can happen both between two labels, as well as between a label and some map feature. Overlapping of map elements dramatically decreases the readability of the map, as the reader might easily get confused (overlapping may break previously mentioned clear graphic associations), as well as he may not even be able to read the label contents.

In the algorithms implemented as a part of this thesis, candidate position generation is focused on following all other rules, without considering any overlaps. Later, in the label placement phase, the placement itself is mostly based on minimization of overlaps.

3.4.1 Point feature labels

As already mentioned before, algorithms in this thesis use “discrete” space for solving the problem. In order to discretize the solution space, all labels must have a set of generated candidate positions, defining where they might be placed. There are many ways to generate these positions – they can be calculated according to various rules, or they can even by randomly generated (with a rare chance to find some excessively good position).

In the software part of this thesis, all candidate positions for labels belonging to either point, line are area map features will be generated according the rules described in the following sections. All of them are deterministic, so they can be recalculated at any time with the same result (unless some configuration has been changed).

Positions for point feature labels will be calculated according to some of the Yoeli’s [8] rules. Basic four positions are demonstrated in Figure 3.1. All of these positions are easily computable, based on label size and point position in the map. Point can be also represented by a small polygon, which defines a buffer around the point – this is quite important, because otherwise the labels would touch the point directly and make the point barely visible.

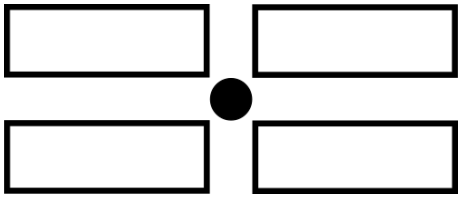


Figure 3.1. Basic four positions for point labels

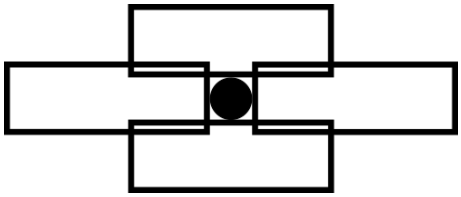


Figure 3.2. Additional four positions for point labels

However, having only four positions would be quite restricting. In Figure 3.2 there are another four positions, which are also very trivial to determine:

Usually the positions from the first set are considered first (with the top-right position being considered as the best one), while the positions from the second set are considered as the second option. In each of these two sets, some positions are better than others – but these preferences will be mentioned later, in the description of metrics.

Generally, point feature labels should be preferably placed on the right side of the feature (better than on the left side), and labels placed on the top are preferred to the ones on the bottom [7]. Labels should be as near to the point as possible, but they can be also placed further from the owning feature and connected using a leader line.

3.4.2 Line feature labels

Placing labels describing line features is a bit tricky, as those linear features could be made of various curves or similar graphic elements. Quite often those labels have a shape that copies the shape of the owning element (e.g. label for a river is often placed along the river, following the shape of the river). For the purpose of this thesis, all labels are expected to be straight – but this is not a restriction for the placement algorithms, as they work with bounding boxes and hence they do not need any more information about formatting of the label content.

In order to maintain readability, label should not cross the feature it belongs to. It should be preferably along the feature, not too far from it but also not touching it. Ultimately, the label should be placed as horizontally as possible.

Label generator used by the algorithms in this thesis offers multiple approaches to generate the labels for linear features. Labels could be placed either along the line, perpendicular to the line (or a specific segment of a line feature), or they can be placed horizontally without respecting the angle of the line, as shown in Figure 3.3.

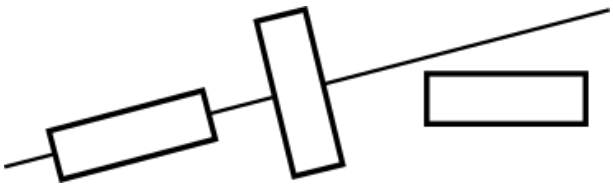


Figure 3.3. Rotations of line feature labels

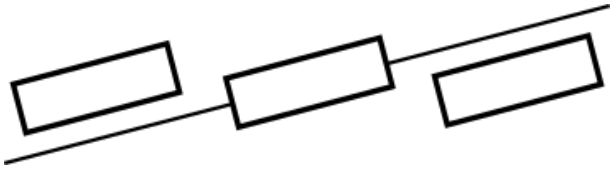


Figure 3.4. Vertical positions of line feature labels

In a similar manner, the labels can be placed either above, belong or directly on the line, as shown in Figure 3.4.

Ultimately, last proposed preference is the position on the feature – whether the label should be placed on the left end, in the center or on the right end, as shown in Figure 3.5. Unlike the remaining rules, which restrict possible positions for the candidate position generator, this setting is just a preference for the placement algorithm – not a constraint.

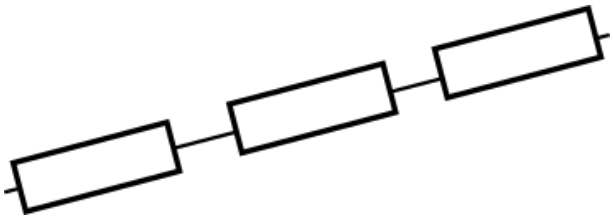


Figure 3.5. Horizontal positions of line feature labels

3.4.3 Area feature labels

Probably the most complicated map features are areas. They are usually bordered by some linear object, and there can be many other objects inside the area. Moreover, the areas could even overlap and that makes finding possible label positions quite difficult task.

Similar to line feature labels, the text inside the labels could be deformed in many ways, in order to fit into the map. Labels for area features can lie inside or outside of the area (but they shall not cross the border, as well as they shall not cross borders of other areas). Especially in case of overlapping areas, it is necessary to ensure the clear graphic associations between the area and the label – once more, leader lines might be helpful for this.

In this thesis, the problem of area feature labeling is not described any further as it is quite complicated and not directly the topic of this work. In the software part, all area feature labels are always placed exactly in the center of the area.

Chapter 4
Metrics

Algorithms described later in this thesis are designed to assign each label the best available position. But that task brings one important question: Which position is the best?

The question is not easy to answer. Cartographer (a human) usually decides based on his senses and subjective perception – what he believes that looks the best. For the computer, this task is complicated as "being the best" is quite difficult to define – in the world of computers, everything has to be described by numbers, and it could be a pretty difficult task to "numerize" a human's subjective point of view.

In fact, this simple question can be even divided into two separate questions, both of them being important for the solution of the problem:

1. Which label position is the absolutely the best?
2. Which label position is the best for the whole placement task?

In the first part of this chapter, labels are analyzed and scored individually, in order to decide which position is the best for a given label, without considering any other labels in the problem. This part is based on the rules described in the previous chapter.

In the second part, scoring of the problem is analyzed globally, considering positions of other labels as well. This part is applicable in the label placement algorithms, as this is the metric they're trying to minimize.

4.1 Scoring individual labels

Finding the best position for a specific label is basically equal to a minimization of penalties. There are three basic categories that can be penalized:

1. **Label positions**, because some of them are nicer than the others, e.g. if the top-right corner of a point is considered to be the best, all other (non-optimal) solutions should be penalized
2. **Conflicts with map features** should be penalized because of overlaps between map features and labels tend to decrease readability of the map
3. **Conflicts with labels** should be penalized as well, because overlaps between two labels also decrease the readability

Final score of the label position is equal to the weighted sum of these three penalties. In the software part of this thesis, default weights are 10% for label position penalty, 40% for conflicts with map features and 50% for conflicts with other labels. However, these weights are not fixed and can be changed to better fit a specific map.

4.1.1 Label position penalties

In the previous chapter, there was a list of rules used for labeling point, line and area features. It was also mentioned, that positions generated according to those rules have different priorities, i.e. some of them are better and nicer than the others.

Obviously, the computer must somehow consider and respect those priorities as well. One of the ways to approach this issue is to consider the priority as a part of the metric. In the implementation developed as a part of this thesis, the label position penalties are hardcoded for each candidate position generation rule. The penalties are between zero and one, and in the final metric they are multiplied by the respective weight.

Point feature labels

For point feature labels let's recall the basic eight positions presented in the previous chapter, as they are pictured in Figure 4.1.

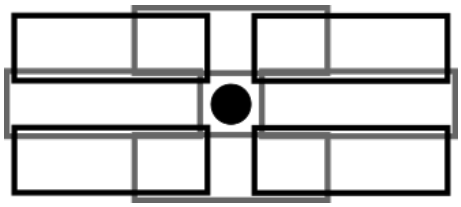


Figure 4.1. Basic eight positions for point labels

Generally, the best position for point feature label is considered to be the top-right corner and hence its penalty is zero. Other available positions from the basic set are top-left corner with penalty of 1/8, bottom-right corner with penalty of 2/8 and bottom-left corner with penalty of 3/8.

Another four positions are on the right side of the label with penalty of 4/8, on the left side with penalty of 5/8, on the top with penalty of 6/8 and ultimately on the bottom with penalty of 7/8.

It is obvious that fractions of seven could have been used instead of fractions of eight, but this decision would not make any significant difference and fractions of eight are easier to represent – there is no higher purpose in this decision.

Line feature labels

Candidate positions of the line feature labels can have three preferred positions – on the left end of the line, in the center or on the right end of the line, and calculation of the label position penalty differs based on this preference:

- **Left end** – labels on the left end have zero label position penalty, while those on the right end have label position penalty of 1. Penalties of all positions between left and right end are calculated linearly, hence the position in the center of the line would have a penalty of 1/2.
- **Center** – labels in the center have zero position penalty, and labels on the left or right ends have a position penalty of 1. Penalties for all points between center and any end of the line are calculated linearly.
- **Right end** – labels preferring right end are calculated in the similar manner as labels which should be placed on the left end. However, labels on the right end of the line have zero position penalty, while those on the left end have a penalty of 1.

Area feature labels

Since area feature labels are not handled in this thesis, they are always placed in the center of their owning map feature and hence the penalty is always zero (as there are no other candidates).

4.1.2 Map feature conflicts

Conflicts (overlaps) between labels and map features can have various impact on the label readability – same overlap with a specific feature can do just a little harm by covering part of a huge label, but it can also cover nearly whole area of a small (but important) label.

To handle these differences, label penalty for conflicts with map features is calculated as a ratio of label area covered by map features and the area of the whole label. This means, that an uncovered label should have zero feature conflict penalty, while fully covered label should have a penalty of one.

This penalty can possibly even exceed one, if the label area is covered by multiple objects. This happens because upper limit for penalties is not implemented, in order to make the problem easily tractable even for mathematical optimization solvers, where each additional constraint can have a huge performance impact.

4.1.3 Other labels conflicts

Label conflict penalty works exactly the same way as the map feature conflict penalty – the penalty is equal to the ratio of a label area covered by overlaps with other labels, and the whole area of the label. However, same conflict can inflict different penalties for different labels, as each label can have a different size and hence the damage caused by the conflict can vary.

This penalty is calculated separately, because it can have different weight as the impact on the overall map quality can be different.

4.2 Scoring whole solutions

It is nice to be able to find the best placement for individual labels, but solvers presented in this thesis are interested in finding the best possible placement of all labels on the map, and hence they have to minimize a fitness function representing the quality of the whole placement.

Proposed fitness function for evaluating the placement of all labels on the map is quite straightforward – the total fitness value is equal to the sum of fitness values of all individual labels as they're currently placed in the map.

This fitness function is used in all algorithms mentioned further in this thesis, and to evaluate the maps in Chapter 9.

Chapter 5
Greedy algorithms

One of the simplest (but also very common) programming paradigms for solving various algorithmic problems is the application of greedy algorithms [23]. Greedy algorithms generally choose the best option available at a time, without reconsidering any decisions previously made.

Obviously, it is possible to reconsider the options by running the algorithm again in an iterative manner, but the basic paradigm is still the same – within one iteration of the algorithm, any choice is "fixed" and cannot be changed until the next iteration.

These properties make greedy algorithms quite fast, but also very fragile as the solution can easily fall into a local minimum and it is not very likely that it will ever get out of it. Local search based on iterative variant might help to handle this problem, but there are still no guarantees of success (and it also slows down the process, as it forces the algorithm to reassign the same variables again and again).

There are three algorithms presented in this chapter. Two of them are single-iteration algorithms, as they only assign candidate position to each label once. First of these two algorithms is simple and fast, as it calculates all important metrics in the beginning, without changing them during the optimization. Second algorithm presents a possible improvement by updating the metrics during the optimization process, based on the choices previously made by the algorithm.

Third presented algorithm makes use of the "GRASP" paradigm, which starts with a greedy algorithm to find an initial solution, and then improves the solution using a greedy based local search. This solution is inspired by works of Aleš Kobr [4] and Brazilian scientists Cravo, Ribeiro and Lorena [22], but uses the improved version of the greedy algorithm as an initial step.

5.1 Basic greedy

The first greedy algorithm initially computes metrics for all candidates of all labels (while all possible conflicts with other candidates belonging to other labels are counted). Later, the algorithm passes through the list of all candidates in the ascending order defined by their metrics and assigns them to their labels, if they were not assigned any other candidate so far.

The implementation available as a part of this thesis internally uses min-heap priority queue for sorting and processing all label candidates one by one. It would be also possible to use standard arrays (or similar data structures) and apply any other sorting algorithms.

All metrics calculated in the preprocessing step of the algorithm are final and will not be recalculated during later phases. This obviously degrades the quality of solutions (as the metric considers conflicts which cannot happen anymore as the affected candidates could not be selected), but makes the algorithm very fast.

Pseudocode of the Basic greedy algorithm is quite simple:

```
// Calculate penalties
foreach candidate1 in candidates:
   foreach candidate2 in candidates:
      if candidate1.label != candidate2.label:
         candidate1.penalty += intersection(candidate1, candidate2)

// Add candidates to priority queue
pq = new priority queue
foreach candidate in candidates:
   pq.push(candidate.penalty, candidate)

// Process queue
for candidate in pq:
   label = candidate.label
   if label.selected_candidate == null:
      label.selected_candidate = candidate
```

Analysis and complexity

Since the Basic greedy algorithm is expected to be deterministic (depending on data structure used for storing labels – using lists would make the algorithm deterministic, while using sets may not, and also on selection of the algorithms used for sorting the list/queue of candidates), the complexity of the algorithm can be found easily.

In order to find out the complexity of the Basic greedy algorithm, let's assume that L is a set of all labels and C is a set of all possible label positions (candidates) available in the problem. Consequently, $|L|$ is the number of labels and $|C|$ is the number of candidates in the problem.

As a first step, the algorithm calculates conflicts between all pairs of candidates. Assuming that there are $|C|$ candidates present in the problem, the complexity of this preparation step is

$$O(|C|^2)$$

For each of these pairs, it is necessary to calculate their intersection, which could be quite computationally expensive, but the complexity of a single intersection calculation is expected to be negligible in comparison with the total number of candidate pairs to be processed.

According to the pseudocode, the second step puts all candidates to the priority queue. If it is possible to put the whole list of candidates to the list at once, the complexity could be lower, but in case of inserting the candidates one by one, the complexity is

$$O(|C| \cdot \log |C|)$$

Finally, in the last step the candidates are popped out of the priority queue and processed one by one until all of the labels have been assigned some candidate. The complexity of popping all elements from the priority queue is again

$$O(|C| \cdot \log |C|)$$

If the algorithm would use an array instead of priority queue, the complexities of pushing/popping new elements would be lower, but there would be additional cost for sorting the array.

Ultimately, the whole algorithm analyzed above have a complexity of

$$O(|C|^2)$$

as the calculation of all label conflict penalties is the most expensive part.

5.2 Advanced greedy

The speed of Basic greedy algorithm is awesome, but since the label conflicts are calculated only at the beginning (and they are calculated considering all possible candidates), they are getting more and more obsolete as the algorithm progresses – most of the possible label candidates will get deprecated over time, but the penalty would be still considered. In order to fix this issue, the Advanced greedy algorithm decreases penalties of candidates affected by selecting or rejecting other candidates.

The first part of the algorithm is exactly the same as in the previous case – intersection penalties are calculated for all possible candidate pairs.

Unfortunately, priority queue is not exactly the best option for storing objects with variable penalties (because with each penalty change, the algorithm would have to find the original element in the queue, pop it and enqueue it again – there is no easy way to decrease key/penalty of existing item in most priority queue implementations). Because of this, the list of candidates is not stored in a priority queue, but it is just a generic array (or list structure), where the candidate with the minimum penalty has to be found by iterating through the whole list.

The other half of the algorithm is also quite similar to the Basic greedy – but, obviously, without having the minimum element easily found by the priority queue, it has to be found by traversing through the whole candidate list in each round. However, the most important difference happens when the candidate is assigned to a specific label – as the selection of a specific label candidate automatically rejects all other candidates belonging to a specific label, the algorithm finds all candidates conflicting with those newly rejected candidates and decreases their penalty. This way, penalty always reflects only intersections with candidates that are still in the game, and does not consider intersections that could not happen anymore.

The algorithm is described by the following pseudocode:

```
// Calculate penalties
foreach candidate1 in candidates:
   foreach candidate2 in candidates:
      if candidate1.label != candidate2.label:
         candidate1.penalty += intersection(candidate1, candidate2)

// Create list of labels without candidate
unassigned_labels = new list
unassigned_labels.push_all(labels)

// Process candidates (always the one with minimum penalty)
while unassigned_labels not empty:
   best = null
   foreach candidate in candidates:
      if candidate.label in unassigned labels:
         if best == null or candidate.penalty < best.penalty:
            best = candidate
   label = best.label
   label.selected_candidate = best
   foreach denied in label.candidates where denied != best:
      // Recalculate penalties for relates of denied candidate
      foreach related in find_related_candidates(denied):
         related.penalty -= intersection(denied, related)
```

Analysis and complexity

Identically to the Basic greedy, the Advanced greedy algorithm is also expected to be deterministic if the data structures used for storing labels and candidates preserve the order of elements stored inside (once more, this rule may not always hold e.g. for sets).

Same as in the previous case, in order to find out the complexity of the Advanced greedy algorithm, let's assume that L is a set of all labels and C is a set of all possible label positions (candidates) found in the problem. Consequently, $|L|$ is the number of labels and $|C|$ is the number of candidates in the problem.

The first step (calculation of conflicts between all pairs of candidates) remains exactly the same as in the Basic greedy algorithm. Assuming that there are $|C|$ candidates present in the problem, the complexity of this preparatory step is

$$O(|C|^2)$$

For each pair of candidates, it is necessary to calculate their intersection, which could be quite computationally expensive, but the complexity is expected to be negligible in comparison with the total number of candidate pairs being processed. It may also be a good idea to note down the list of related candidate pairs, in order to be able to find them quickly during the following steps.

On the other hand, the selection of candidates and application of this choice is much more complicated here (in comparison with the Basic algorithm). In this part, there are $|L|$ labels to be assigned and processed, causing $|L|$ rounds to be run. For each of them, it is necessary to check $O(|C|)$ candidates to find the one with the lowest penalty, causing this part to cost

$$O(|L| \cdot |C|)$$

In the same $|L|$ rounds, the best candidate found is assigned to the label it belongs to and $O(|C|)$ candidates are marked as deprecated (unless there is a specific candidate limit known for each label). For each of these deprecated candidates, there are $O(|C|)$ related candidates and for each of them the metric is decreased, as the conflict between the deprecated candidate and the related candidate is not possible anymore. Complexity of this step is then

$$O(|L| \cdot |C|^2)$$

However, it is really important to realize that there are like to be only a few candidates for each label and only a few candidates related to them, so the complexity looks horrific but with real data it is not so dramatic.

Ultimately, the complexity of the analyzed algorithm is

$$O(|L| \cdot |C|^2)$$

as a consequence of the last part, which is the most computationally expensive one.

5.3 GRASP

The Advanced greedy algorithm is nice – it still works quite fast in comparison with other types of algorithms, and the result usually is not so bad – but it definitely can be much better! The logic behind the decisions made in the Advanced greedy is simple – consider all conflicts with already placed labels, together with all possible conflicts with candidates belonging to the not yet placed labels.

Obviously, this heuristic is quite pessimistic as most of the considered conflicts will never happen. In comparison with the Basic greedy, it is improved in the fact that those candidates are no more considered when they are denied, but it is still not perfect.

Let's consider a situation after the greedy algorithm has finished (i.e. there is a position assigned for each label). Now, if the algorithm would try to improve the solution and select better position for a specific label, the decision would be faster as the algorithm can only calculate conflicts with other currently placed label positions (and does not have to consider positions with other, currently unused positions). This way, after the initial position is found (constructed), the algorithm could go through all labels and check if there is a better placement choice for them, based on current situation (and this way it will likely improve the solution).

This metaheuristic is known as **Greedy Randomized Adaptive Search Procedure (GRASP)** [24]. It is a pattern consisting of two basic steps:

- **Greedy (randomized) construction** is a way how to obtain some solution. Sometimes, the greedy solutions are good enough to be an optimal solution, but in this case, it is perfectly enough to construct some feasible solution.
- **Local search** then improves (or at least tries to improve) the solution constructed by the greedy algorithm. Various kinds of local search methods could be used, including some greedy ones (in that case, the GRASP results in a greedy optimization for constructing initial solution, improved by another greedy optimization step).

Usually, GRASP runs these two steps many times, as the greedy construction is expected to be randomized. In that case, it constructs multiple solutions and each of them is then improved by the local search. However, due to randomness, the algorithm is not deterministic.

In the algorithm designed and implemented as a part of this thesis, only one iteration is applied – one solution is constructed and then improved. Both parts are also deterministic (under the same circumstances as are the previously described Basic and Advanced greedy algorithms):

- **Greedy (randomized) construction** step is equal to the Advanced greedy algorithm, i.e. the best label position is determined, based on conflicts with both already placed labels and all not yet denied label positions of unplaced labels.
- **Local search** step iteratively traverses through the list of labels and for each of them, it checks whether there is a position that would fit better, considering current positions of other labels. If so, the label is moved and hence the fitness will improve in most cases. Number of the local search iterations (where each iteration consists of one pass through the list of labels) is limited by a constant number, but the improvement process can be terminated earlier if there is zero number of changes done in a specific iteration, because any improvement in subsequent iterations is impossible – due to the determinism of the algorithm, the next iteration would result in the exactly same result (i.e. no changes at all).

Despite the fact, that GRASP will likely provide better results than just a pure greedy constructive algorithm, there is still a chance of getting stuck in a local minimum, as only one label is being moved at a time, which is quite limiting (like most local search algorithms). Fortunately, the results are quite good (and if the outcome of the Advanced greedy algorithm was considered good, this is even better) and for most situations, they're pretty nice. The algorithm is described by the following pseudocode:

```
// Run the Advanced greedy algorithm as an initial step
advanced_greedy(labels, candidates)

// Now, run N iterations of improving local search
for iteration in 1..N:
   // For each label...
   foreach label in labels:
      // ...find the best candiate...
      best = null
      foreach candidate1 in label.candidates:
         // ...according to other current choices
         candidate1.penalty = 0
         foreach label2 in labels where label1 != label2:
            candidate2 = label2.selected_candidate
            candidate1.penalty += intersection(candidate1, candidate2)
         if best == null or candidate1.penalty < best.penalty:
            best = candidate1
      label.selected_candidate = best
```

Analysis and complexity

Same as in the Basic and Advanced greedy algorithms, it is a good idea to use deterministic data structures for storing data, to ensure the determinism of the whole algorithm.

In order to find out the algorithmic complexity of the GRASP algorithm, let's assume that L is a set of all labels and C is a set of all possible label positions (candidates) found in the problem. Consequently, $|L|$ is the number of labels and $|C|$ is the number of candidates in the problem. Also, let's assume that N is the maximum allowed number of improvement iterations.

The first part of the algorithm is the Advanced greedy algorithm itself, together with its complexity of

$$O(|L| \cdot |C|^2)$$

It would be possible to use another algorithm to find the initial solution as well (possibly faster one), but in most cases the Advanced greedy algorithm is fast enough and finds reasonable solutions, that could be used as a base for the subsequent improvement iterations.

In the second part of the algorithm, there are up to N improvement iterations. In each of these iterations, all of $|L|$ labels are reconsidered, and for each label there are $|L| - 1$ possible conflicts calculated, setting the complexity of this step to be

$$O(N \cdot |L|^2)$$

Assuming that number of iterations is lower than the number of all label possible label positions for all labels, the complexity of the whole algorithm is determined by the first step:

$$O(|L| \cdot |C|^2)$$

In the end, the improvement part of the algorithm generally takes shorter time than the search for initial solution. Knowing this, the GRASP combination of a the constructive greedy part and the improving local search part is better option than "just" the single-round option, as the extra performance required for the improvements is not so high (at least for reasonable number of local search iterations).

Chapter 6
Mathematical optimization

Even though the greedy algorithms can find the solution quite fast, the solution quality could be very poor. On the other hand, there are ways to find exactly the best (and therefore the most optimal) solution – without any dependence on non-deterministic algorithms or approximations.

One of these ways is describing the task as a mathematical optimization problem and solving it. Unfortunately, optimality guarantee is computationally hard task and therefore very time demanding. But there are many commercial or non-commercial solvers, able to apply various optimizations and heuristics to find the optimal solution without searching the whole solution space.

6.1 Mathematical programming

Mathematical optimization (or mathematical programming) is a problem of finding the best assignment of variables in order to optimize some objective – most commonly to minimize or maximize the value of some objective function, dependent on a set of variables, while ensuring that additional constraints (if given) will hold.

There are many subfields of mathematical programming, but for the purpose of this thesis, the following subfields (or their combinations) are the most interesting ones:

- **Linear programming** is a subset of mathematical programming, where the objective function is linear and all constraints are linear equalities or inequalities.
- **Quadratic programming** is a subset of mathematical programming, where the objective function can be quadratic, but all constraints are still limited to be linear equalities or inequalities.
- **Bilinear programming** [25] is a subset of quadratic programming, where the objective function contains product of two different variables. There are special heuristics for solving this type of tasks, such as fixing one of the variables and hence simplifying the remaining problem to be a linear one.
- **Integer programming** is a subset of mathematical programming, where all of the variables can be assigned only integer values. It is quite common to combine linear and integer programming, known together as Integer linear programming (ILP) task.
- **Mixed integer linear programming** is a subset of linear programming, where some variables can be assigned only integer values (like in ILP), while other variables can be assigned non-integer values as well.
- **0-1 integer programming** is a subset of integer programming, where the variables can be assigned only 0 or 1, making them binary variables.

One more related field is **constrained programming** [26], which is not a subset of mathematical programming, but may allow to solve problems that could not be solved using mathematical programming techniques, as it allows more complex constraints and offers different solving methods.

Problem variations described in this thesis utilize the 0-1 integer linear and bilinear programming techniques, as the decision task is always about choosing some label position or not (restricting the variables to be binary), and calculating the fitness based on individual positions of labels or pairwise relations between labels.

6.2 Branch and cut

In order to solve the mixed integer linear programming tasks, most solvers use an algorithm called Branch and cut [27].

Since the (mixed) integer linear programs are just "standard" linear programs with additional integrity restrictions, the solver first solves a relaxed version of the program, which ignores all of these integrity restrictions. Those relaxed linear programs may be solved using simplex method.

When the optimal solution of the relaxed problem is known, the solver has to check whether the solution complies with integral restrictions. If so, then the solution is the optimal solution of the whole MILP program.

However, in most cases this does not hold and some variables have decimal value. In that case, one of those variables with decimal values is chosen (possibly with the aid of various heuristics and strategies) and the problem is split into two subproblems: for a branching variable x with optimal value of $\bar{x}$, first new subproblem will contain an additional restriction of $x \leq \lfloor \bar{x} \rfloor$, while the other subproblem will contain additional restriction of $x \geq \lceil \bar{x} \rceil$.

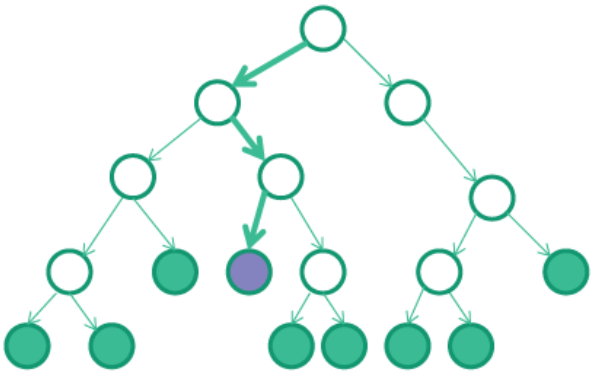


Figure 6.1. Example of branching of the MILP problem[1]

This process creates a huge tree, which the optimizer has to traverse. Fortunately, there are some ways to limit the number of nodes to be traversed. Any subtree of the problem has a solution of it's own relaxed linear program, which cannot be surpassed by any integer solution in that particular subtree (hence its criterion value is a lower limit of that subtree). As the optimizer finds some solutions, it can remember the best criterion value of the (mixed) integer linear program found so far as a global upper limit, because there is no point in finding any solution that is inferior to this one. Knowing this, any subtree which has a lower limit higher than is the global upper limit at a time, is not interesting for next analysis and therefore can be ignored.

[1] `http://www.gurobi.com/resources/getting-started/mip-basics`

6.3 Available solvers

There are tens of various software suites available for solving many kinds of mathematical optimization problems. The following list describes four major tools, three of them being developed as a commercial product, while the last mentioned one is licensed as open source and freely available.

6.3.1 CPLEX

IBM ILOG CPLEX Optimizer[1] is a commercial optimization toolbox currently developed by IBM. CPLEX contains multiple solvers for solving various linear programming, (mixed) integer programming, quadratic programming and constrained programming tasks (the last one mentioned could utilize different techniques than the "pure" mathematical optimization solvers, as already mentioned before).

CPLEX can be integrated into custom applications using APIs available for various programming languages (including Matlab plugin) or it can be used to run tasks defined using various modelling languages, including the OPL (Optimization Programming Language) which is intended for usage specifically with CPLEX.

There are two freely available versions for academic use – CPLEX Optimization Studio Community Edition is the publicly available free edition (with restriction to problems with up to 1000 variables and 1000 constraints). The other edition is a standard CPLEX Optimization Studio available through the IBM Academic Initiative. The latter mentioned is also available at the **MetaCentrum**[2] compute grid.

6.3.2 Gurobi

Gurobi Optimizer[3] is a commercial optimization toolbox developed by Gurobi Optimalization and like CPLEX, it contains multiple solvers for linear, (mixed) integer, quadratic and various forms of constrained programming.

Usage options are also quite similar to CPLEX, as Gurobi offers API for most used programming languages, Matlab and R connectors and it can process tasks described using common modelling languages.

There is academic license available for students and academic institutions, but it is not available on MetaCentrum. Free version is not available.

6.3.3 MOSEK

The last mentioned commercial optimizer is **MOSEK**[4]. Like already mentioned CPLEX and Gurobi solvers, it can solve linear, (mixed) integer, quadratic and other programming tasks. Its main strength is in fast solving continuous linear, quadratic and conic problems.

MOSEK can be officially used through APIs, as a Matlab toolbox and it can execute tasks defined in AMPL (which is also one of the modelling languages also supported by the two previously mentioned suites).

Like in case of Gurobi, there is no free version available, but there is an academic license offered to students and academic institutions.

[1] https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/
[2] https://www.metacentrum.cz/en
[3] http://www.gurobi.com/products/gurobi-optimizer
[4] https://www.mosek.com/products/mosek

6.3.4 GLPK

GNU Linear Programming Kit (GLPK)[1] is a free and open source suite backed by the Free Software Foundation. It is capable of solving various linear programming and mixed integer programming tasks.

GLPK offers bindings for various programming languages and can be also run from command line to solve tasks defined in GNU MathProg modelling language (which is specific to GLPK, but it is a subset of AMPL).

6.3.5 Problem definition

There are many ways to formulate the problem, but let's start with the most common and already known formulation, and reformulate it as a mathematical optimization problem. Slightly less formally, the problem can be described as follows:

"There is a list of labels and corresponding possible label positions (candidates) for each of these labels. For each label, there must be exactly one candidate selected (placed on the map). There is a non-negative penalty for each selected candidate (representing conflicts with map objects), and also a non-negative intersection penalty for each pair of two selected candidates (representing intersections of these labels). The goal is to minimize the total sum of penalties in the whole solution."

This description can be formulated as the following 0-1 bilinear programming task:

$$min \sum_{c \in C} s_c \cdot P_c + \sum_{c_1 \in C} \sum_{c_2 \in C} s_{c_1} \cdot s_{c_2} \cdot I_{c_1,c_2} \tag{1}$$

subject to

$$\forall c \in C, l \in L : s_{c,l} \leq A_{c,l} \tag{2}$$

$$\forall l \in L : \sum_{c \in C} s_{c,l} = 1 \tag{3}$$

$$\forall c \in C : \sum_{l \in L} s_{c,l} \leq s_c \tag{4}$$

The description above uses the following symbols:

- L is a set of label identifiers
- C is a set of label candidate (possible label position) identifiers
- $s_c \in \{0, 1\}$ is a boolean variable representing the choice whether the candidate c is selected or not
- $s_{c,l} \in \{0, 1\}$ is a boolean variable representing the choice whether the candidate c is selected for label l or not
- $P_c \geq 0$ is a non-negative integer penalty for selecting the candidate c
- $I_{c_1,c_2} \geq 0$ is a non-negative integer intersection penalty for selecting both candidate c_1 and c_2
- $A_{c,l} \in \{0, 1\}$ is a boolean flag stating whether the candidate c belongs to label l

Constraints in the definition have a quite straightforward meaning – for each label only its candidate positions can be chosen, for each label exactly one candidate position has to be chosen, and if some position choice flag is true for some label/candidate pair, it is true for the choice of candidate itself as well.

[1] `https://www.gnu.org/software/glpk/`

Linear vs bilinear definition

The problem definition above describes a bilinear programming problem. The good thing about it is that all variables are binaries (0-1 integers). The not so good thing is that the problem is bilinear, instead of just a linear one.

Basically, it is not a problem to convert the task to be a linear one, by simply introducing a variable for each pair of candidates, stating whether both of them are chosen. Unfortunately, it is not very effective because this little change would generate another $|C|^2$ new variables and constraints (at least one constraint for each new variable). For just one thousand candidate positions, there would be another million of additional variables and a million of additional constrains.

Good optimizers can do very effective preprocessing and eliminate most of the unnecessary variables, but many of them will still remain in the problem. For this reason, there is the bilinear definition used in this work, as CPLEX can handle the problem quite well (and much easier than the linear one with many additional variables).

6.4 Using CPLEX solver

For the purpose of this thesis, CPLEX will be used to solve the task, because it is probably one of the most effective software tools available for this class of problems, and offers free academic license. Even more important fact is that this academic license is usable on MetaCentrum cluster, used for comparing algorithms later in this work (as the computer cluster offers much more computational power than all personal computers available during the time of writing this thesis together).

There are few basic ways to solve mathematical optimization problems using CPLEX:

- **Providing definition in a supported format** is the lowest-level usage mode offered by CPLEX. Problem definition stored in LP (Linear Programming) or other supported format has to be passed directly to the solver, which finds out the optimal solution and writes it into SOL (Solution) file, that is XML based and can be parsed by any other software. The LP definition describes the optimization problem in its purest form, without any additional abstractions like arrays or similar data structures.
- **OPL (Optimization Programming Language)** (or other modelling languages like AMPL) represent a higher-level way of using the optimizer. OPL describes the problem using various abstraction techniques like arrays (including multi-dimensional ones), and allows writing the data in more user-friendly manner, using various loops and integrated functions. Ultimately, the OPL can also contain additional scripts, e.g. for writing the output to a file when the solution is found.
- **APIs** can be used to run the optimizer directly from the source code of other applications. CPLEX provides API interfaces for many programming languages like Java or C#, or as a Matlab toolbox. This way it is possible to manipulate the data in more complex and easier ways (in comparison with quite restrictive modelling languages), and there is no need for the additional intermediary step of calling the CPLEX solver (or OPL runner) from a command line.

The software implementation distributed as a part of this thesis uses the OPL models to pass the data to the optimizer. This approach allows more flexibility, in comparison with LP files (such as more comfortable way to describe the problem using arrays and loops, or the ability to write out the solution to a simply readable file like CSV, instead of parsing complex XML files produced directly by the optimizer).

Another option considered during the program implementation was the direct usage of CPLEX through Java API. However, since the problem is likely to be solved on another computer (e.g. high performance server) than the one used to design it, it would be necessary to have the program available on that server as well. With OPL, the program does not have to be installed on the CPLEX machine as only the OPL model file is necessary. After the OPL execution, there is a single file created, which has to be fetched and loaded into the application.

6.5 Terminating the optimization

Mathematical optimization is an exact method and can provide truly optimal solutions. Unfortunately, with the optimality comes the price – there might be enormous amount of time and computational power needed to find the solution, and this factor can be limiting for anybody trying to solve anything more complex than just the most trivial problems.

Because both time and computational power are usually limited (and also connected with each other, as more computational power will likely decrease the amount of time required to find the optimal solution), it is necessary to decide when it's the best time to stop the whole optimization process.

In CPLEX (and likely in the other optimization toolkits as well), there are multiple conditions that can be used to stop the MILP (Mixed Integer Linear Programming) optimization process at the right time:

- **Optimal solution has been found** is the best option if it will happen in a reasonable time. It requires the problem to be either completed by ensuring that any better solution does not exist, and in case of using branch and cut method it requires the whole branching tree to be processed (or cut away). However, for larger problems this is likely to take unacceptable amount of time.
- **Time or memory limits** can limit the time spent on the optimization process, or computer memory used by the optimizer (as the branch and cut tree data can occupy huge amount of memory). When the time or memory limit is reached, the optimizer returns the best result found so far and marks it as a final result.
- **Relative and absolute gaps** are restrictions specific to the branch and cut method. During the optimization process, the difference between the value of the best feasible solution found so far (upper limit) and the value of the best discovered (or expected to be discovered) solution that might still be found (lower limit) is called the "gap". When this gap is small enough, the optimizer process can be finished as searching for more optimal solution may take too much time considering just the little improvement that can be found.

Thanks to these termination conditions, the mathematical optimization is more usable for solving real world problems, as people usually need solution that is "good enough", but in a reasonable time and without having to buy costly pieces of hardware.

6.6 Performance issues

Although CPLEX is a very powerful software and the problem itself is very simplified during the preprocessing part of the optimization process, it still needs a lot of computational resources in order to find the optimal solution of the task.

Basically, there are two steps of the mixed integer programming solution process in CPLEX. In the first step, the problem is reduced by relaxing unnecessary variables and constraints from the problem. In terms of the label placement, the intersection matrices representing penalties for label intersections (conflicts) are expected to be pretty much sparse (as each label usually has conflicts only with few other labels), causing many variables and constraints to be eliminated in this step. The amount of memory needed in the preprocessing step depends on the problem size, and generally is lower than the memory needed further during the optimization process.

The second step is the optimization itself. Depending on the optimization problem type, combinations of various methods like simplex method or branch and cut are used. Using the branch and cut forces the algorithm to consume huge amount of memory as it remembers a lot of data about processed solutions. As mentioned in the previous section, it is possible to set memory restrictions for the memory used by the solver, but according to experiences during the preparation of this thesis, in most cases the memory limits are not really respected.

On MetaCentrum computer cluster used evaluating all algorithms mentioned in this work, there are two options how to handle memory usage limits – either allow the optimizer to take as much memory as it wants, and kill it when these limits are exceeded, or to hard restrict the memory available to the CPLEX process. With the first option, there is a risk that the optimizer will cross the limit before writing out at least some of the results and causing the whole optimization process to be halted without producing any result. In the latter case the optimizer cannot cross its allocated memory limit, but if the available memory is not enough for some optimization step, the optimizer might halt the process as well.

For testing the algorithms on MetaCentrum, the first option is used – there are no hard limits configured for the process, but if the allocated memory is not enough to finish the process, the whole process is halted. In order to successfully solve the problem, it is necessary to assign enough memory allowance to the task.

Chapter 7
Genetic algorithms

In previous chapters, the label placement problem was solved either using quick (but very "fragile") greedy algorithms, or by describing the task as a mathematical optimization problem and passing it to some of the high performance solvers in order to find a solution.

Unfortunately, mathematical optimization is an exact science and optimizers have to consider all possible options in order to be able to guarantee optimality of the solution. This way, the outcome of the algorithm is completely deterministic and will be found for sure. But everything comes with a price – in this case, finding the optimal solution can take enormous amount of time.

On the other hand, what if some non-deterministic algorithm would be used instead? There would be absolutely no guarantees of finding the optimal solution, but there is always a chance of finding quite good solution within a short time periods – but still, no guarantees. Some of the possible representatives of these non-deterministic algorithms are called **evolutionary algorithms**.

Evolutionary algorithms are optimization techniques inspired by biological evolution, and share its basic concepts like selection, mutation and reproduction mechanisms. However, in contrast to the biological form, individuals evolved in evolutionary algorithms are solutions to the given optimization problem. There are multiple types of evolutionary algorithms like **genetic algorithms** (searching for the genotype representing the optimal solution) or **genetic programming** (searching for optimal computer program).

This part of thesis solves the label placement problem using **genetic algorithm** [28]. Moreover, the presented genetic algorithm introduces hierarchical clustering as an enhancement allowing the usage of point based crossover functions. Furthermore, it introduces a memetic part in the form of local search step, which allows faster evolution of promising individuals.

7.1 Parts of genetic algorithm

Genetic algorithm itself is just a metaheuristic inspired by biological evolution and natural selection. However, in order to apply it on a real world (or any other) problems, some basic, problem dependent building blocks have to be defined as a part of the algorithm.

In the following sub-sections, all important parts of genetic algorithms are described one by one, together with commonly used examples.

Genetic representation

In the world of genetic algorithms, all individuals (possible problem solutions) are represented by **genotypes** (or chromosomes). Genotypes are made of variables, known as **genes**. In most cases, genotypes can be represented as strings (either consisting solely of binary values, integers, or other generic values).

In a couple with each genotype, there should exist a **phenotype**, which represents the genotype meaning in the problem solution domain. It is also essential to be able to encode phenotypes to genotypes, and – even more important, to be able to decode genotypes of the solutions back to the phenotypes.

Initialization

As a first step in the genetic algorithm, it is necessary to get the initial **population** of **individuals** (candidate solutions). This population (set of individuals) can be established by many ways – individuals could be generated randomly without the need of any deeper knowledge of the problem domain, or they can even be a product of another (likely fast) optimization algorithm.

Individuals in the initial population should be feasible, forcing the initialization algorithm to respect this condition as well. There are two ways to achieve this property – either by designing generator providing only feasible solutions, or by generating "some" solutions and then fixing them to make them feasible.

Fitness function

In order to compare individuals and decide which one is better (and hence has higher chance to survive), there must be some metric able to score each genotype and assign it a comparable value. In case of genetic algorithm, this metric is called the **fitness function** and generally assigns each individual a real number score.

Since the fitness function is used very often (as the whole evolution process is based on comparing individuals), it is absolutely necessary for it to be fast. It is a very good idea to use various caching techniques to reduce the number of fitness calculations as much as possible, but in case of very complicated fitness functions it may be also a good idea to use reasonable level of approximations in order to speed up the process.

Selection operator

In each iteration, the algorithm creates a new **generation** – population evolved from the population produced in the previous iteration. Population is evolved using crossover and mutation operators as described further, but in order to perform the crossover, it has to be somehow decided which individuals from the previous generation are so good that they should be preserved or passed to the crossover process.

When talking about the possibility of preserving a candidate from the previous generation without modification, this option is called **elitism** and it may be a good idea to "pass" few best candidates from the previous iteration without having to go through the crossover and mutation process, as the candidate could be degraded during these steps and the good candidate would be lost, without ability to recover it and use it anymore in subsequent iterations.

There are many ways to **select** individuals from a generation and here are some of them:

- **Random selection** is for sure the easiest option – candidates for the crossover are selected randomly, even without looking at their fitness values (which can, on the other hand, be quite performance consuming).
- **Roulette wheel selection** is more advanced method, where the options are given a probability of being selected equal the ratio of the fitness function value for given individual, and sum of fitness values of all individuals in the population. Candidates for the next generation (either for elitism or crossover) are then selected according to these weighted probabilities.

- **Tournament selection** consists of two steps. In the first step, k individuals are randomly selected from the previous population. Selected individuals are then sorted by their fitness and the winner is selected randomly with a given probability p (in the manner that the best candidate has a chance p of being selected, and in case of not being selected, then the second best candidate has a p chance of being selected, and so on). In a special case, tournament selection can become just a pure random selection (when $k = 1$) or the second ("tournament") part can be skipped by always selecting the candidate with highest probability (when $p = 1$).

Crossover operator

Finally, when two (or possibly even more) parents are selected for the **crossover** step, it is time to combine them to produce a new individual (child, also called an offspring). In most cases, the crossover step is designed so that two parents are transformed to two new individuals, but depending on the selected crossover operator these attributes can vary.

In the following list, there are some of the most common crossover operators, but the choice (or design of custom operator) is highly dependent on the chosen genetic representation and the problem definition.

- **Single-point crossover** is easily applicable technique, usable for most string genotype representations. A single position in the genotype (crossing point) is selected and each child receives the first part (from the beginning of the genotype to the selected crossing point) from one parent, while the other part (from the crossing point till the end of the genotype) is copied from the other parent.
- **Two-point crossover** is similar to the previous one, but there are two crossing points. Therefore, first child will receive first and last part from the first parent, and the middle part from the other parent. The other child will be created in the same manner, taking middle part from the first parent and the rest from the other parent.
- **Uniform crossover** might look a bit drastic, but in many cases, it could be quite effective to choose each separate gene randomly from one of the parents. This way, crossover is done separately for each gene, instead of crossing only whole segments as in case of single-point or two-point crossover.

After applying any crossover operator, it is necessary to check whether the generated genotype is feasible (i.e. if the corresponding phenotype has a meaning in the problem solution domain and if no constraints are violated). If the produced solution is not feasible, it has to be repaired (if possible) or trashed.

Mutation operator

After performing the crossover part of the reproduction process, there is one more way to improve (or worsen) the newly created children individuals – **mutation**. In this step, some genes in the child's genotype can be changed to different genes.

Some basic universal options are flipping randomly selected genes to another ones (especially in combination with binary string representation of genotype), either randomly selected or acquired using other techniques (lower/upper bounds, or averages).

However, mutation operators could be much more powerful when they are tailored directly to meet needs of a specific problem, as the mutation is going to be much more meaningful in terms of the problem domain. Also, same as in the crossover step, the mutation outcome has to be feasible, which is much more likely to be fulfilled with custom mutation operator than just a generic one.

Termination condition

When the algorithm is in progress, it iteratively evolves the generations, one by one. But at some point in the time, this whole lifecycle has to stop. Here are some basic **termination conditions**, which could be used to determine the right time for stopping the algorithm:

- **Fixed number of generations** is probably the most intuitive termination condition, and it is common to use it as an "emergency break" in combination with other options.
- **Optimal solution is found** is a very nice and powerful condition, but only in case when the algorithm is able to recognize the optimal solution – which is usually not easy.
- **Fitness function call count** can be restricted in a similar manner as the number of iterations.
- **The solution is not improving anymore** (or the improvement is very small) for some number of generations, and therefore any further improvement in the subsequent iterations is considered unlikely (however, this condition is quite dangerous due to stochasticity of genetic algorithms).

7.2 Genetic algorithm for label placement

Previous section generally described the structure of the genetic algorithm. This section describes the genetic algorithm designed specifically for label placement problem.

The algorithm follows the basic scheme as illustrated in the Figure 7.1:

1. Initial population is generated and the fitness is evaluated for each individual
2. In each iteration, new parents are selected and used for crossover. Afterwards, mutation is applied to the offsprings and the fitness values are evaluated for each newly created individual.
3. The whole process continues, iteration by iteration, until the termination condition stops it.

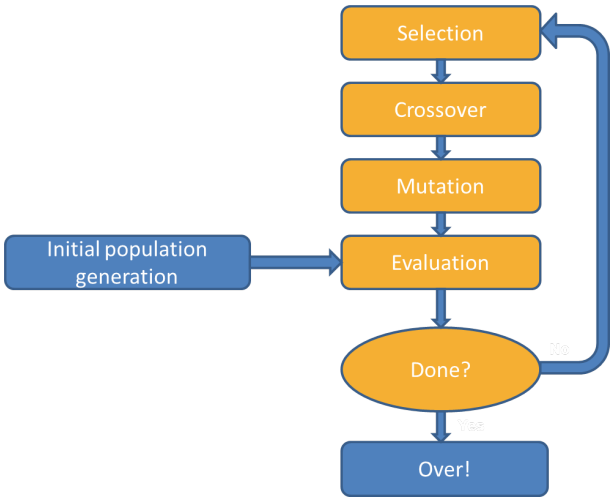


Figure 7.1. Basic scheme of a genetic algorithm[1]

[1] https://genetic.io/en/introduction-genetic-algorithms/

7.2.1 Building blocks

In the previous part, there is a list of generic building blocks of genetic algorithms. In this part, specific components used in the implemented algorithm are described in detail.

Genetic representation

The problem being solved by the algorithm is simple – every label has to be assigned a candidate position, which has to be chosen from the list of previously generated candidates.

Each label can be assigned a sequence number, representing its position in some integral list of labels. In a similar manner, each label position candidate can be assigned a sequence number representing its position in a previously generated candidate list. This leads to a natural choice of a genetic representation – an array of integer values, where i-th value represents candidate choice for i-th label.

However, there is one more question remaining to be answered – would there be a difference, if the labels in the genotype would have any other order? In fact, there could be a big difference, especially in combination with some operators like single-point or two-point crossover. In case of random label numbering, these types of crossover will likely do more harm than good, as the crossover would just combine two randomly chosen sets of label positions.

Better idea might be to order labels, so that labels near each other ("clusters") would be placed together. In that case, if the crossover chooses a crossover point on the border between two clusters (or near to it), there is a chance of using one half of genotype describing good placement of labels in specific clusters, and joining it with half of another genotype representing good placement of the remaining clusters.

To allow this, the algorithm presented in this thesis uses **hierarchical clustering** [29] to find a good order of labels for a genotype. There are many other techniques available (various kinds of clustering, bandwidth minimization algorithms, etc.), but hierarchical clustering was chosen because it does not need any prior knowledge about the number of clusters and it is easy to use. As a distance function, the number of conflicting label candidates between two labels is used (in a slightly modified way, so that the highest number of conflicting label candidates represents the nearest labels).

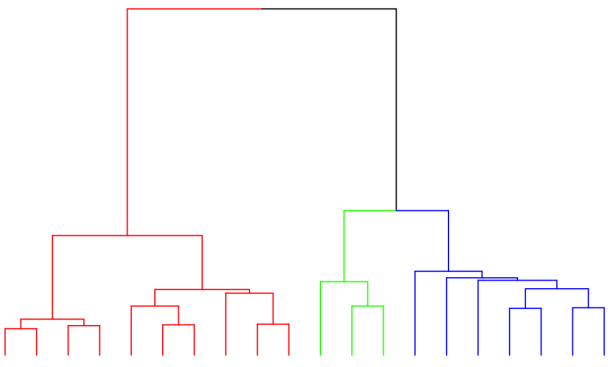


Figure 7.2. Outcome of hierarchical clustering represented as a dendrogram[1]

For the implementation available as a software part of this thesis, hierarchical clustering is provided by the freely available Java library by Lars Behnke[2].

[1] http://flybrain.mrc-lmb.cam.ac.uk/si/nblast/www/nblast_desktop/
[2] https://github.com/lbehnke/hierarchical-clustering-java

Initialization

Now, when the genetic representation (or gene mapping) is known, it is time to create an initial population of individuals. In this algorithm for label placement, it is undesirable for any candidate to be infeasible, as there would have to be a method to "repair" individual genotypes during the evolution.

That said, all individual genotypes at any step of the evolution should be feasible. In terms of population initialization, the easiest way is to generate random individuals by randomly assigning one of the available candidates to each label.

There are more sophisticated ways to generate individuals for an initial population, like local search – which is, in fact, used in the presented algorithm in this thesis as well. This is described later in the section about memetics.

Fitness function

Fitness function of this algorithm is exactly the same, as the one used in the whole thesis – each label candidate position has a metric value consisting of a static part (representing conflicts with other map objects, whose positions do not change during the placement process, and also penalties for representing a less preferable positions), and a relative part (representing conflicts with other label positions – or label candidate positions, which will likely change during the optimization process).

The fitness function then sums all penalties (both static and relative) of candidate positions selected for all individuals, weighted according to the given metric definition (containing weights of label conflict penalty, map object penalty and label position penalty).

Since the fitness function value is not cached during the evolution, all penalties for conflicts between two labels are recalculated in each iteration. In case of more complex geometries (e.g. polygons with high number of vertices), this might be computationally expensive, but in most cases it is easier than storing data about all possible overlaps.

Selection operator

Selection in this algorithm is quite straightforward, as it uses the classic **tournament selection** with 4 randomly generated individuals considered for each selection, and the one with the best (i.e. lowest) fitness value is always selected.

This selection operator is quite common and often used in many genetic algorithms.

Crossover operator

Crossover is probably the most dangerous part of the whole genetic optimization process, as there is a high chance of "breaking" the solution. First of all, it is necessary to ensure the feasibility of the solutions produced by crossover – this is not difficult as currently used genetic representation allows only a specific set of values for each gene, and there are no additional constraints to be held.

Algorithm uses a **single-point crossover**, which combines two parents into two offsprings. First offspring has the first part of genotype (ending at arbitrary crossover point) from the first parent, and the remaining part of the genotype from the other parent. The second offspring has first part from the second parent, while the other part from the first parent. Crossover is illustrated in Figure 7.3.

Thanks to the usage of clustering in the preparation of genetic representation, crossover has a chance to be processed in a point near to the border between clusters, and the offsprings can possibly benefit from getting two good parts merged together.

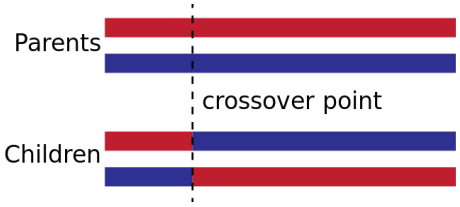


Figure 7.3. Single-point crossover[1]

However, due to possible dramatic effect on the fitness of the offsprings, the crossover is processed only sometimes, with limited probability. In the remaining cases, parents are just copied and passed to the next step.

Mutation operator

During the mutation step, all labels are processed one by one. Each label has a quite low probability that its gene will get changed to another randomly chosen (but feasible) candidate.

Termination condition

Termination condition is quite simple – number of iterations of the algorithm is limited. It would be possible to implement other conditions, e.g. stopping after some number of iterations without any fitness improvement. However, it would be necessary to have in mind that many improvements happen during the memetic part described further.

7.2.2 Memetics

In fact, the algorithm implemented as a part of this thesis is not just a pure genetic algorithm, but it is an example of a **memetic algorithm** [30]. This kinds of algorithm is a combination of two approaches – evolutionary algorithm (e.g. a genetic one) and a local search method, like hill climbing or simulated annealing.

In this label placement algorithm, local search is injected into two phases:

1. During the initialization of the first population, some of the randomly generated individuals are improved by the local search
2. During the evolution, once per an arbitrary number of generations, few randomly chosen individuals are improved by the local search and passed to the next generation

The local search method used for improving the individuals is exactly the same as the one used in the GRASP. The algorithm iteratively traverses through the list of labels and for each of them, it checks whether there is a position that would fit better, considering current positions of other labels in the same solution (individual). If so, the label is moved by changing the value of a gene representing the label.

[1] `https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)`

Chapter 8
Software implementation

Integral part of this thesis is the implementation of all mentioned algorithms, and user interface providing comfortable way to test and benchmark them. In this chapter, the whole software part is briefly introduced, together with description of used technologies and available features.

In order to use any of the label placement algorithms, user needs "only" one essential thing – the data. The best way to try the proposed algorithms is to bring already prepared data set, and transform it into one of the formats supported by the "Label Placement" application. When the data is loaded, application allows user to display all geometric objects and labels present in the map, and to perform basic modifications. Ultimately, as the most important feature, the software allows user to apply available labeling algorithms to the data and visually evaluate the output.

In order to try the algorithms without having data in advance, it is possible to import map objects existing maps in ESRI Shapefile[1] format, which can be obtained from sources later described in Evaluation and results section.

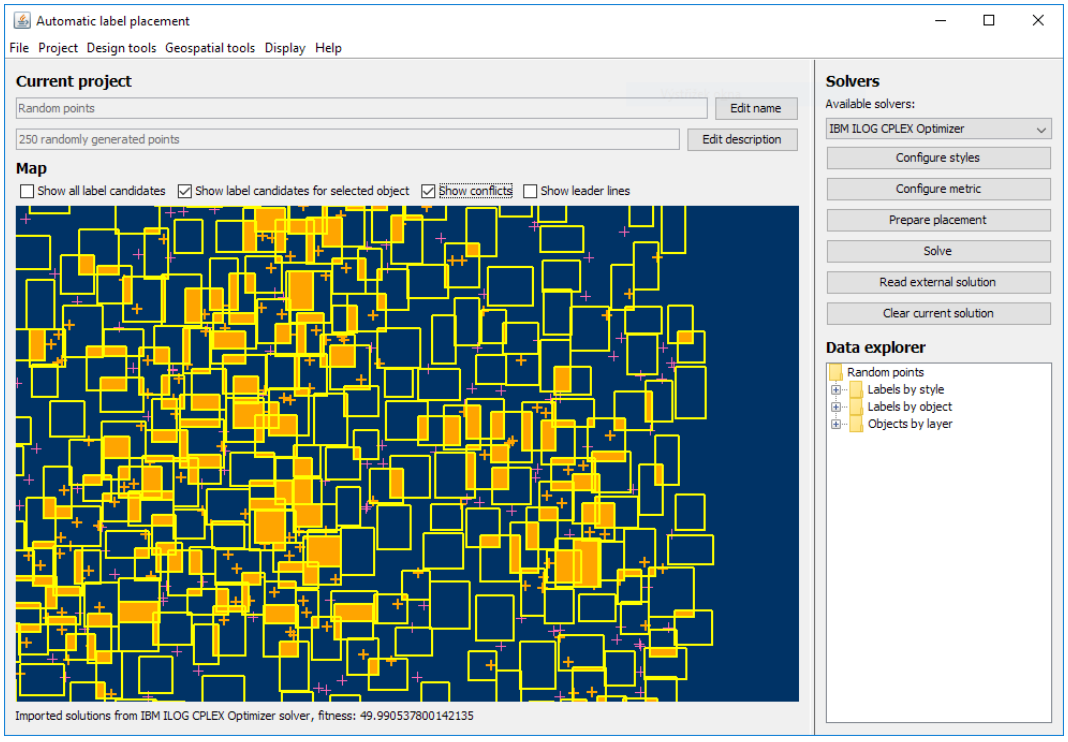


Figure 8.1. User interface of the Label Placement application

[1] https://www.esri.com/library/whitepapers/pdfs/shapefile.pdf

8.1 Used technologies

In the client part of the application, many technologies and frameworks were used. These are the most important ones:

- **Java SE**[1] is an imperative object-oriented programming language and platform originally developed by Sun Microsystems and currently maintained by Oracle. Java language is popular because of its platform independence – code is compiled into a bytecode, that can be run on almost any commonly used platform. Java language and the platform itself is known as Java SE (Standard Edition), while the web a business logic technologies are known as Java EE (Enterprise). In this work, all algorithms are implemented in Java SE 7, together with the user interface (and so are all additional libraries used in the project).
- **Maven**[2] is a build management and project management utility designed primarily for Java development. It maintains the whole build process, including the compilation itself, unit testing and possibly even deployment of the final packages intended for release. The whole label placement application is developed as a set of Maven modules, distinguished by their purpose – core (with geometries, problem definitions, etc.), user interface module, solvers and geospatial tools for working with Shapefiles.
- **Jackson**[3] is a library allowing to easily work with JSON files in Java. It allows to serialize and deserialize objects to/from JSON and few other formats, including XML and YAML. All these three formats are supported by the Label Placement application, as described later.
- **GeoTools**[4] is a library for working with geospatial data in Java. It can work with many geospatial data formats including Shapefile, which is interesting for importing data into the Label Placement application. GeoTools provide access to many geospatial and geometric operations, but the only feature used in this work is loading files and parsing geometries with their metadata.

8.2 Features

In the following sections, there is a brief description of major features implemented in the label placement application.

8.2.1 Available algorithms

Probably the most important part of the program is a set of available label placement algorithms. The list of algorithms follows and corresponds to the algorithms described in previous chapters:

- **Random** is just a test algorithm, which assigns randomly selected candidate to each label.
- **Greedy Basic** is a fast version of greedy algorithm, based on initial calculation of possible conflicts between all pairs of the candidates, and subsequent processing of the candidates one by one, starting the one with the lowest metric.

[1] http://www.oracle.com/technetwork/java/javase/overview/index.html
[2] https://maven.apache.org/
[3] http://wiki.fasterxml.com/JacksonHome
[4] http://www.geotools.org/

- **Greedy Advanced** is similar to the Greedy Basic, but the Advanced version recalculates metrics for remaining unassigned labels, based on choices made for the already assigned labels.
- **GRASP** is based on Greedy Advanced, which is used to generate the initial label placement. After some solution is found, the algorithm iteratively traverses through all labels and tries to find better positions for them, based on current arrangement of other labels.
- **IBM ILOG CPLEX Optimizer** is a deterministic option for those who want optimal results, but are willing to wait some time. Label Placement application generates OPL module, which has to be run by the user on a server or workstation equipped with the commercial or academic version of CPLEX. The result is saved into a CSV file, which can be imported back to the application and displayed. By default, OPL contains instruction to terminate the calculation process after one hour and return the best solution found so far (if the process did not finish yet), but this restriction can be lifted by manually altering the OPL script.
- **Genetic algorithm** is non-deterministic algorithm, which has a chance to find a good result in a short period of time, but there are no guarantees at all. While the algorithm works, it is possible to observe the progress, as the best solution found so far is periodically rendered to the screen.

8.2.2 Candidate generation

As a preprocessing step shared for all of the listed placement algorithms, it is necessary to generate list of possible positions (described as "candidates" in this work) for each label. This process is executed only once (unless the user executes the preparation again, because of style or metric changes), and generates possible candidates based on placement style configured for each object (either individually for specific objects, or commonly for the whole layer or even whole project).

The following rules and styles are used for generation of label candidates:

- **Points** – for each feature point label, there could be either basic four (top-left, top-right, bottom-left, bottom-right) or eight (basic four + top, bottom, left, right) positions generated. Their priorities (labels with higher priority have lower label position penalty) are defined by rules described previously in Chapter 3.
- **Lines and polylines** – line feature labels have more complex rules. All generated candidates could be rotated either along the line (or specific part of a polyline), perpendicular to the line or they can be just placed horizontally. Labels could be placed above, below or directly on the line and the preferred label position could be on the left end, right end or in the center of the line (polyline) – again, this preference determines the label position penalty later considered by the placement algorithms.
- **Rectangles and polygons** – label is always placed to the middle of the object.

Placement rules for labels belonging to rectangles and polygons were omitted from this thesis, as they would be either too complicated (because of various possible shapes and sizes of those map objects), or they would be non-deterministic (which is not very handy, considering that some most of the examined placement algorithms are deterministic, so it would be pointless to give up this property by just choosing bad preprocessing methods). In that case, generating a good set of candidates could be considered as another optimization problem, which is out of this thesis's scope.

8.2.3 Map editor

Even though the map data are usually expected to be imported from external data source, there is also a simple editor integrated in the Label Placement application to allow creating or removing map content.

There are few basic geometry types supported – point, line, polyline, polygon and rectangle (which is, in fact, also a polygon). Drawing tools are available for all of them, and it is also possible to automatically generate default labels for newly created objects.

In order to simplify work with map features and labels, each map feature belongs to a named layer intended for holding objects with some level of similarity (type, location, etc.), and each label is attached to a specific map feature.

Another level of distinction is style, which defines basic placement rules for a given set of labels. Styles can be applied for specific labels, layers or a whole project (in case of conflict, the one nearest to the label level is used), and they describe placement rules for labels belonging to all kinds of supported map objects.

8.2.4 Metrics and styles

Application contains basic editors for metrics and styles, which define supporting rules for candidate generation and the label placement itself.

Metric editor allows to define weights of three basic penalty factors:

- **Label position** for penalization of not ugly positions, with default weight of 10
- **Conflicts with other labels** for penalization of overlaps between labels, with default weight of 50
- **Conflicts with map objects** for penalization of overlaps between labels and other map features, with default weight of 40

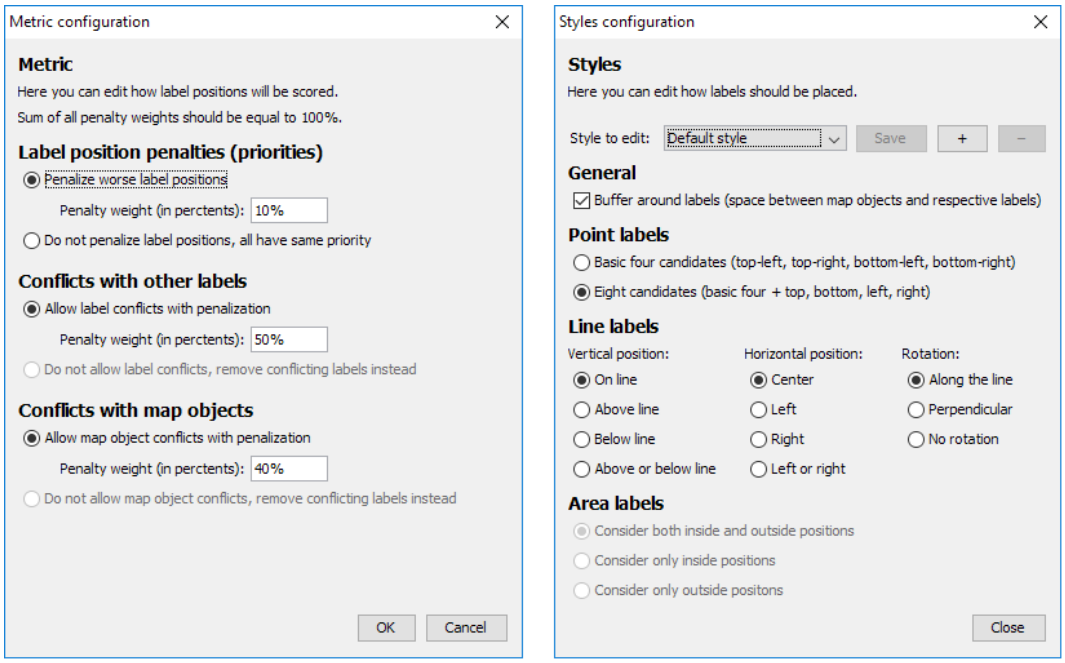


Figure 8.2. User interface of metric and style editors

Like the metric editor, style editor allows to modify styles that are later applied to the labels or layers and used for candidate generation.

8.2.5 Geospatial data import

In order to import existing real data into the application, it is possible to make use of Shapefile import capabilities. Using the GeoTools library, geometries can be imported as map objects. By analyzing the metadata attached to the geometries, it is possible to automatically classify the meaning of objects being imported, and the corresponding labels can be generated as well.

Shapefile (which, in fact, consists of multiple files with geometries and metadata) contains a definition of geometries found in the map (similar to the geometries internally used by the Label Placement application, but usually a bit more complicated) and metadata about each object. Content of the metadata varies between data sources, some maps contains tens or even hundreds of parameters about each object, but some of them contain just a handful of these.

Unfortunately, due to severe differences in structure of various map data, it is necessary to have a specific "view" describing scheme of each file being imported. As a part of the implementation, few views corresponding to selected Shapefile sources are available and applicable to the Natural Earth data, described later in the Evaluation and results section. Additional feature of these views is filtering – each of them allows to filter the imported data based on some criteria, like the name of the country or the continent.

8.2.6 Data persistence

In order to provide standard save and load functionality, Label Placement client allows to export and import data in three commonly used data exchange formats:

- **XML**[1] (eXtensible Markup Language) is a markup language standardized by the World Wide Web Consortium (W3C) as a platform independent format for publishing various data (this is the reason why it is called eXtensible) and transferring them between different computer systems. Along the XML data format itself, there are also many other technologies inside taking part in the XML ecosystem, like XML schemas describing the content form and structure, XSLT transformations (which easily transform XML data into another form, even a human readable one like HTML) and XQuery for executing queries on the data.
- **JSON**[2] (JavaScript Object Notation) is a lightweight portable data format originally designed with a similar syntax as the one used for describing objects in JavaScript. Nowadays, it is a very popular format for storing and transferring data, especially in combination with REST web services. In comparison with XML, it has much smaller overhead, but it is schemaless and not so strict as is XML.
- **YAML**[3] (YAML Ain't Markup Language) is a format offering more or less the same possibilities as JSON does, but it is also clearly human readable. XML or JSON can be well readable be a person when it is nicely formatted, but YAML is nicely readable by definition. It is also schemaless, but usually has a larger overhead than JSON.

Using these more or less standard data formats also allows to exchange data (especially map data) with other applications.

Due to complexity of existing file formats for storing map data, this application uses its own data structure, that is simplified as much as possible to contain only the

[1] `https://www.w3.org/XML/`
[2] `http://www.json.org/`
[3] `http://yaml.org/`

necessary data. There are some standardized formats like GML[1], but they're still too complex and dealing with various parts of these formats and recalculating data between different coordinate systems would exceed the scope of this thesis. Therefore, any map data must be converted to supported data scheme.

The internal data scheme structure usable by the application can be easily stored and read to/from all of the mentioned formats (XML, JSON, YAML). For the examples distributed as the part of the thesis, YAML is used in order to be able to easily read the data even in a plain text, without running the application and rendering all the geometries.

8.3 User interface

The "Label Placement" application offers two different interfaces for communication with the user:

- **Graphical user interface (GUI)** is a full featured graphical application, offering all features described above. However, due to its dependence on graphical libraries, it may not be possible to use this client on all devices.
- **Command line interface (CLI)** on the other hand is a very restricted client intended purely for running solvers with already prepared input data. It is independent on any graphical libraries, making it usable on servers, where it can either run internal solvers directly, or prepare input files for external solvers.

Since the whole source code is implemented in Java, the application should be usable on all standard platforms including Windows, Linux and Mac.

[1] `http://www.opengeospatial.org/standards/gml`

Chapter 9
Evaluation and results

After all algorithms have been described, it is time to benchmark them and compare them. This section describes the data sets used for testing, and presents the most significant results.

9.1 Map categories

In order to benchmark the described algorithms, it is necessary to define basic categories of maps that shall be used. Unfortunately, it is extremely difficult to find usable map data (especially in some raw and simply readable way), as most of the companies owning these data are guarding them very carefully (since it is the essential part of their business).

Randomly generated maps

The easiest way to get some map is to generate it randomly (or pseudo-randomly, in order to get some specific characteristics). Random maps were also used in many works cited in this thesis, as they allow for testing the behavior of the algorithms in specific situations. It is easily possible to benchmark the algorithms on maps with various label counts, densities and label sizes.

Unfortunately, there is one obvious disadvantage of automatically generated maps – they usually do not even look like real maps. They are very nice for theoretical performance comparison, but in real maps the algorithms can behave differently.

Realistic maps

Another important class of maps contains real (or real-like) maps. As already mentioned, they are usually not freely available (as they are mostly used commercially), but they serve as a very good benchmark for finding algorithms suitable for the real world problems – and not just for academic comparisons of theoretical performance.

Although it is quite difficult task to find some freely available map sources, it is not impossible. There are multiple open source projects maintaining their own cartographic data and distributing them freely over the internet.

Open Street Map

Probably the most known project focusing on public map data is Open Street Map[1]. Unfortunately, the data is definitely not perfect – the maps look very nice when displayed on a computer screen, but sometimes the data are not really suitable for machine processing (many features are duplicated or encoded in strange ways).

However, many companies still use them, as they are probably the most reliable free source, so it is better for them to clean and preprocess the data to make it usable. Notable advantage is that Open Street Map is often based on local cadastral maps, making the maps quite precise.

[1] `https://www.openstreetmap.org`

Natural Earth

Another interesting map data source is Natural Earth[1]. Volunteers involved in this project maintain various cultural and physical data sets, including maps of countries, populated places, coastlines, rivers, etc.

These data are published in ESRI Shapefile format, which can be imported by most GIS software suites (even by the open source ones like QGIS[2]).

In comparison with Open Street Map, these data are much "nicer and cleaner", which makes their processing easier. However, they are definitely not such complex and they contain only major cartographic features – they miss any low scale data, like small cities or streets.

9.2 Environment

Hardware and software

All algorithms were evaluated on MetaCentrum clusters Zebra[3] and Zefron[4].

Machines in these clusters are running on Intel Xeon E7-4860 (2.27 GHz) processors in case of Zebra cluster, or Intel Xeon E5-4627 v3 (2.6 GHz) processors in case of Zefron cluster. Total available memory on the compute nodes is 256 GB on Zebra nodes and 1 TB on Zefron nodes.

All used machines are running Debian 8.8, Oracle Java JDK 8 and CPLEX 12.6.1.

Algorithm configuration

During the evaluation process, the following settings were configured in the Label Placement application and algorithms:

- **CPU limit**: 1 core
- **Memory limit**: As much as required by the applications, up to 100 GB
- **Metric weights**
  - **Label position penalty**: 10%
  - **Penalty for conflicts with other labels**: 50%
  - **Penalty for conflicts with map objects**: 40%
- **CPLEX**
  - **Time limit**: 1 hour
  - **Memory limit for Branch and cut tree**: 1 GB
- **Genetic algorithm**
  - The whole algorithm was started 10 times for each data set
  - **Population size**: 100
  - **Termination**: After 100 generations
  - **Gene mapping**: Based on hierarchical clustering
  - **Initializaton**: 50% random, 50% random with 2 iterations of local search
  - **Elitism**: 1 individual per generation
  - **Selection mode**: Tournament with 4 candidates
  - **Crossover mode**: Single point crossover with 10% probability
  - **Mutation rate**: 0.4% for each label

[1] http://www.naturalearthdata.com
[2] http://www.qgis.org/en/site/
[3] https://metavo.metacentrum.cz/pbsmon2/resource/zebra.priv.cerit-sc.cz
[4] https://metavo.metacentrum.cz/pbsmon2/resource/zefron.priv.cerit-sc.cz

- **Memetics**: Every 5 generations, 10 randomly chosen individuals are improved by 2 iterations of local search

9.3 Data sets

Algorithm were evaluated on 23 data sets (maps) listed in Table 9.1. They're divided into three basic categories:

- **Random points**: Dense generated maps with three different size levels (small, medium and large) containing only points to be labeled
- **Populated places**: Country maps with populated places and sometimes with regional borders as well. Populated places are labeled and considered as point features.
- **Roads**: Country map with roads that are labeled as line features.

Label coverage in the Table 9.1 is determined as a ratio of the area used by all labels, and the area used covered by a convex hull of all map features. Even though this metric is not exact and precise, it can give an overview on the density of the map. Area of a convex hull of all features is used instead of the area of the whole map, in order to ignore possible huge uncovered areas in the corners of the map.

Map name	Labels	Features	Label coverage
100 Random points - Small labels	100	100	5.65%
100 Random points - Medium labels	100	100	17.67%
100 Random points - Large labels	100	100	42.42%
250 Random points - Small labels	250	250	12.21%
250 Random points - Medium labels	250	250	40.58%
250 Random points - Large labels	250	250	97.04%
500 Random points - Small labels	500	500	23.7%
500 Random points - Medium labels	500	500	78.77%
1000 Random points - Small labels	1000	1000	46.55%
Australia - Populated places	224	317	21.14%
Canada - Populated places	254	664	16.02%
Czech Republic - Populated places	12	27	6.66%
Germany - Populated places	57	117	7.29%
Spain & Portugal - Populated places	66	74	12.41%
Great Britain - Populated places	57	114	10.35%
Sweden - Populated places	34	75	13.56%
Turkey - Populated places	84	90	16.25%
Ukraine - Populated places	55	88	9.19%
Czech Republic - Roads	157	257	6.86%
Spain & Portugal - Roads	334	544	11.78%
Ireland - Roads	57	164	5.65%
Poland - Roads	327	660	13.36%
Sweden - Roads	195	640	6.55%

Table 9.1. List of data sets used for the evaluation.

9.4 Results

This section briefly describes observations and results achieved on the described data set, summarized according to the basic categories which the labels belong to.

9.4.1 Random points

Random points maps contain labels of various sizes. The best results for most of these maps were provided by CPLEX, but this is not a rule – for some larger data sets (with higher percentage of covered map area), the CPLEX solver returned suboptimal solutions, mostly because of time and memory limitations.

Table 9.2 presents the best the fitness values of the solutions found by each evaluated algorithm, on each data set. "GA" denotes the best result provided by the genetic algorithm over all 10 runs, "Greedy 1" denotes the result of the Basic greedy algorithm and "Greedy 2" denotes the result of the Advanced greedy algorithm. Bold values in the table represent the best fitness value for given data set.

Map	GA	Greedy 1	Greedy 2	GRASP	CPLEX
100 small	**0.10**	1.03	0.15	**0.10**	**0.10**
100 medium	**0.69**	3.82	1.06	0.76	**0.69**
100 large	3.66	15.20	6.70	4.28	**3.21**
250 small	1.25	7.89	2.54	1.42	**1.20**
250 medium	8.68	43.35	16.73	10.01	**6.43**
250 large	**55.38**	131.14	74.22	56.72	65.54
500 small	7.61	45.39	15.19	8.15	**6.39**
500 medium	**73.05**	206.71	109.90	77.52	86.91
1000 small	50.21	231.50	90.92	54.25	**42.55**

Table 9.2. Best solutions found for random points maps

Table 9.3 presents the times needed to find a solution using given algorithms. Time listed for the genetic algorithm is a time required to perform a single algorithm run, and hence the time needed to execute all 10 runs was approximately 10 times higher. CPLEX results with times marked by "M" or "T" symbols were terminated before the optimal solution was found, either due to memory (M) or time (T) limits, and therefore the solutions returned in these cases represent the best solutions found before the limits were exceeded.

Map	GA	Greedy 1	Greedy 2	GRASP	CPLEX
100 small	0:40	0:01	0:01	0:02	0:01
100 medium	0:25	0:01	0:01	0:01	0:01
100 large	0:45	0:01	0:01	0:01	0:28
250 small	2:34	0:02	0:02	0:03	0:02
250 medium	6:53	0:04	0:05	0:07	3:18
250 large	4:03	0:02	0:06	0:06	(M) 33:19
500 small	13:46	0:07	0:13	0:15	0:09
500 medium	11:41	0:05	0:17	0:49	(T) 1:00:01
1000 small	1:12:25	0:25	59:00	1:16	(T) 1:00:17

Table 9.3. Solution times for random points maps

For maps with label coverage lower than 50%, CPLEX was the winner. Usually, the results provided by genetic algorithm (over 10 runs) were also very good, followed by the solutions provided by GRASP. Other greedy algorithms are far behind these three mentioned and they're not pictured in any charts, as their results are not very convincing.

Typical evolution progress is obvious from Figure 9.1, which shows the evolution of randomly generated map with 500 medium sized labels, depicted in Figure 9.2.

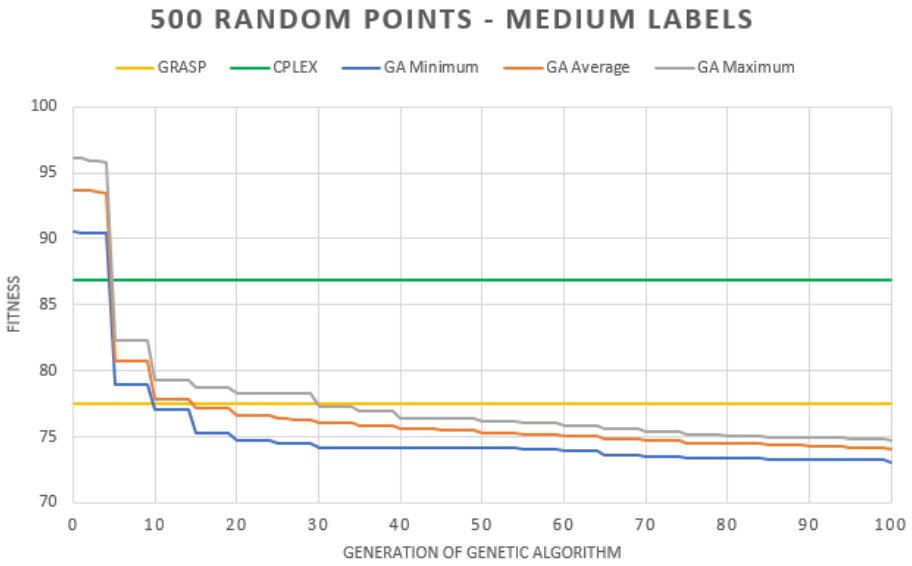


Figure 9.1. Progress on map of 500 random points with medium sized labels

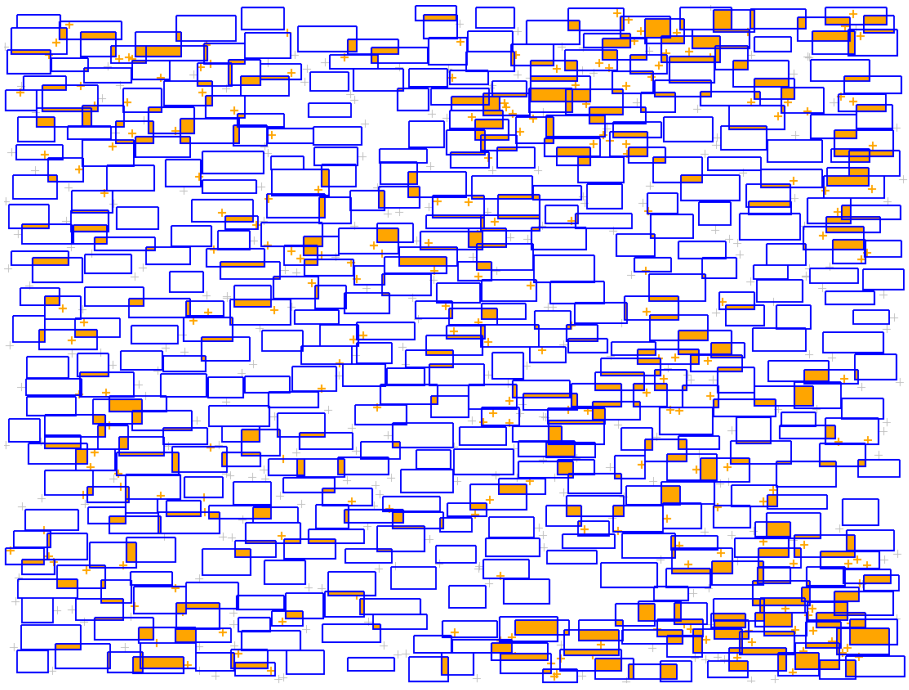


Figure 9.2. Map of 500 random points with medium sized labels

In the evolution progress, there is an obvious influence of the local search during the optimization, as the memetic part triggering the local search is activated in each fifth iteration – exactly at the moments with significant fitness improvements.

9.4.2 Populated places

Similar situation is in the category of populated places, as these maps also contain only labels associated with point features. In contrast to the generated maps, there are also other map features. These are without labels, but they have much more complicated shapes – e.g. in the map of Canada, as pictured in Figure 9.3.

Figure 9.3. Map of populated places in Canada

Fortunately, these complicated geometries do not have to be considered during the label placement itself, as the overlaps between label candidates and map objects do not change during the placement process – and therefore they can be precalculated during the preparation of candidate positions.

Once more, best results were provided by the CPLEX solver – in all cases except the map of Australia, which is visually quite similar to Canada (as most of the populated places in Australia are located near the coastline, and therefore all labels have to fit around it).

Complete results are presented in Tables 9.4 and 9.5, showing the best fitness values found by the algorithm and time required for the optimization. Markup used in these tables is exactly the same as the one in case of random points results.

Map	GA	Greedy 1	Greedy 2	GRASP	CPLEX
Australia	**17.56**	59.62	28.15	19.38	19.37
Canada	15.70	60.34	27.47	16.34	**15.19**
Czech Republic	**0.10**	**0.10**	**0.10**	**0.10**	**0.10**
Germany	**1.22**	2.59	1.53	1.39	**1.22**
Spain & Portugal	**0.92**	3.14	1.42	0.98	**0.92**
Great Britain	1.14	4.21	2.32	1.48	**1.09**
Sweden	0.66	2.93	0.92	0.75	**0.64**
Turkey	1.05	5.58	1.84	1.13	**1.02**
Ukraine	**0.18**	0.66	0.27	0.20	**0.18**

Table 9.4. Best solutions found for populated places maps

Map	GA	Greedy 1	Greedy 2	GRASP	CPLEX
Australia	3:16	0:04	0:05	0:07	(M) 7:35
Canada	5:32	0:10	0:14	0:15	(M) 6:06
Czech Republic	0:01	0:01	0:01	0:01	0:01
Germany	0:17	0:01	0:01	0:01	0:01
Spain & Portugal	0:16	0:01	0:01	0:01	0:01
Great Britain	0:16	0:02	0:02	0:02	0:02
Sweden	0:03	0:01	0:01	0:01	0:01
Turkey	0:35	0:01	0:01	0:02	0:01
Ukraine	0:15	0:02	0:02	0:02	0:02

Table 9.5. Solution times for populated places maps

Very nice results were also provided by GRASP and the genetic algorithm. Due to the low complexity of some populated places maps, the solutions were usually found quickly by most of the algorithms.

There are also two interesting data sets (populated places in the Czech Republic and Ukraine), that were always optimally solved during the initialization phase of the genetic algorithm. These maps are quite easy to process, and hence the local optimizer used for the initialization of the algorithm always solved the problem directly. This happened in all 10 runs of the algorithm, as depicted in Figure 9.4 (the lower line represents CPLEX solution combined with minimum, average and maximum of the genetic algorithm solutions).

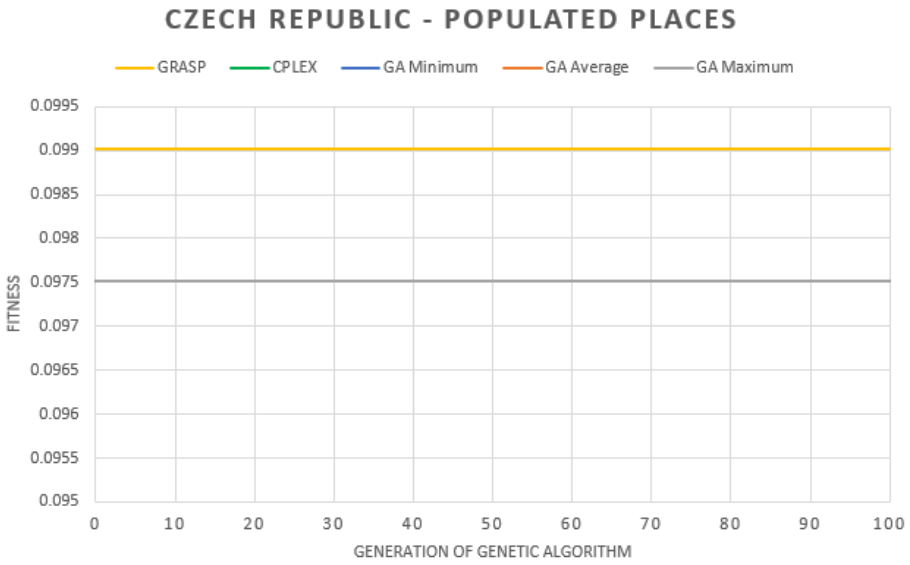


Figure 9.4. Progress on map of populated places in the Czech Republic

9.4.3 Roads

Last category contains maps of the major roads in European countries. These roads are represented as line features, and generally their geometries are polylines (because the roads are usually not straight).

Line feature labels are computationally more expensive than point based ones, as the number of candidate positions is higher – each point feature label had 8 candidate positions, while each line feature label has 32 possible positions. This drastically increases the problem complexity for all placement algorithms.

Complete results are presented in Tables 9.6 and 9.7, showing the best fitness values found by the algorithms and time required for this optimization. Markup used in these tables is exactly the same as the one in case of random points and populated places results.

Map	GA	Greedy 1	Greedy 2	GRASP	CPLEX
Czech Republic	**3.89**	6.02	4.06	3.91	4.02
Spain & Portugal	**19.85**	25.21	22.46	20.11	**19.85**
Ireland	**2.62**	3.22	2.84	2.64	**2.62**
Poland	**28.79**	34.76	31.96	29.01	29.06
Sweden	**17.58**	20.70	18.81	17.73	17.70

Table 9.6. Best solutions found for roads maps

Map	GA	Greedy 1	Greedy 2	GRASP	CPLEX
Czech Republic	7:04	0:34	0:44	0:49	26:00
Spain & Portugal	54:14	4:08	6:03	6:05	(M) 26:56
Ireland	1:39	0:11	0:14	0:14	0:05
Poland	31:49	2:21	3:37	3:57	(M) 33:16
Sweden	9:37	1:13	1:43	1:48	(M) 41:41

Table 9.7. Solution times for roads maps

Surprisingly, very similar results were provided by all three major algorithms – GRASP, CPLEX and genetic algorithm. The difference between solution fitness values were around 1% on all road maps. However, due to the high number of candidate positions, some algorithms needed a lot of time and computational resources in order to find a solution. This applies especially to the genetic algorithm (where a single run on the Spain & Portugal took nearly 1 hour to compute) and for CPLEX, which consumed over 80 GB during the preprocessing phase, and in most cases the optimization was terminated due to the limited amount of memory available for the Branch and cut tree.

Figure 9.5 shows the map of roads in the Czech Republic, as placed by the GRASP algorithm. The placement looks nice, but this does not necessarily apply for all maps.

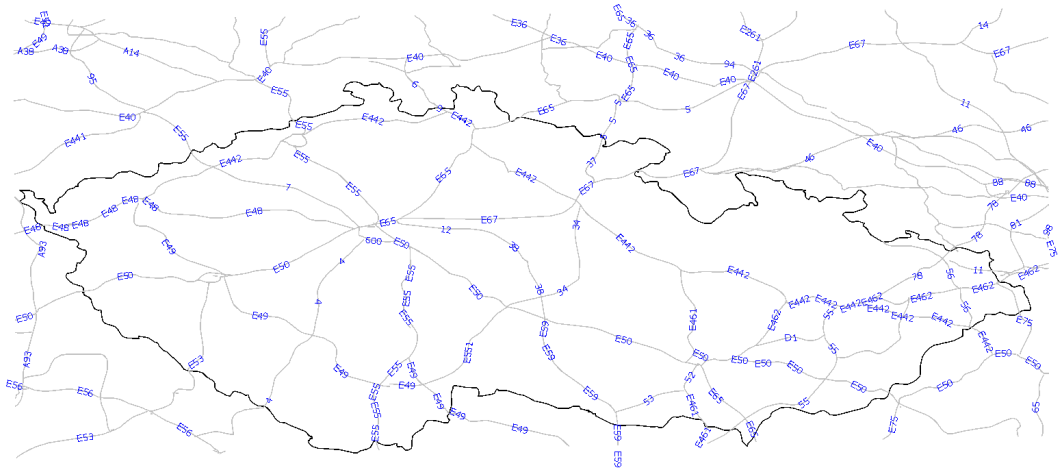


Figure 9.5. Map of roads in the Czech Republic

As a contrast to the previously displayed map, Figure 9.6 shows the map of roads in Poland. This map obviously has too many labels and therefore the map looks messy. However, this is mostly caused by the quality of the original data source, which described each road as a set of multiple geometries.

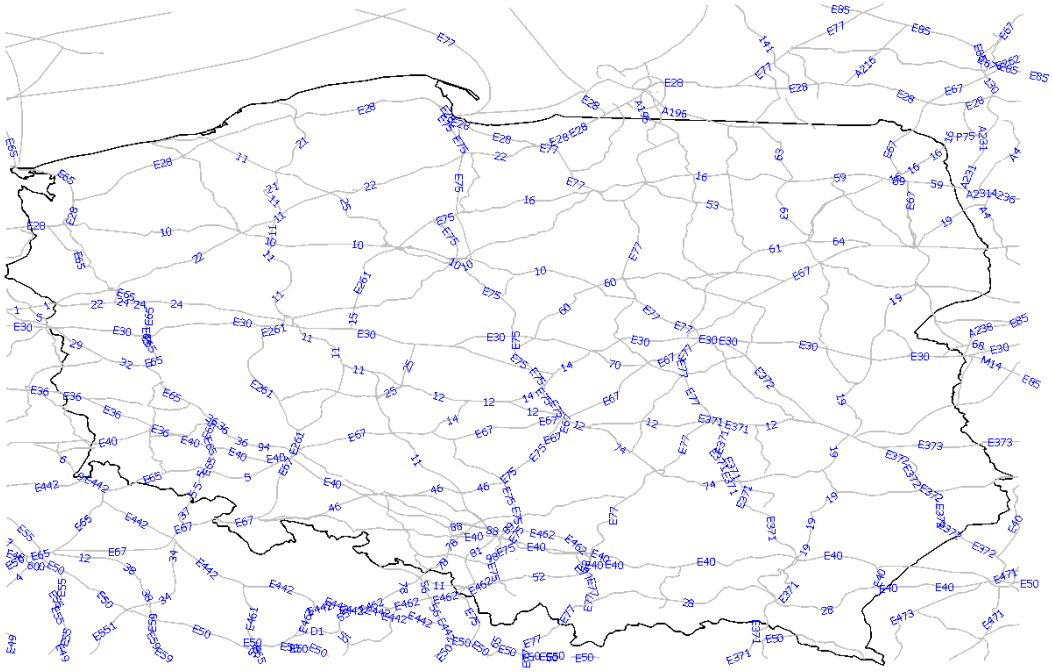


Figure 9.6. Map of roads in Poland

Evolution progress of the described map of roads in Poland is shown in Figure 9.7. It is a nice example of an evolution with a major progress in later iterations, and not only in the beginning.

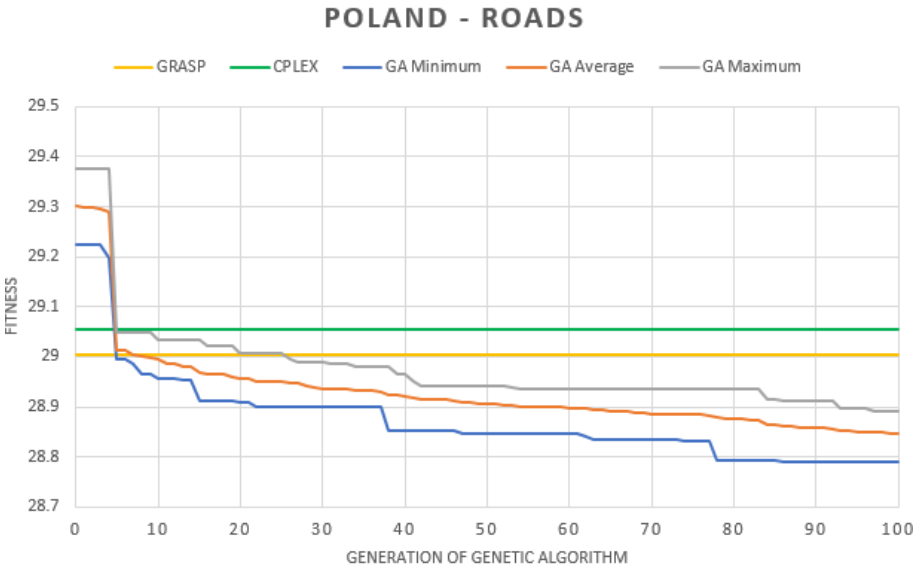


Figure 9.7. Progress on map of roads in Poland

The only suspicious result is the placement of the Czech map provided by CPLEX. The final solution found is worse than some of the other algorithms. However, accord-

ing to the log provided by the CPLEX solver, the algorithm was terminated because it reached a small gap (this topic was described in the chapter about mathematical optimization). In this case, the default gap limit probably caused the premature termination of the optimization process.

9.5 Summary

According to the described benchmark, there are three major algorithms which generally provide very good results – GRASP, CPLEX and the genetic algorithm.

In most cases, GRASP was the fastest algorithm and provided quite nice results. However, in most situations the results provided by other algorithms were slightly better. A big advantage for many users is the determinism of the algorithm, which returns always the same solution (for the same input data).

CPLEX can always provide the best results, but for a very high price – it can either return beautiful solutions after a long processing time, or it can return suboptimal solution after the allowed time or memory limit has been reached. Unfortunately, CPLEX often consumes overwhelming amount of resources, especially memory – sometimes even during the preprocessing phase, which cannot be simply controlled. Like GRASP, the algorithm is deterministic.

Genetic algorithm performed quite well, but it is not deterministic and hence the algorithm had to be restarted multiple times in order to find a nice solution. But still, the genetic algorithm usually outperformed the GRASP algorithm and it was able to find a reasonable solution with much lower computational power than CPLEX.

Remaining greedy algorithms were always outperformed by all other algorithms, but they work really fast and the Advanced greedy algorithm is a good starting point for GRASP.

Chapter 10
Conclusion

Thesis described and compared three different approaches to the map label placement problem. In order to compare the solutions provided by the algorithms, it defined a metric penalizing maps with overlaps between labels and other map elements (map features or other labels), and prioritizing placements having a good position according to standard cartographic rules.

Greedy based algorithms were able to provide some results very quickly, but the solution quality was not exactly the best. However, when the greedy algorithm was improved by adding a local optimization step (together called "GRASP"), the performance got much better and preserved the speed of the greedy approach. The algorithm is deterministic, which is an important advantage for many potential users.

Mathematical optimization (represented by the 0-1 bilinear programming, and solved using the IBM ILOG CPLEX toolbox) showed, that exact methods can find the optimal results. Unfortunately, with the growing problem size, the complexity becomes intractable and the optimization process consumes huge amount of time and computational power. However, for smaller problems the mathematical optimization might be a good way to go. For larger problems, it is still an option, but it is necessary to properly limit resources (especially time) allocated to the solver.

Because of the time and computational power demands of the mathematical optimization solvers, it is often necessary to use external servers or workstations. Such hardware could be very costly, and considering the price of the major mathematical programming solvers, the financial aspect could be also important for deciding whether this solution shall be used or not.

Genetic algorithm performed quite well, especially because of the memetic part. The evolutionary approach itself has just a limited capability of improving the solution, but the integration of local search into the placement process brings the ability to evolve selected solutions faster, speeding up the whole optimization process. However, due to non-determinism of the algorithm, it may be necessary to restart the evolution multiple times to find satisfying results.

Unlike many other similar works, this thesis evaluated the presented algorithms on real cartographic data, instead of only random and fictional data sources. Interesting improvement presented in the thesis is the combination of the local search techniques together with advanced greedy algorithm, which has excellent time-performance ratio. Another interesting algorithm is the genetic algorithm, combined with a local search and hierarchical clustering, allowing meaningful crossovers during the evolution.

References

[1] Anthony C. Cook and Christopher B. Jones. A Prolog Rule-Based System for cartographic Name Placement. *Computer Graphics Forum*, 9(2):109–126, 1990.

[2] Jan-Menno Kraak. Settings and needs for web cartography. In Jan-Menno Kraak and Allan Brown, editors, *Web Cartography*. Taylor & Francis, London, 2001.

[3] Joe Marks and Stuart Shieber. The Computational Complexity of Cartographic Label Placement. Technical Report TR-05-91, Harvard University, 1991.

[4] Aleš Kobr. Automatic map label placement. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2013. Available only in Czech.

[5] Alexander Wolff and Tycho Strijk. The Map-Labeling Bibliography. `http://i11www.ira.uka.de/map-labeling/bibliography`, 1996. Accessed: 2016-09-30.

[6] Eduard Imhof. Die Anordnung der Namen in der Karte. *International Yearbook of Cartography*, pages 93–129, 1962.

[7] Eduard Imhof. Positioning Names on Maps. *The American Cartographer*, 2(2):128–144, 1975.

[8] Pinhas Yoeli. The Logic of Automated Map Lettering. *The Cartographic Journal*, 9(2):99–108, 1972.

[9] 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.

[10] Karolína Burešová. Placement of map symbols. Bachelor's thesis, Charles University, Faculty of Mathematics and Physics, Prague, 2015. Available only in Czech.

[11] Alexander Wolff. *Automated Label Placement in Theory and Practice*. PhD thesis, Free University of Berlin, 1999.

[12] Robert G. Cromley. An LP Relaxation Procedure for Annotating Point Features Using Interactive Graphics. In *Proceedings of the Seventh Auto-Carto Conference*, pages 127–132, 1985.

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

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

[15] Donald Meagher. Geometric Modeling Using Octree-Encoding. *Computer Graphics and Image Processing*, 19(2):129–147, 1982.

[16] Steven van Dijk. *Genetic Algorithms for Map Labeling*. PhD thesis, Utrecht University, 2001.

[17] Oleg V. Verner, Roger L. Wainwright, and Dale A. Schoenefeld. Placing Text Labels on Maps and Diagrams using Genetic Algorithms with Masking. *INFORMS Journal on Computing* , 9(3):266–275, 1997.

[18] Günther R. Raidl. A Genetic Algorithm for Labeling Point Features. In *Proceedings of the International Conference on Imaging Science, Systems and Technology*, pages 189–196, 1998.

[19] Lucas Bradstreet, Luigi Barone, and Lyndon While. Map-labelling with a Multi-objective Evolutionary Algorithm. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, pages 1937–1944, 2005.

[20] Michael Formann and Frank Wagner. A Packing Problem with Applications to Lettering of Maps. In *Proceedings of the Seventh Annual Symposium on Computational Geometry*, SCG '91, 1991.

[21] Chung Keung Poon, Binhai Zhu, and Francis Chin. A polynomial time solution for labeling a rectilinear map. *Information Processing Letters*, 65(4):201–207, 1998.

[22] Gildásio Lecchi Cravo, Glaydston Mattos Ribeiro, and Luiz Antonio Nogueira Lorena. A greedy randomized adaptive search procedure for the point-feature cartographic label placement. *Computers & Geosciences*, 34(4):373–386, 2008.

[23] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, third edition, 2009.

[24] Mauricio G. C. Resende and Celso C. Ribeiro. Greedy Randomized Adaptive Search Procedures. In Fred Glover and Gary A. Kochenberger, editors, *Handbook of Metaheuristics*, pages 219–249. Springe US, Boston, MA, 2003.

[25] Artyom G. Nahapetyan. Bilinear programming. In Christodoulos A. Floudas and Panos M. Pardalos, editors, *Encyclopedia of Optimization*, pages 279–282. Springer US, Boston, MA, 2009.

[26] Susanne Heipcke. Comparing Constraint Programming and Mathematical Programming Approaches to Discrete Optimisation – The Change Problem. *Journal of the Operational Research Society*, 50(6):581–595, 1999.

[27] Manfred Padberg and Giovanni Rinaldi. A Branch-and-Cut Algorithm for the Resolution of Large-scale Symmetric Traveling Salesman Problems. *SIAM Review*, 33(1):60–100, 1991.

[28] Melanie Mitchell. *An Introduction to Genetic Algorithms*. The MIT Press, Cambridge, MA, 1996.

[29] Lior Rokach and Oded Maimon. Clustering Methods. In *Data Mining and Knowledge Discovery Handbook*, pages 321–352. Springer US, Boston, MA, 2005.

[30] Pablo Moscato. On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts - Towards Memetic Algorithms. Technical Report 826, California Institute of Technology, Pasadena, CA, 1989.

[31] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice*. Addison-Wesley, Reading, MA, second edition, 1990.

[32] Ivan E. Sutherland and Gary W. Hodgman. Reentrant Polygon Clipping. *Communications of the ACM*, 17(1):32–42, 1974.

[33] Kevin Weiler and Peter Atherton. Hidden Surface Removal Using Polygon Area Sorting. *ACM SIGGRAPH Computer Graphics*, 11(2):214–222, 1977.

Appendix A
Computational geometry algorithms

In order to properly penalize overlaps between pairs of labels, or conflicts between labels and map objects (which are important to score the label positions and even the whole solutions), there must be a way to properly work with various geometries and to be able to find their intersections.

Algorithms described in this part belong to the category of computational geometry, and allow to find intersections of lines or polygons with another polygon. Fortunately, all problems solved in order to find a solution to the label placement problem, have to be solved just in two-dimensional space.

Also, all polygons representing labels geometries could be considered as convex ones. Of course, there could be some strange and non-convex shaped label in the problem as well, but replacing these non-convex polygons by their convex hulls should not do much harm. Such relaxation could rarely change possible label positions, but this would be very uncommon, while having all polygons convex can heavily simplify many operations like line clipping or polygon clipping.

A.1 Convex hull

"Convex hull of a set of points is a smallest convex polygon for which each of the points in the given set is either on the boundary of the convex polygon or in its interior." [23]

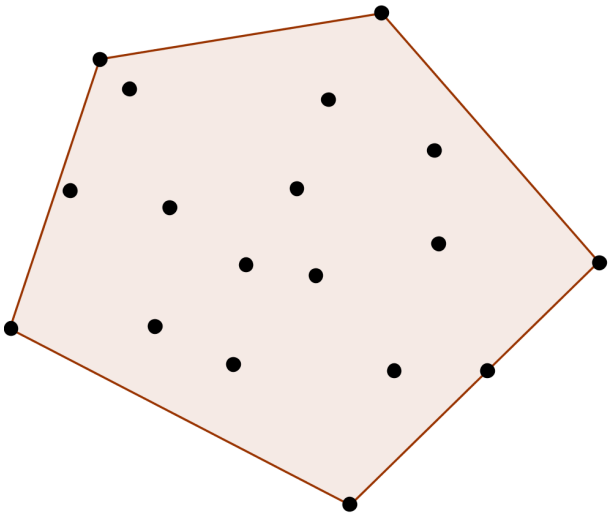


Figure A.1. Convex hull of a point set[1]

Less formally said, convex hull is the smallest convex polygon that covers all of the given points. Convex polygon (also not so formally) is such polygon, where any point

[1] `https://hcmop.wordpress.com/tag/combinatorial-geometry`

between the two points belonging to the polygon is also inside or on the border of the polygon.

Convex hulls are interesting for the geometry tasks that are to be subsequently executed during the preparation of the label position candidates and execution of the algorithms themselves, as it is much easier to calculate intersections of convex polygons than just generic ones without the convexity property.

There are many algorithms designed for the calculation of convex hulls, and the most known ones are the following two:

- **Graham's scan algorithm** is an algorithm with $n \log n$ complexity (where n is number of points in the set). First, the points are sorted by polar angle, related to some arbitrary point (usually the bottom-left one). Then, all of the points are processed one by one and added to the convex hull. In case when adding some point shows up that the previously added point breaks the convexity property, then the previously added points are removed until the hull is convex again.
- **Jarvis's march algorithm** (gift wrapping) is alternative algorithm with nh complexity (where n is number of points in the set and h is number of points in the hull). In each iteration of the algorithm, one new edge to the hull is added (by adding one point) so that all of the remaining points on the set are on the right side of line represented by that specific edge. This way, when the hull is closed, all points must necessarily be inside the set and the hull is guaranteed to be convex.

In the implementational part of this thesis, there is the Graham's scan algorithm used to calculate convex hulls of label geometries.

A.2 Line clipping

In order to calculate intersections of labels and various linear map objects (like lines or polylines), it is necessary to have some line clipping algorithm. There are multiple commonly used algorithms for line clipping, and some or the most known ones are [31]:

- **Cohen-Sutherland algorithm** is only intended for clipping lines by rectangles. The algorithm divides the area around the rectangle to eight areas (top, left, top-left, etc.), as these can be simply determined based on coordinates of the clipping rectangle – and with the same ease, any point can be classified either to be inside the rectangle or to belong to one of these eight areas. And based on specific rules, it quickly determines whether the processed line lies inside the rectangle, outside the rectangle or if it somewhere intersects the border of the rectangle. If so, the line is clipped by the intersecting part of the rectangle border and the process is repeated until the line is clipped completely.
- **Cyrus-Beck algorithm** is slightly more advanced algorithm, which can clip lines by any complex polygon (hence, it is more universal method than Cohen-Sutherland). Internally, the algorithm uses parametric equation of the given line and calculates conflicts with the edges of the clipping polygon in order to clip the line appropriately. Generally, this algorithm tends to be more effective than the Cohen-Sutherland (which, on the other hand, has the ability to quickly filter out lines that have no intersection with the clipping rectangle at all).

These algorithms are intended for clipping lines by rectangles or convex polygons, and this operation can always produce only a single line, single point or no intersection geometry at all. If there would be any need for intersecting lines with non-convex polygons, the output could possibly be a set of lines.

In case of intersection of polyline and any polygon (even convex one or a rectangle), the result of the operation could be a set of lines as the polyline could freely enter and leave the clipping object multiple times. Hence, it is necessary to analyze each line in the polyline (or polygon border – which is in fact also a polyline) separately.

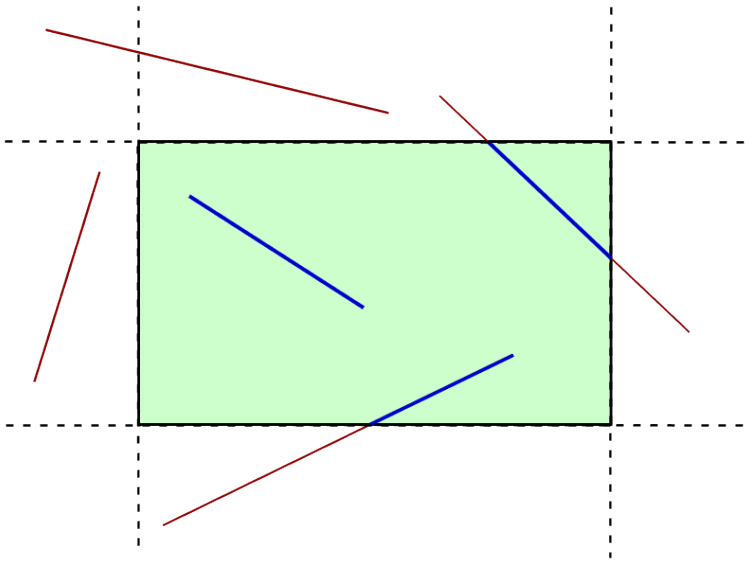


Figure A.2. Line clipping using Cohen-Sutherland algorithm[1]

In the implementation part of this thesis, Cohen-Sutherland is used to compute intersections of lines and rectangles, while an algorithm inspired by the Cyrus-Beck is used to find intersections of lines and convex polygons. Due to convexity, it is not even necessary to compute all intersections with edges of the polygon, but only the first two (as the line can enter the polygon no more than once, and in a similar way it can leave the polygon no more than once).

A.3 Polygon clipping

For calculating label overlaps, it is essential to be able to clip a polygon by another polygon. Again, there are multiple algorithms designed to solve this problem, and the two most notable follows:

- **Sutherland–Hodgman algorithm** [32] is primarily intended for clipping a "subject" polygon by a convex "clipping" polygon, resulting a new polygon if the original two had some intersection. If both of the input polygons are convex, the outcome is guaranteed to be a single polygon. The algorithm processes all edges from the clipping polygon, and finds all its intersections with the subject polygon by edges. If some intersections are found, the subject polygon is cropped accordingly and only the updated polygon is considered in the subsequent iterations. After all edges have been processed, updated version of the subject polygon is the intersection of both input polygons.
- **Weiler–Atherton algorithm** [33] is a more complicated algorithm, which can also clip any simple polygon (i.e. polygon where any two edges do not intersect each other) by another simple polygon, without relying on convexity. It is also possible to extend

[1] `https://en.wikipedia.org/wiki/Line_clipping`

this algorithm to support more complex polygons containing holes. Result of this algorithm might consist of multiple different polygons.

Since all label geometries in this thesis are already restricted to be convex polygons (or they're converted to convex polygons using an algorithm for finding convex hulls, as mentioned before), there is no need for the complexity of Weiler–Atherton algorithm and hence the implementationally simpler and faster Sutherland–Hodgman algorithm could be used.

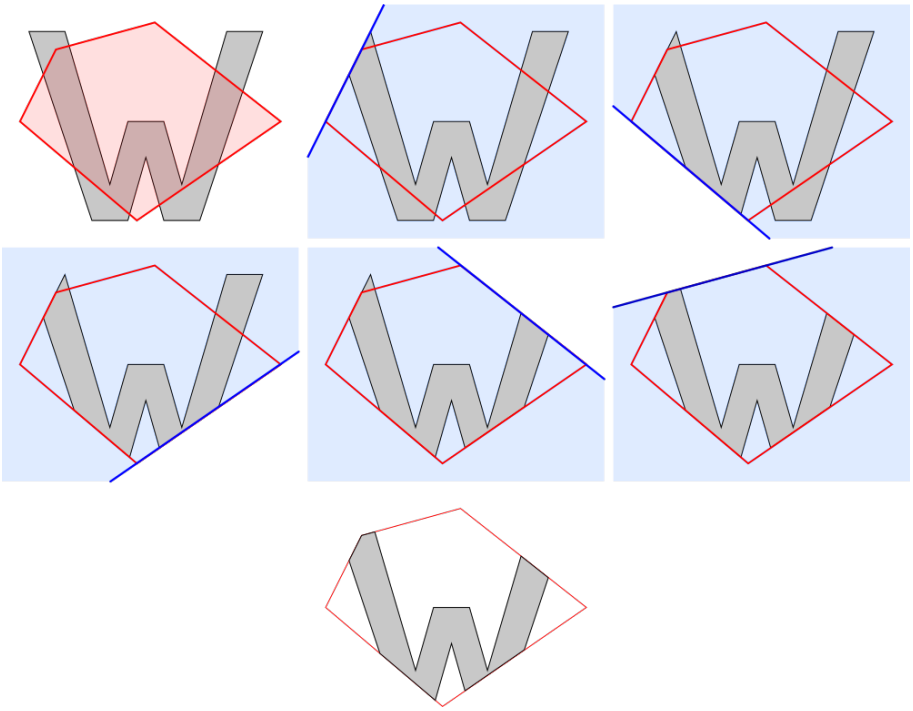


Figure A.3. Polygon clipping using Sutherland–Hodgman algorithm[1]

[1] https://en.wikipedia.org/wiki/Sutherland-Hodgman_algorithm

Appendix B
Contents of the attached CD

Important part of this thesis is a compact disc containing the following files:

data/	map data used for the evaluation
sources/	Java implementation of the Label Placement UI and algorithms
tex/	TeX sources of this thesis
thesis.pdf	this thesis in PDF

Table B.1. Contents of attached CD

Appendix C
List of abbreviations

Following abbreviations were used in this thesis:

API	Application Programming Interface
CAD	Computer Aided Design
CLI	Command Line Interface
CSP	Constraint Satisfaction Problem
GA	Genetic Algorithm
GIS	Geographic Information System
GLPK	GNU Linear Programming Kit
GML	Geography Markup Language
GRASP	Greedy Randomized Adaptive Search Procedure
GUI	Graphical User Interface
ILP	Integer Linear Programming
JSON	JavaScript Object Notation
LP	Linear Programming
MILP	Mixed Integer Linear Programming
OPL	Optimization Programming Language
PDF	Portable Document Format
SAT	Satisfiability (Boolean Satisfiability Problem)
XML	eXtensible Markup Language
YAML	Yet Another Markup Language / YAML Ain't Markup Language