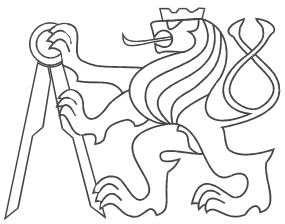
Czech Technical University in Prague

Faculty of Electrical Engineering

Department of Cybernetics



Bachelor's thesis

Metro-Line Crossing Minimization Problem on Czech Hiking Trails

May 23, 2013

Petr Ryšavý

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Petr R y š a v ý

Studijní program: Otevřená informatika (bakalářský)

Obor: Informatika a počítačové vědy

Název tématu: Metro-Line Crossing Minimization Problem na turistických trasách KČT

Pokyny pro vypracování:

1. Zpracujte přehled současné literatury k Metro-line Crossing Minimization Problem.
2. Vyberte či navrhněte vhodný algoritmus pro použití na české turistické trasy.
3. Přístup ověřte na datech dostupných z projektu OpenStreetMap.

Seznam odborné literatury:

[1] Nöllenburg, M.; Wolff, A.: Drawing and Labeling High-Quality Metro Maps by Mixed-Integer Programming. IEEE Transactions on Visualization and Computer Graphics, vol. 17, no. 5, May 2011.

[2] Nöllenburg, M.: An Improved Algorithm for the Metro-Line Crossing Minimization Problem. GD'09 Proceedings of the 17th International Conference on Graph Drawing, pp. 381-392.

[3] Argyriou, E. N.; Bekos, M. A.; Kaufmann, M.; Symvonis, A.: On Metro-Line Crossing Minimization. Journal of Graph Algorithms and Applications vol. 14, no. 1, pp. 75-96 (2010).

[4] Argyriou, E. N.; Bekos, M. A.; Kaufmann, M.; Symvonis, A.: Two Polynomial Time Algorithms for the Metro-Line Crossing Minimization Problem. In: Tollis, I. G. and Patrignani, M. (Eds.): GD 2008, LNCS 5417, pp. 336–347, Springer, Heidelberg, 2009.

[5] Asquith, M.; Gudmundsson, J.; Merrick, D.: An ILP for the metro-line crossing problem. In: Harland, J.; Manyem, P. (eds.) Proc. 14th Computing: The Australasian Theory Symposium (CATS2008), Wollongong, NSW, Australia. Conferences in Research and Practice in Information Technology (CRPIT), vol. 77, pp. 49 – 56. Australian Comput. Soc. 2008.

[6] Wolff, A.: Drawing SubwayMaps: A Survey – Informatik Forsch. Entw. (2007) 22: 23–44

Vedoucí bakalářské práce: Radomír Černoch, MSc.

Platnost zadání: do konce zimního semestru 2013/2014

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE Fakulta elektrotechnická -7-

L.S.

prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 10. 1. 2013

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Petr R y š a v ý

Study programme: Open Informatics

Specialisation: Computer and Information Science

Title of Bachelor Project: Metro-Line Crossing Minimization Problem on Czech Hiking Trails

Guidelines:

1. Review the recent literature on the Metro-line Crossing Minimization Problem.
2. Choose or develop a suitable algorithm for optimizing the layout of Czech hiking trails.
3. Evaluate your design on the data provided by the OpenStreetMap project.

Bibliography/Sources:

[1] Nöllenburg, M.; Wolff, A.: Drawing and Labeling High-Quality Metro Maps by Mixed-Integer Programming. IEEE Transactions on Visualization and Computer Graphics, vol. 17, no. 5, May 2011.

[2] Nöllenburg, M.: An Improved Algorithm for the Metro-Line Crossing Minimization Problem. GD'09 Proceedings of the 17th International Conference on Graph Drawing, pp. 381-392.

[3] Argyriou, E. N.; Bekos, M. A.; Kaufmann, M.; Symvonis, A.: On Metro-Line Crossing Minimization. Journal of Graph Algorithms and Applications vol. 14, no. 1, pp. 75-96 (2010).

[4] Argyriou, E. N.; Bekos, M. A.; Kaufmann, M.; Symvonis, A.: Two Polynomial Time Algorithms for the Metro-Line Crossing Minimization Problem. In: Tollis, I. G. and Patrignani, M. (Eds.): GD 2008, LNCS 5417, pp. 336–347, Springer, Heidelberg, 2009.

[5] Asquith, M.; Gudmundsson, J.; Merrick, D.: An ILP for the metro-line crossing problem. In: Harland, J.; Manyem, P. (eds.) Proc. 14th Computing: The Australasian Theory Symposium (CATS2008), Wollongong, NSW, Australia. Conferences in Research and Practice in Information Technology (CRPIT), vol. 77, pp. 49 – 56. Australian Comput. Soc. 2008.

[6] Wolff, A.: Drawing SubwayMaps: A Survey – Informatik Forsch. Entw. (2007) 22: 23–44

Bachelor Project Supervisor: Radomír Černoch, MSc.

Valid until: the end of the winter semester of academic year 2013/2014

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE Fakulta elektrotechnická -7- LS

prof. Ing. Vladimír Mařík, DrSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 10, 2013

Poděkování

Rád bych poděkoval vedoucímu bakalářské práce Radomíru Černochovi, MSc. za ochotu a mnoho užitečných rad nad rámec běžných povinností, které mi pomohly se řešením problému a psaním této práce. Dík patří i mým rodičům, kteří mi umožňují studovat to, co mě zajímá.

Prohlášení

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 23. května 2013

Petr Ryšavý

Acknowledgement

I would like to thank the project supervisor Radomír Černoch, MCs. for his helpfulness and for many useful suggestions beyond the normal duties that helped me to solve the problem and write this thesis. Thanks also goes to my parents, who allow me to study what I am interested in.

Declaration

I 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 academic thesis.

In Prague at May 23, 2013 Petr Ryšavý

Abstrakt

Účelem této práce je popsat Metro-Line Crossing Minimization Problem a vyvinout algoritmus pro řešení obecné varianty tohoto problému. Máme-li rovinný graf $G = (V, E)$ a množinu linek L v tomto grafu, pak je naším cílem naleznout nakreslení těchto linek ke hranám grafu G takové, že se linky kříží co nejméně krát. Graf G může odpovídat kolejím nebo cestám a linky z L jsou pak tramvajové linky nebo turistické značky.

První část práce ukazuje motivaci, proč je vhodné se zajímat o tento problém. Další kapitoly představí notaci a terminologii spojenou s tímto problémem a uvedou formální definice. Vyvineme algoritmus založený na technologii CSP, který řeší nejobecnější variantu Metro-Line Crossing Minimization Problem. Tento algoritmus otestujeme na datech získaných z OpenStreetMap. Také zmíníme ostatní známé algoritmy pro řešení tohoto problému. Tyto algoritmy popíšeme pouze krátce a zaměříme se hlavně na jejich porovnání. Práce také zmiňuje příbuzné problémy, zvláště pak *Metro-map Layout Problem*.

Od čtenáře se očekává základní znalost teorie grafů.

Klíčová slova: MLCMP, minimalizace křížení, vykreslování grafů, CSP, turistické trasy, mapy s linkami metra

Abstract

The aim of this work is to describe the *Metro-Line Crossing Minimization Problem* and develop an algorithm for the general form of this problem. Given a plane graph $G = (V, E)$ and a set of lines L on this graph we want to draw the lines from L along the edges of G such that they cross each other as few times as possible. The graph G may represent some rails or tracks and the lines from L may be some tram lines of tourist trails.

The first part of the work motivates the problem. Next chapters set up notation and terminology connected with this problem and will introduce formal definitions. We will develop a CSP algorithm that solves the most general form of the Metro-Line Crossing Minimization Problem. This algorithm will be tested on data provided by the OpenStreetMap. Further we will describe known algorithms solving this problem. We will touch only a few details of these algorithms and our intention will focus on their comparison. Furthermore the thesis briefly discusses some related problems, especially the *Metro-map Layout Problem*.

As for prerequisites, the reader is expected to be familiar with basic graph theory.

Keywords: MLCMP, crossing minimization, graph drawing, CSP, tourist trails, metro maps

Contents

Chapter 1 Introduction

This document deals with the the Metro-Line Crossing Minimization Problem. This particular problem is one of subjects of interest of graph drawing. The Metro-Line Crossing Minimization Problem (or shortly MLCMP) often arises when drawing some maps with lines along some path, like tourist trails or public transport maps. One of the things that make the map more comfortable to read is the minimization of crossings between the trails or transportation lines.

This chapter is intended as an informal attempt to motivate this problem. It shows some practical applications, where the problem arises. We will briefly introduce OpenStreetMap as the source of data needed for testing the program.

1.1 Metro map drawing history

For thousands of years people needed maps for their orientation. First maps were only simple drawings on cave walls that showed nearest neighbourhood. Such drawings can be found in *Pavlov* cave in Czech Republic as well as at many other places all around the world. [21] Over time these diagrams developed into first maps that presented area of countries or even whole continents.

For a long time there were only handmade maps that showed mere details of the area. In recent decades with the advent of modern technologies maps can be more precious and contain more and more data. But greater amount of data in a map decreases its readability, so that there is a need of better data visualization. The first attempts developed into specialized maps that show only a specific area of interest.

One of these particular map types is the *metro map*. It is a map that shows public transportation lines as metro lines or tram lines. The scale of such maps is not strictly given, but usually is smaller around the city center and bigger at the periphery. Also the directions of rails are not strictly preserved, but they are only approximate as the rails are drawn under some specific conditions. These conditions will be precisely described in section 3.3.

Now it is time to mention British graphic designer *Henry Beck*, who lived from year 1902 to 1974. In the year 1931 he draw first public transportation map of London underground that is similar to modern maps. His map was published two years later in 1933 and his concept became very successful and other transportation maps have adopted rules he used for drawing the map. [20]

Maps prior to Beck showed lines drawn in strict geographical layout, but Beck's diagrams broke this layout. He placed stations in such way that those in city center were at similar distance from each other as those at the periphery. As a result his diagrams were easier to read than exact maps.

Lately Henry Beck also produced two diagrams of Paris metro as well as many other maps of London underground as the network was changing. Metro maps, as we know

them today, are based on Beck's ideas. The similarity is obvious at first glance. Beck's map similarly as today maps use colors to distinguish between lines. Moreover lines are drawn only under eight cardinal directions. [20]

1.2 OpenStreetMap

In this section we will closely look at the *OpenStreetMap*. The OpenStreetMap collects and distributes free geographical data. The project is inspired by similar community projects like Wikipedia. The users collect the data and then modify the map so that all other users can see them.[17]

OpenStreetMap project was started by *Steve Coast* in the year 2004. Initially the main focus was placed on mapping the United Kingdom, but now the map contains data from all around the world. On 6th January 2013 the OpenStreetMap reached one million users. [18]

Data from OpenStreetMap can be downloaded by anyone under the *Open Data Commons Open Database License*. Everyone can export the data into a picture, embed the map into his own website or download data in XML format for other usage. The data can be used for example in GPS devices. [16]

OpenStreetMap map can be found at `http://www.openstreetmap.org/`. The data download page is `http://planet.openstreetmap.org/`. There are many projects that use the OpenStreetMap data for their own purposes. We can mention 3D map of Czech Republic `http://osm.kyblsoft.cz/3dmapa/` or Czech tourist map at `http://mtbmap.cz/`. There are also many commercial projects that use OpenStreetMap data. One example of them is the Foursquare project. [7]

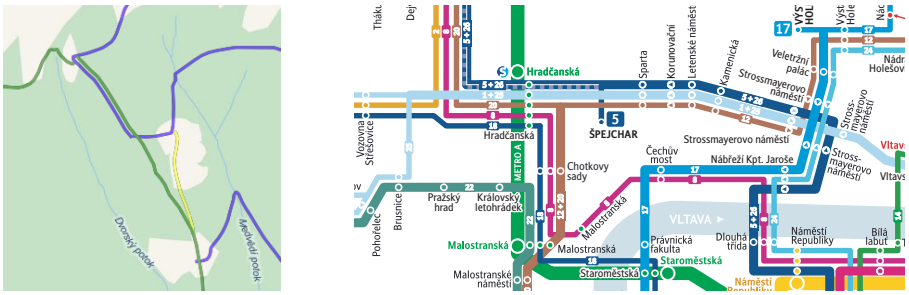
1.3 Crossings in map

In the map, there are very often some trails, tracks, ways or lines that tell us were to go. Often two or more of them share a common physical path, so you cannot tell which of them lays more to the north or west. Examples of such lines in maps can be tourist trails.[1] They are following exactly same track, but in the map they are shown next to each other. Another example of such lines in maps are public transportation lines like underground or tram lines. They use same rails, but in maps they are drawn using multiple lines.

We can define many criteria to draw parallel lines onto a map, but the main point of them should be the improvement of map readability. The choice of the lines order is arbitrary. A map where the number of lines crossing is minimized is supposed to be better readable than a map which contains too many redundant crossings. This fact is widely seen in public transportation maps. If you look on a map of a city you do not know, you have to follow the lines from one place to another in order to find the way.

What may be surprising is the fact that many of official tourist maps do not respect this fact and they introduce many crossings that can be easily removed. The figure 1.1a shows redundant crossing of green and blue trail near Martinova bouda in Giant mountains in Czech Republic. The green trail can be drawn on the left side of the blue one and two crossings will not be in the map anymore. Exactly the same useless crossing

[1] Depends on that in which country you currently are. For example in Austria or Switzerland only one color is mainly used for guiding you through mountains. But in Central European Region there are many colors that are used to distinguish between the trails. Czech trails system uses four colors - red, blue, green and yellow. Each of them has special meaning, red trails are the most important ones and are very long, and on the opposite the yellows are often short and only connect trails with other colors. Moreover they often share the same track and then they split at some point.



(a) From OpenStreetMap [15] (b) From Prague transportation map [6]

Figure 1.1: Examples of unnecessary crossings in the maps

can be found in Czech online tourist map at `http://mapy.cz/#x=15.585876&y=50.771671&z=14&l=16`, as well as in many paper maps.

The most of official transportation maps minimize the number of crossings. But sometimes a redundant crossing can be found in such a map. The figure 1.1b shows a clip from the official Prague transportation map. If the line 20/12 was drawn above the line 1+25, there would be only one crossing instead of two in the map. But anyway it can be seen that this map has introduced very few crossings that can be omitted.

Now we can think about crossings in a map on a larger scale. The number of crossings can be minimized not only locally but in the whole map. The whole work deals with the question, how to draw the map with lines or tourist trails on it such that they cross each other as few times as possible. We want to find the drawing of map, with no redundant crossings.

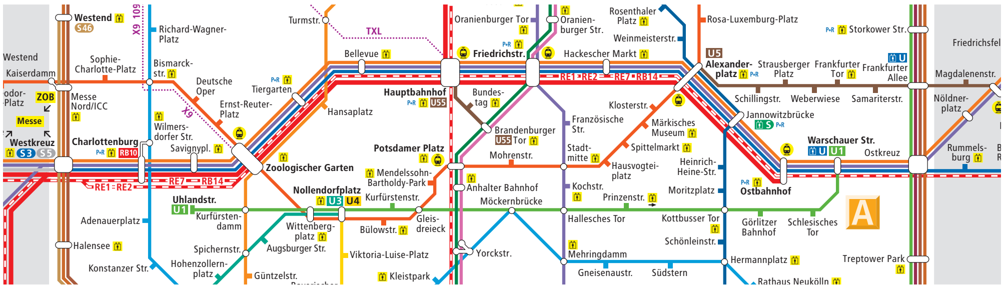
The Metro-Line Crossing Minimization Problem deals with minimization of number of crossings in a map. There are several variants of the problem. One area where they differ is how the lines look like. They can be paths in the map as in the most of public transportation maps, but they can be also trees or moreover they can contain cycles. The second place for differences is the graph of rails or tracks. Many metro maps contain a restriction that the line can enter only on the left side and exit on the right side of the station. In other maps lines can enter and exit in four or eight directions.

We will illustrate different types of restrictions on the map using the figure 1.2 with clips of the official Berlin 1.2a and Sydney 1.2b transportation maps. The green U1 in Berlin map 1.2a or violet Inner West Line in Sydney map 1.2b are good examples of lines, that ore only paths. On the other site orange Bankstown Line in Sydney map 1.2b is a tree. In the Berlin map 1.2a there are many stations that lines enter or exit in four directions. Such examples are Westkreuz, Zoologischer Garten, Hauptbahnhof or Ostkreuz. On the opposite stations like Tiergarten are just two-side. This means that lines enter on one side and exit on the other. In the clip of Sydney CityRail 1.2b, there are no stations which lines enter in four directions.

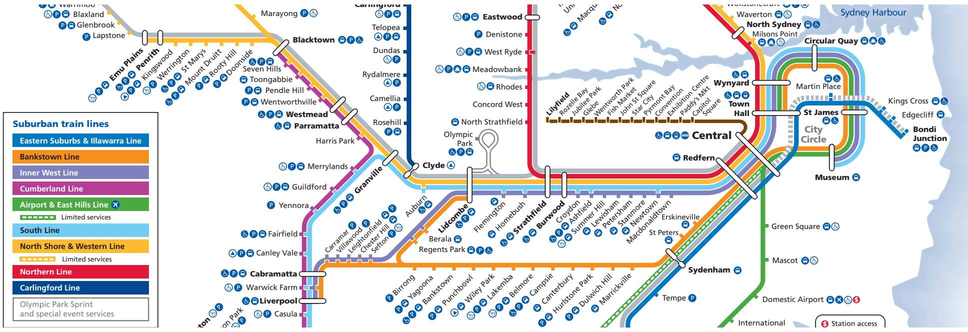
Notice that in both maps each edge is drawn in one of the eight cardinal directions. This is one of the usual constraint that are imposed on a good metro map.

In the Metro-Line Crossing Minimization Problem, the positions of lines, trails, stations and so on in the plane are given. Our task is only to find the ordering of the lines on each edge. The underlying graph can be given by exact geographical layout (as for the tourist trails example) or by previous work of some designer or another program.

On the other site the Metro-map Layout Problem has the whole drawing as the output, especially the positions of all stations in the map. This positioning is given by the geographical layout, but not strictly. In the center of big city stations are very close, on the contrary in the periphery they can be far away. Metro map that would use strict geographical layout will be very unclear in the area of the city center. Thus the scale in



(a) Clip from Berlin S-bahn map [11]



(b) Clip from Sydney CityRail map [19]

Figure 1.2: Different public transportation maps examples

the area of center is smaller than in the periphery. Our task is then to find a nice drawing that does not strictly correspond to the geographical layout, but enables orientation in the map. In section 3.3 we will look closely on Metro-map Layout Problem.

The Metro-Line Crossing Minimization Problem and the Metro-map Layout Problem could be used for solving another problems after slight modifications. They could be used for drawing some diagrams or schemes as well as placing wires on circuit boards.

Chapter 2 Model

In this chapter we set up the notation and terminology connected with the Metro-Line Crossing Minimization Problem. We will define the problem of minimization of crossings of lines. In order to do that we will need to formally define the ordering of lines on edges and crossing of lines.

2.1 Problem definiton

The Metro-Line Crossing Minimization Problem has to be given an undirected graph $G = (V, E)$ with planar embedding $v \mapsto \mathbb{R}^2$ (where $v \in V$ is arbitrary vertex of G). This graph represents the physical structure of the map. In literature G is often called *underlying graph* or *underlying network*. Some of the authors (for example [Benkert et al., 2007] or [Argyriou et al., 2010]) restrict the graph G to be connected, but the problem will be clearly equivalent as we can find the optimum on each of the graph components. These optima are independent so we can restrict ourselves on the connected underlying graph G. The nodes of graph G are often referred as *stations*.

Note that the graph G cannot be a multigraph, i.e. we do not allow parallel edges in G. This has a natural reason as it would be unclear which edge should be the first or the second in some ordering that has to compare these edges. No one can say which of the parallel edges goes for example northern. Moreover we could not be able to say whether two lines on parallel edges cross or not. To do that, we have to know which of the parallel edges goes for example northern of the other. That can be done by adding a new vertex into one of the parallel edges, so that they are no longer parallel.

The underlying graph G also cannot contain loops. Again if there was any in the graph we would not be able to compute the angle between the loop and other edges. Again if underlying structure requires some self loops, they can be modeled by adding two new vertices into the graph.

Definition 2.1 (Line) *Let $G = (V, E)$ be the underlying graph. Then a **line** is a graph $l = (V', E')$, where $E' \subseteq E$ and V' is a set of all vertices v such that there is an edge $(v, .) \in V'$.*

Each vertex of the line must be connected to some other vertex of the same line. Line is defined by the edges, not by its vertices. One line corresponds to an underground line, tram line or a tourist trail sign. In the definition 2.1, we do not put any restrictions on the lines, but in mentioned literature lines are studied as simple paths on the graph G. Only [Asquith et al., 2008, page 7] refers shortly to more general case when l is any binary tree. We will denote L the set of all lines.

If a line is not a connected graph, we can find the optimal drawing of each connected component separately. These optima are independent and as a result we can suppose that all lines are connected graphs. If this assumption does not hold true, then we can

split line l into lines $l_1, l_2, \ldots, l_k$, where l_i $(i = 1, 2, \ldots, k)$ represents one of the connected components of graph l and solve this equivalent problem.

Also note that no line l is multigraph. This is straightforward consequence of the fact that the underlying graph G is not multigraph. Let $e, f \in l$ be two distinct parallel edges that connect the same two vertices. As G does not have any parallel edges, e, f must be equal. But this is contradiction because we supposed e, f to be distinct.

Vertex $v \in V_l$ is called *terminal* of line l when $\deg_l v = 1$.[1] Otherwise v is called *internal station* [Argyriou et al., 2009] or *intermediate vertex* [Benkert et al., 2007]. When l is just single path, then the number of line terminals is 2 and all intermediate vertices are of degree 2 with respect to graph l. But when l can be arbitrary graph, we can introduce some more notation of the stations. Let us call the vertices with $\deg_l v = 2$ *inline vertices* and the vertices with $\deg_l v \geq 3$ *cross-vertices*. The figure 2.1 illustrates all these terms.

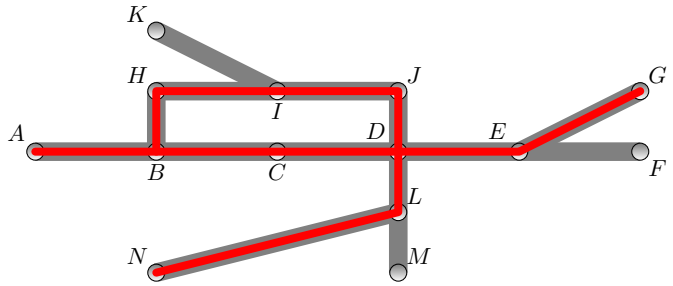


Figure 2.1: The grey lines represent an underlying graph G and red line is a line l on the graph G. Vertices A, N and G are terminals of the line l, vertices C, E, H, I, J and L are inline vertices of the line l and vertices B and D are cross-vertices. Vertices F, K and M do not belong to the line l.

Having defined the lines and the underlying graph, we can define the Metro-Line Crossing Minimization Problem.

Definition 2.2 (Metro-Line Crossing Minimization Problem)
Given an underlying graph $G = (V, E)$ and a set of lines L on that graph, find drawing of lines from L among the edges of G such that the number of crossings among pairs of lines is minimized.

In other words we have to find an ordering of lines along each edge in E. Let us denote L_e the set of lines that go through an edge e, i.e.

$$\forall e \in E : L_e := \{l \in L \mid e \in l\}.$$

Similarly we can define L_v as the set of lines that go through an vertex v, i.e.

$$\forall v \in V : L_v := \{l \in L \mid \deg_l v \neq 0\}.$$

2.2 Line ordering

In the Metro-Line Crossing Minimization Problem we have to find a total ordering $<_e$ for each edge e of the underlying graph, that minimizes the number of crossings. The total ordering can be then easily transformed into the drawing. There are many ways how to do that, but we will adopt the one stated by [Nöllenburg, 2010].

[1] By symbol V_l we denote the set of vertices of line l. By $\deg_l v$ we mean degree of vertex v in the graph l.

2.2.1 Line ordering on edge

Definition 2.3 (Line ordering on edge) *Let $G = (V, E)$ be the underlying graph and $e \in E$ an edge of this graph such that $e = (u, v)$. Let $l_1, l_2 \in L_e$ be a tupple of lines that go through the edge e. Then we say that*

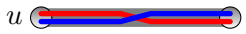
$$l_1 <_e^u l_2$$

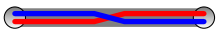
iff l_1 is drawn right of l_2 on e at the point u with respect to the direction of edge $e = (u, v)$.

It can be seen that the order of lines at particular vertex u is reversed if we switch the endpoints of the edge e. That means that $l_1 <_{(u,v)}^u l_2$ holds true if and only if $l_2 <_{(v,u)}^u l_1$. The line ordering allows us to define *edge crossing*. Two lines cross on an edge e if they switch their relative order on that edge.

Definition 2.4 (Edge crossing) *Let e be an edge of the underlying graph $G = (V, E)$ and l_1, l_2 two distinct lines from L_e. Then we say that lines l_1 and l_2* ***cross*** *at the edge $e = (u, v)$ if*

$$\left(l_1 <_{(u,v)}^u l_2 \text{ and } l_2 <_{(u,v)}^v l_1\right) \quad \text{or} \quad \left(l_2 <_{(u,v)}^u l_1 \text{ and } l_1 <_{(u,v)}^v l_2\right).$$





We write that $\text{cross}(l_1, l_2, e) = 1$. *Otherwise lines l_1 and l_2 do not cross each other on the edge e and the value of* $\text{cross}(l_1, l_2, e)$ *is* 0.

2.2.2 Vertex line ordering

We have already defined the line ordering on a common edge of two lines l_1 and l_2. But intuitively we can define a similar ordering of two lines on the vertices in which they split. Moreover we can define an ordering of lines incident to any vertex v using the ordering of lines on the common edge. If we choose a direction d, then we have given an ordering of edges incident with an arbitrary vertex v by the size of the oriented angle between d and the chosen edge, i.e. we can sort the edges clockwise.[2] This gives us a partial ordering of lines at each vertex using ordering of edges on which they lie. This partial ordering can be easily generalized to a total ordering by using the line ordering of lines that share the same edge.

The following definition 2.5 is not the only possible ordering that can be created above the underlying graph and the set of lines on it. For example [Bekos et al., 2007, page 9] states a unique numbering along all of nodes of the graph called *Euler tour numbering*. But this ordering does not work in all possible metro map graphs.

Just remind that $\measuredangle AVB$ stands for oriented angle between rays VA and VB. This angle is always between $\langle 0, 2\pi)$ and grows anticlockwise from direction of VA to direction VB. For convenience we make an agreement, that if VA is same as VB, then $\measuredangle AVB = 2\pi$ instead of 0. If two edges e, f share a common vertex v, i.e. $e = (u, v)$ and $f = (w, v)$, then by $\measuredangle ef$ we mean $\measuredangle uvw$ in the planar embedding of the underlying graph.

Definition 2.5 (Vertex line ordering) *Let l_1 and l_2 be two distinct lines on the underlying graph $G = (V, E)$, $v \in V$ a vertex and $e, f \in E$ edges such that $l_1 \in L_e$, $l_2 \in L_f$ and e and f are both incident to v. Moreover let d be a given direction. Then we say that*

$$l_1 \prec_{e,f} l_2$$

if and only if

- $l_1 <_e^v l_2$, *for* $e = f = (v, .)$;
- $\measuredangle de < \measuredangle df$ *otherwise.*

[2] Clockwise direction is chosen in order to maintain consistency with the line ordering on an edge.

From now we make the assumption that the direction d is fixed for all vertices. In a geographical layout there is one direction that has a privileged position. Hence it is natural to set d to go from south to north. We will make the following agreement. In order to have the edges sorted in the clockwise order, we want to have the edge in direction of d first, so we will take $\measuredangle dd = 2\pi$ instead of $\measuredangle dd = 0$, as we said before. This is just for our convenience, everything will work the same way if we did not make this assumption.

Now we can define the number of crossings of two lines at a vertex v. We will state an informal definition and then show an equivalent algebraical formulation. Let us choose one of the lines, call it l_1. Then drawing of line l_1 split the plane into $\deg_{l_1}(v)$ segments. Figure 2.2d illustrates how the plane is divided into these segments. Now let us focus on l_2, which lies in n of these segments. Line l_2 can lie in one segment without introducing any crossing with l_1. If l_2 lies in two segments there must be at least one crossing between l_1 and l_2. Similarly if l_2 lies in n segments, then l_1 and l_2 must have at least $n-1$ crossings. If we allow vertices to occupy some area in the plane, not only an infinitely small point, then the minimal number of crossings needed is $n-1$. This leads us to the following definition of *vertex crossing*. But there are also many other possible ways how to define the number of vertex crossing.

Definition 2.6 (Vertex crossing) *Let v be a vertex of the underlying graph $G = (V, E)$ and $l_1, l_2 \in L_v$ be two distinct lines going through this vertex. Let us have segments of the plane given by splitting the neighborhood of v by the drawing of the line l_1. Moreover let n be the number of these segments that contain the drawing of the line l_2. Then we define*

$$\mathrm{cross}(l_1, l_2, v) = n - 1$$

*as the number of **crossings** between the lines l_1 and l_2 at the vertex v. If $\mathrm{cross}(l_1, l_2, v) = 0$ then we say that the lines l_1 and l_2 do not cross each other at v.*

In the following text we will widely refer to this division or splitting of the plane. It will be the base point of many of the given proofs.

Figure 2.2 illustrates the concepts of the edge and vertex ordering as well as the edge and vertex crossing introduced in the preceding paragraphs. Previous definition 2.6 would be counterintuitive if the number of crossings between l_1 and l_2 was not equal to the number of crossings between l_2 and l_1. That formulates the following proposition.

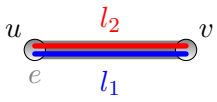
Proposition 2.7 *For any pair of lines l_1 and l_2 and any vertex v holds*

$$\mathrm{cross}(l_1, l_2, v) = \mathrm{cross}(l_2, l_1, v)$$

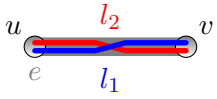
under the condition that these terms are defined.

Proof We split the area into segments similarly as in the definition 2.6. The only difference will be in the boundaries that will be formed with l_1 as well as l_2 edges. Call *mixed segments* such segments that have one boundary formed by a l_1 edge and the other by a l_2 edge. Suppose that there are m mixed segments in the drawing. Figure 2.3 shows this division.

We can observe that for each segment from definition 2.6 given by splitting the plane by l_1 edges that contains a l_2 edge there are exactly two mixed segments. That means that there are $\frac{m}{2} - 1$ crossings. But the same holds if we switch the role of l_1 and l_2. The number of the mixed segments will remain the same. Hence $\mathrm{cross}(l_1, l_2, v) = \mathrm{cross}(l_2, l_1, v)$ which completes the proof. ■



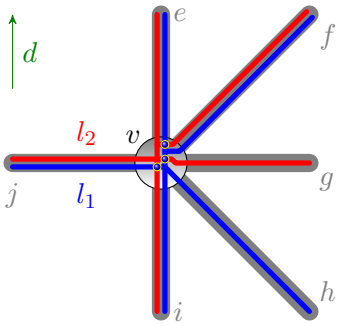
(a) An illustration of the *line order*. In this example holds $l_1 <^u_{(u,v)} l_2$, $l_1 <^v_{(u,v)} l_2$, $l_2 <^u_{(v,u)} l_1$ and $l_2 <^v_{(v,u)} l_1$. The lines l_1 and l_2 do not cross.



(b) An example of an *edge crossing*. At vertex u, we can write $l_1 <^u_{(u,v)} l_2$ and at vertex v holds the opposite inequality, i.e. $l_2 <^v_{(u,v)} l_1$. Hence lines l_1 and l_2 cross each other on the edge e. Moreover $l_2 <^u_{(v,u)} l_1$ and $l_1 <^v_{(v,u)} l_2$.

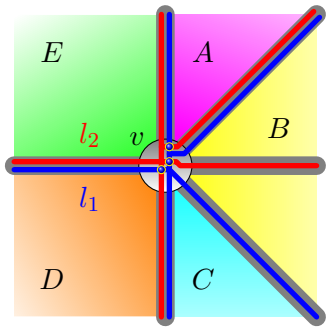
(c) An example of *vertex ordering*. The chosen reference direction d goes from south to north. The lines can be ordered in descending order by size of the oriented angle between d and edge. That means that their clockwise order is

$$(e, f, g, h, i, j).$$

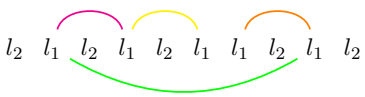


For example $l_1 \prec_{f,f} l_2$ because $l_1 <^v_f l_2$. Similarly $l_2 \prec_{i,i} l_1$, because $l_2 <^v_i l_1$. We can write $l_1 \prec_{h,g} l_2$, because $l_1 \in L_h$, $l_2 \in L_g$ and $\measuredangle dh < \measuredangle dg$. Similarly $l_2 \prec_{i,g} l_2$, $l_2 \prec_{j,f} l_1$ or $l_2 \prec_{j,e} l_2$. All in one, we can order lines clockwise (i.e. in descending order of the ordering $\prec$) as

$$\underbrace{l_2 l_1}_{e}\ \underbrace{l_2 l_1}_{f}\ \underbrace{l_2}_{g}\ \underbrace{l_1}_{h}\ \underbrace{l_1 l_2}_{i}\ \underbrace{l_1 l_2}_{j}.$$



(d) Line l_1 splits the vertex neighborhood into five area segments A, B, C, D, E. Line l_2 lies in four of them, namely in A, B, D and E. Line l_2 can go through one of these segments without crossing l_1, hence l_2 and l_1 cross each other three times. The yellow points show these three crossings.



(e) Figure related to section 2.3. The number of areas created by l_1 in which lies l_2 can be calculated algebraically. If we have the ordering of lines (which is in real cyclic) we can count the number of pairs of l_1 edges such that no l_1 edge lies between them and at least one l_2 lies between them. The colors in figure correspond to colors of the equivalent areas in figure 2.2d. We see four such areas, hence the number of crossings is 3.

Figure 2.2: An illustration to terms related with edge and vertex line ordering and crossing of lines.

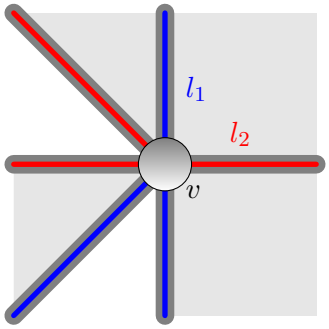


Figure 2.3: To the proof of proposition 2.7. Each of the segments given by the splitting is bounded by two of the l_1 or l_2 edges. The mixed segments are then marked with light gray color. The remaining white segments bounded by two edges of same color do not increase the number of crossings of l_1 and l_2. There are 4 mixed segments, hence the lines l_1 and l_2 cross once.

2.3 Another definition of vertex crossing

In this section an equivalent way to calculate the number of vertex crossings is formulated. This section is not necessary for understanding the rest of the thesis, but the proposition formulated in the following paragraphs will give us handy way how to calculate the number of crossings at vertex without drawing the graph.

If the edges from vertex v are clockwise ordered like $e_1, e_2, \ldots, l_n$ and if the vertex line ordering of all lines on v is $l_1, l_2, \ldots, l_n$, then we can say that first $|L_{e_1}|$ lines are only a permutation of L_{e_1}. Next $|L_{e_2}|$ lines are a permutation of L_{e_2} and so on. This means that the vertex line ordering only takes lines exiting the vertex v clockwise, starting from the north. In this ordering a pair of l_1 edges containing l_2 edge between corresponds to a segment given by splitting the plane by line l_1 edges on that l_2 edge is drawn. Any sequence of l_2 closed in a pair of l_1 corresponds to a situation when l_2 lies in the particular segment. This leads us to following proposition.

Proposition 2.8 *Let v be a vertex of the underlying graph $G = (V, E)$ and $l_1, l_2 \in L_v$ two lines going through this vertex. Let $l_{m_1}, l_{m_2}, \ldots, l_{m_n}$ be their vertex ordering, where $n = \deg_{l_1} v + \deg_{l_2} v$ and $\forall i \in \{1, 2, \ldots, n\} : m_i \in \{1, 2\}$. Then* $\mathrm{cross}(l_1, l_2, v)$ *is equal to*

- *the number of pairs of indices $i, j \in \{1, 2, \ldots, n\}$, such that $i < j - 1$, $m_i = m_j = 1$ and $\forall k \in \{i + 1, i + 2, \ldots, j - 1\} : m_k = 2$, minus one, if $m_1 = m_n = 1$;*
- *the number of pairs of indices $i, j \in \{1, 2, \ldots, n\}$, such that $i < j - 1$, $m_i = m_j = 1$ and $\forall k \in \{i + 1, i + 2, \ldots, j - 1\} : m_k = 2$, otherwise.*

Proof We will use the concept of mathematical induction on $n = \deg_{l_1} v + \deg_{l_2} v$ to prove this proposition. The minimal possible n is equal 2, as $l_1, l_2 \in L_v$. Then two lines terminate at the vertex v and they do not cross each other. The only possible orderings are $l_1 l_2$ and $l_2 l_1$ and there is no pair of indices i, j from the proposition. So for the trivial case the proposition holds.

Now suppose that the proposition holds for some n.

- If we draw new edge of line l_1 between two lines l_2 we add one crossing, because we split one segment with border l_1 into two segments containing l_2. Algebraically if the l_1 was added somewhere in the middle of the ordering (not at first or last position), the number of pairs i, j increased by one. The value of m_1 and m_n clearly remains the same. If the position was first or last then initially $m_1 = m_n = 2$ and we introduced exactly one pair of i, j.

- If the l_1 was drawn between l_1 and l_2, then there was introduced one segment without l_2. That means that the number of crossings remained unchanged. If l_1 was added somewhere middle into the ordering, the number of indices i, j remained unchanged as well as m_1 and m_n. If l_1 was added first or last, then initially one of m_1 and m_n was 1 and the other 2. If we added l_1 before (or after if $m_n = 1$) l_1 then nothing changed, if we added l_1 before (after) l_2 we introduced one new pair of indices i, j, but since now $m_1 = m_{n+1} = 1$. Hence the number crossings given by the proposition remains unchanged.
- The last possible situation is when l_1 is added between two edges of l_1. We split one segment without l_2 into two which means that the number of crossings remains same. If l_1 was added in the middle of the ordering, the number of pairs i, j remained the same as well as m_1 and m_n. If l_1 was added at beginning or at the end, no new pair of i, j was introduced. The first and last index m remained equal to one, so the number of crossing is still equal.
- If we introduce l_2 instead of l_1 we do not introduce any new segments, but we can from a segment that did not contain l_2 make one containing l_2. The proof is similar to the situation when we add new l_1 edge, so we will omit it. Again it is needed to handle with three situations. Other possible way is to take advantage of symmetry of crossings $\text{cross}(l_1, l_2, v)$ that we proved in proposition 2.7. That means that it is possible to switch marking of line l_1 and line l_2.

We see that for each of the particular situations that can arise the proposition holds for $\deg_{l_1} v + \deg_{l_2} v = n+1$ under condition that the proposition holds for $\deg_{l_1} v + \deg_{l_2} v = n$. We can conclude using the principle of mathematical induction that proposition holds for each $\deg_{l_1} v + \deg_{l_2} v \in \{2, 3, \ldots\}$. ■

This proposition gives us not really straightforward solution how to calculate the number of crossings manually, but this formula can be easily transformed into a computer program, because it calculates only with algebraical structure of the ordering. In only single linear scan of the ordering we can find out how many times two lines cross.

Figures 2.2d and 2.2e illustrate proposition 2.8 and connections between the geometric and algebraic intuition of vertex crossing. Now we are ready to state an alternative, more formal, definition of the Metro-Line Crossing Minimization Problem which is equivalent to the original one.

Definition 2.9 (Metro-Line Crossing Minimization Problem)
Given an underlying graph $G = (V, E)$ and a set of lines L on that graph, find an ordering $\prec$ for each vertex $v \in V$ such that

$$\underbrace{\sum_{e \in E} \left(\sum_{l_1, l_2 \in L_e | l_1 \neq l_2} \text{cross}(l_1, l_2, e) \right)}_{\textit{number of edge crossings}} + \underbrace{\sum_{v \in V} \left(\sum_{l_1, l_2 \in L_v | l_1 \neq l_2} \text{cross}(l_1, l_2, v) \right)}_{\textit{number of vertex crossings}}$$

is minimal among all feasible orderings $\prec$.

2.4 Common subpath

We will define one more term which is widely used in the literature and we will use it extensively in the following chapters. The *common subpath* is a sequence of common edges of two lines, such that they are represented by a single path on the underlying graph. Moreover, we want this sequence to be as long as possible. We do not permit common subpaths to contain cross vertices.

Definition 2.10 (Common subpath) *Let l_1 and l_2 be two lines on the underlying graph $G = (V, E)$. Then a* ***common subpath*** *is any nontrivial path $e_1e_2 \cdots e_n$ through vertices $v_0, v_1, \ldots, v_n$ in graph G such that $\forall i = 1, \ldots n-1$ holds*

$$\deg_{l_1} v_i = \deg_{l_2} v_i = 2$$

and there is no longer path through vertices $v_0, v_1, \ldots, v_n$ that meets this condition.

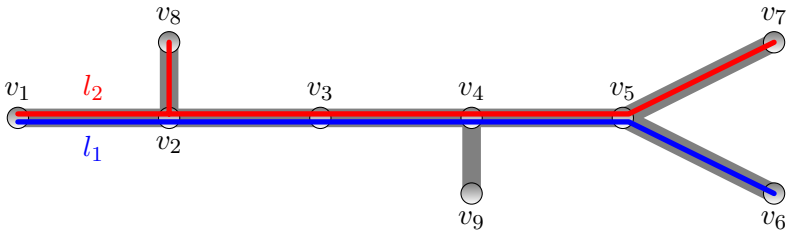


Figure 2.4: Illustration to the term *common subpath* defined in 2.10. There are two common subpaths of l_1 and l_2, namely v_1v_2 and $v_2v_3v_4v_5$. Path v_2v_8 is not common subpath as there is only l_2, path v_2v_3 is not common subpath as there exists longer common subpath through v_2 and v_3. Finally $v_1v_2v_3v_4v_5$ is not common subpath of l_1 and l_2 as $\deg_{l_2} v_2 = 3$.

Chapter 3 Problem variants and related literature

In this chapter we will present some variants of the Metro-Line Crossing Minimization Problem studied in previous works. We will describe their properties and briefly look on the known algorithms for solving these problem sub types. Finally we touch the related problem named Metro-map Layout Problem and we will focus on differences between the Metro-Line Crossing Minimization Problem and the Metro-map Layout Problem.

3.1 Problem variants

This section summarizes possible variants of the problem. The definition from section 2.1 allows us to place some other requirements on the desired ordering of lines (or drawing of lines). As we introduced in previous sections, one area where may problems differ is in the lines set L. Lines can be single paths, trees or arbitrary graphs. Example of maps, where lines are just single paths, is the most of underground maps. Examples of tree lines are public transportation maps in bigger cities and good example of a map, where lines are arbitrary graphs, is any tourist trail map. In section 1.3 we showed examples of such lines and compared them on figure 1.2, that displays clips from public transportation maps.

The other literature is restricted on the path lines case, hence the problem variants stated by the other authors are mostly defined for paths lines, but some of them can be generalized for arbitrary model.

3.1.1 k-side model

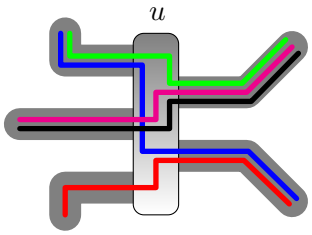
At first, we will talk about k-side model and its variants. Here the problem definition does not change at all, but we place some further requirements on underlying graph $G = (V, E)$ and its planar embedding. Most of good public transportation maps have edges drawn octilinearly, i.e. in eight cardinal directions. But this puts a restriction on vertices of the underlying graph. If edges leave a vertex in eight directions, its degree cannot be clearly greater than 8 unless we allow two edges leave the same side of the station. So then we can represent any vertex like an octagon with lines leaving at its sides. This model is called 8-side model.

Many publications solve the problem under 2-side model or 4-side model, when we allow edges to leave the node in 2 or 4 directions. If we allow the lines to exit one of the vertex sides under any direction, then no condition on the vertex degree holds. We will again illustrate this on the figure 1.2. The map 1.2b is drawn under 2-side model, i.e. all lines enter the node in one side of the vertex and exit on the opposite. On the contrary the map in 1.2a is drawn under 4-side model, the lines enter and exit in one of four possible directions.

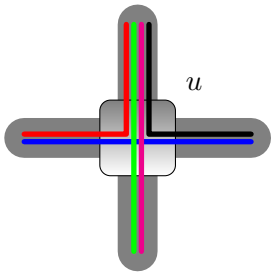
Tourist trail map does not need to have the edges drawn in some preferred directions, but they should match the geographical coordinates. Thus vertices can be represented like circles or points and lines can enter or exit wherever is desired. We can talk about this as ∞-side model.

3.1.2 Restrictions on terminal stations

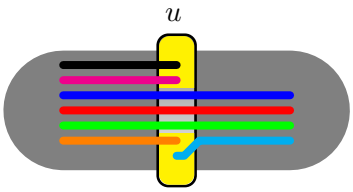
There is also another area where the problem definition may differ. It is sometimes not desired that the middle line ends at some node and the two around continue. This problem solves the *periphery condition*. It states that no terminating line can lie between two continuing lines in the $<^v_{(u,v)}$ order. More formally suppose that vertex v is terminal of line $l_1 \in L_e$, such that $e = (u, v)$. If there exists another $l_2 \in L_e \setminus \{l_1\}$, such that $\deg_{l_2} v \geq 2$, and $l_1 \leqslant^v_e l_2$, then all lines $l'_2 \in L(e) \setminus \{l_1\}$, such that $\deg_{l_2} v \geq 2$, must fulfil the same inequality.



(a) An example of a vertex under 2-*side model*. The edges enter or exit the node u only in two given sides.



(b) An example of a vertex under 4-*side model*. The edges enter or exit the node u only in four given sides.



(c) Illustration to the *periphery condition*. The lines that terminate at vertex u are allowed to terminate only in yellow area called station ends. This variant is called MLCMP-SE. It can be given which line should terminate at top or bottom station end. Then the problem is called MLCMP-FixedSE.

Figure 3.1: Illustration to terms related with k-side model and the periphery condition

The Metro-Line Crossing Minimization Problem after adding the periphery condition is called MLCMP-SE, where SE stands for *station ends*. This name firstly appears in [Asquith et al., 2008] and [Bekos et al., 2007]. The [Nöllenburg, 2010] calls this problem MLCMP-P, where P stands for *periphery*. In [Bekos et al., 2007] is showed that MLCMP-SE is NP-hard even if the underlying graph G is a single path.

We can make this condition even stronger and say on which end of the $<^v_{(u,v)}$ order should terminating line lie. We can illustrate it best on the 2-side model. Let us assume that all lines enter the node v from left and exit on the right side. For each line l_1 with terminal v, we can specify that l_1 terminates at the top or at the bottom of the node. For example if line l_1 goes from left to right and ends at v at the top, then for all continuing lines l_2 (i.e. $\deg_{l_2} v \geq 2$) holds $l_2 <^v_{(u,v)} l_1$.

At which end of the node shall any terminating line lie is the input of the problem

together with the lines. As [Nöllenburg, 2010] mentions, this situation arises when we want to illustrate some physical structure of the terminal station. For example trams turn around on a specified side of the rail which goes along. We may want to present this fact in the map, so we draw the line terminating on the given side of the vertex.

This variant of the problem is in most of the literature ([Bekos et al., 2007], [Asquith et al., 2008] and [Argyriou et al., 2009]) called MLCMP-FixedSE. The name shows the relation with the MLCMP-SE. Again [Nöllenburg, 2010] calls this problem with a different name - MLCMP-PA, or longer Metro-Line Crossing Minimization Problem *with periphery condition and terminus side assignments*.

There is one more special case of the problem. We can restrict that lines terminate only at vertices of the underlying graph $G = (V, E)$ that have their degree equal to one. This means that if G is a tree, than lines may terminate only in leaves of G. This situation is not really realistic in many complex transportation maps with trams or buses. This situation mostly arises in maps where only underground lines are drawn. The metro lines end in the outskirts of the city and go through the city center to the opposite of the city. This problem is called by [Nöllenburg, 2010] MLCMP-T1, where the name illustrates that lines terminate only in vertices of degree 1.

3.2 Algorithms for solving the Metro-Line Crossing Minimization Problem

The aim of this sections is to shortly describe the algorithms developed to solve the Metro-Line Crossing Minimization Problem. We will briefly introduce five algorithms presented in the literature and we will shortly discuss their properties.

The first algorithm developed to solve the Metro-Line Crossing Minimization Problem was published in [Benkert et al., 2007]. This work defined the original variant of the Metro-Line Crossing Minimization Problem and first attempted to solve the problem programmatically. Their algorithm solves the *one edge layout*, i.e. calculates how to draw lines on one edge of underlying graph G so they cross each other as few times as possible.

The algorithm uses dynamic programming and its running time is $\mathcal{O}\left(|L_e|^2\right)$, where e is the edge on that we calculate the layout. However, as the authors say, they could not generalize this approach to more general case. Their dynamic program was based on some interesting properties that the one edge layout have, but they do not hold even if we consider two neighboring edges of the graph G. The authors say that the problem cannot be divided into two independent instances, which is the main requirement of the dynamic programming. As [Nöllenburg, 2010] says it still remains an open problem, whether the original Metro-Line Crossing Minimization Problem without periphery condition is NP-hard.

Second publication we would like to mention is [Asquith et al., 2008]. There is the problem formulated as a set of binary conditions that represent the potential crossings. Then an objective function represented as a sum of these binary conditions. This function is then minimized using the integer linear programming. The algorithm restricts the input to the MLCMP-SE in the plane graph, which is NP-hard.

Because ILP requires exponential time for finding the optima, they developed a heuristic algorithm for this problem. This heuristic shortened time to $\mathcal{O}\left(|L|^3 \cdot |E|\right)$, but the algorithm is no longer guaranteed to find the optimal solution. If the input fulfils some conditions then the solution is optimal.

The publication also shows that the MLCMP-FixedSE instance can be solved under the running time $\mathcal{O}\left(|L|^3 \cdot |E|^{2.5}\right)$.

Next work that is worth to discuss is [Bekos et al., 2007]. The beginning of the paper shows that the MLCMP-SE is NP-hard even on a path. Later it presents a polynomial time algorithm for the MLCMP-FixedSE. It mentions many properties of the optimal solution on a path and a tree. It shows that two lines cross each other at most once or two lines that have a common terminal station do not cross each other.

The author presents a polynomial time algorithm for solving the MLCMP-FixedSE under 2-side model where underlying graph is a path and the MLCMP-T1 in a tree of fixed maximal degree d. Their asymptotic running time is $\mathcal{O}\left(n + \sum_{l \in L} |l|\right)$. The algorithm topologically sorts the vertices and then creates the unique *Euler tour numbering*. Now the algorithm finds the ordering of lines at each node by handling one of three cases of indegree of the node. After applying simple rules, the algorithm is guaranteed to produce an optimal solution without redundant crossings.

Work [Argyriou et al., 2010] presents a polynomial time algorithm that solves the MLCMP-T1 under k-side model in running time $\mathcal{O}\left(\left(|E| + |L|^2\right) \cdot |E|\right)$. For special case of 2-side model is presented an algorithm with running time $\mathcal{O}\left(|V| \cdot |E| + \sum_{l \in L} |l|\right)$. If this algorithm is run on the MLCMP-FixedSE instance, then the running time is $\mathcal{O}\left(|V| \cdot (|E| + |L|)\right)$.

The algorithm is recursive and solves k-side model. The base of recursion solves the trivial network consisting of one central station v and some terminals directly incident to v. Hence all lines in the problem have the length of 2. Then lines are organized into bundles and two lines in a bundle have a common terminal. Clearly lines in a bundle can be switched without influencing the optimality. Now the bundles can be sorted by using the *Euler tour numbering* introduced in [Bekos et al., 2007]. The output is the optimal solution to this simple problem with only inevitable crossings introduced.

Now it is time for the harder part of the algorithm when it is needed to concatenate these simple nodes. The algorithm connects firstly bundles with the same set of lines and splits bundles into two if they differ in some set of lines. These splitted bundles are now sorted and connected if they contain the same set of lines.

The authors used similar ideas as for k-side model also for 2-side model. The problem is restricted, hence it can be solved quickly using some special facts as that lines are x-monotonous. This means that they cannot make 180° turn, as it is prohibited to enter and exit node on the same side.

Last work to mention is [Nöllenburg, 2010]. This work presents a greedy algorithm that is the fastest one between all other solutions to the Metro-Line Crossing Minimization Problem variants MLCMP-FixedSE (in work called MLCMP-PA) and MLCMP-T1. Its asymptotic running time is equal to $\mathcal{O}\left(|L|^2 \cdot |V|\right)$. The MLCMP-T1 is converted in linear time into a MLCMP-FixedSE instance which is then solved.

In the beginning the work carefully compares known results in solving the Metro-Line Crossing Minimization Problem as well as the problem variants. Then it presents the new algorithm that is based on the fact that all crossings in the optimal layout are unavoidable. The proof of this fact is given. The algorithm is greedy and it does not need any heuristics. The first step calculates all common subpaths of all pairs of the lines. If we take one of them, we can decide the lines relative order based on the direction in which each of the lines leaves the common subpath. Now we know whether the lines have to cross each other on the common subpath or one of them is ordered before the second. In the second step the knowledge of relative order of each pair of lines allows to construct the order $<$ of lines on each edge. Hence the output is optimal solution to the MLCMP-FixedSE problem.

3.3 Metro-map Layout Problem

The Metro-map Layout Problem is closely related problem to the Metro-Line Crossing Minimization Problem. It is a more complex problem where we do not have to find only the ordering of lines among the edges, but also the planar embedding of the underlying graph. Over time there have developed many methods for automatic drawing of the metro maps. As the input is given an underlying graph with some physical structure or an initial solution together with set of lines. The output is a metro map that represents this graph.

3.3.1 Layout criteria

The authors dealing with the Metro-map Layout Problem formulated a set of criteria that a good metro map should fulfill. Some of the criteria are strict (or hard) and some of the criteria are soft. Every map must fulfill the strict criteria, but the soft criteria are fulfilled only as good as possible. The criteria differ a little bit between publications, but the skeleton remains the same. The following list shows some possible hard constraints.

- Metro map layout should respect the topology of graph representing physical connections.
- All edges of the metro map must be *octilinear* line segments. Only eight cardinal directions are allowed. Note that the firsts works [Hong et al., 2004] and [Jonathan M. Stott, 2004] did not consider this condition as hard but as soft.
- Each edge is long at least l_{min}.
- Each edge has a minimum distance d_{min} from any nonincident edge.
- Lines are drawn with different colors.
- No edges or labels overlap.

The constants l_{min} and d_{min} are given as part of the input. And the soft constraints are next four. The output should optimize them as good as possible without breaking the hard constraints.

- The metro map lines should have as few bends as possible.
- The sum of edge lengths should be as small as possible.
- For each pair of adjacent vertices their relative position should be preserved.
- The lines cross each other as few times as possible.

We see that minimization of crossings that we studied in the Metro-Line Crossing Minimization Problem is only one part of the Metro-map Layout Problem. There are also some differences in behavior of the system. For example contracting the paths containing only vertices with degree of two has no influence on the Metro-Line Crossing Minimization Problem. But in the Metro-map Layout Problem the influence will be high, because by contracting we forbid bends in the edge. Big problems produce labels placing, because they are not permitted to overlap. Many works deal with labels by introducing new thick lines (containing labels) that can be drawn only at top or bottom of any edge. Therefore the Metro-map Layout Problem is much more difficult. Setting the directions of lines to octilinear makes the Metro-map Layout Problem NP-hard.

Also note that the Metro-Line Crossing Minimization Problem has a clear optimum. We only count the crossings and by comparing this count we easily see which of the algorithm outputs is better. But in the Metro-map Layout Problem we do not have such clear criterion. The soft constraints can be weighted and output can slightly change after modifying these weights. A computer cannot tell us which map is better and a human can only give his personal opinion.

3.3.2 Methods of solving MMLP

The work [Wolff, 2007] discuss the developed methods for solving Metro-map Layout Problem and it is good comparison of works of previous authors. The authors of [Nollenburg and Wolff, 2011] sent a printed version of automatically drawn metro maps to people that are considered to be experts in drawing metro maps and asked them questions about the maps. The result is that official metro maps created by graphic designers are better, but the layout produced by computer is quite usable and can be used as a draft for graphic designer.

Now there are many methods for solving this problem. Work [Hong et al., 2004] solves the Metro-map Layout Problem using the spring embedding algorithm. The algorithm models map layout like a physical system with forces acting on the layout. Hence the final layout is not strictly octilinear. There is also one important difference from other works that the layout has no edge crossings, while vertex crossings are preferred. Later works prefer edge crossings instead of vertex crossings as the maps are better readable.

Another related work is [Jonathan M. Stott, 2004]. The used algorithm is a hill climbing algorithm which minimizes the weighted sum of five metrics that measure how much the actual layout breaks the soft criteria. The algorithm is finding the optima via random movements and hence is slow. The output layout is also not strictly octilinear. The work of [Nöllenburg and Wolff, 2006] produces the strictly octilinear layout. The published algorithm uses a mixed integer program to minimize the similar weighted sum of criteria. Layouts produced by this algorithm are better than these produced by the previous two algorithms, which proves a survey in [Nollenburg and Wolff, 2011].

Chapter 4 An algorithm for MLCMP

The purpose of this chapter is to present a new algorithm for solving the Metro-Line Crossing Minimization Problem in its most general form. Now we consider that the underlying graph as well as lines are arbitrary graphs. This means that the underlying graph is planar and does not contain any parallel edges (in other words, the underlying graph is not a multigraph). Lines are only subgraphs of the underlying graph.

On the opposite the underlying graph as well as the lines can contain arbitrary cycles and they do not have a bounded degree. We also do not require the periphery condition, the lines can end at any end of the stations as well as in the middle between two continuing lines.

4.1 Data properties

As we said in the introduction of this chapter, the solved problem is only more general case of the Metro-Line Crossing Minimization Problem solved in previous works as they were discussed in section 3.2. The motivation for this generalization is a plan to minimize the crossings in tourist trail maps. But there are some differences between drawing metro schemes and drawing tourist trail maps.

Firstly there are many lines in the underground maps, but they are simpler compared to the tourist trails. Almost all of them are just single paths. On the other side there is a small number of tourist trail signs (namely four or five) in the tourist maps. Their structure is very complicated and they consist of many connected components that can contain cycles.

But there is one additional condition on the final solution. It is the preference of edge and vertex crossings. The edge crossings in the optimal solution can be moved into vertex in such way that we do not increase the number of crossings. This holds under the model we adopted, but under another model it does not need to be true. But this statement is not equivalence. Only some vertex crossings can be moved on incident edges without influencing the number of crossings. In metro maps the edge crossings are preferred (with a few exceptions), but in tourist trails maps only the vertex crossings are natural. Hence we allow lines to cross at arbitrary vertex but not on the edges.

It makes no sense to talk about k-side model in tourist trail maps as lines can enter or exit in arbitrary direction. Hence the tourist trail maps will use the ∞-side model.

4.2 Overview of the algorithm

In this section the used technology for solving the problem is described. The presented algorithm consists of two parts - the greedy part followed by searching the state space as a constraint satisfactory problem.

4.2.1 Preprocessing steps

At first the rough input is modified in order to get a simpler problem. At the beginning the graph edges are *contracted*. In the input graph (given for example as OpenStreetMap XML file), there are often paths whose subsequent edges contain exactly the same set of lines. Moreover if this path consists of vertices $v_0v_1v_2 \cdots v_n$, then all of the inner vertices $(v_1, v_2 \cdots v_{n-1})$ are of degree two. Informally it can be said that on this path "nothing happens". This path can be contracted into a single edge v_0v_n without influencing the optimal number of crossings.

In the Metro-Line Crossing Minimization Problem the contraction of edges has no influence on the solution, but it can improve the running time. In a tourist trails map the contraction of lines is an inevitable preprocessing step. For example bends in the trails are modeled by many incident vertices with the same set of lines. The problem complexity decreases dramatically after replacing them with a single edge. But in order to have correctly defined ordering, we have to keep the angles in which edges leave and enter the vertex.

Second straightforward step of preprocessing is splitting the problem into connected components. This can be done when the underlying graph is not connected. The optima in each pair of these components are independent as there is no line connecting them.

4.2.2 Main part of the algorithm

After the preprocessing it is time for a *greedy part* of the algorithm. This greedy part follows some simple rules for preserving optima on some local areas. These rules are such that it is still very likely to find the optimal graph with a minimal number of crossing. Some criteria can lead to all possible optimal solutions, some only to few of them. Some of them may not hold in the optimal solution, but if they will not, we will find that out and we will have to search whole state space then. We will discuss these greedy criteria in section 4.3. As a result the algorithm cannot get stuck at a local optimum.

The overall idea is following. We will find a lower bound of crossings that is in many cases sharp.[1] Next we will try to find a solution that reaches this lower bound. If we are not able to do that then we will have to search whole state space. The case when the lower bound is not reached is not realistic for any real data input, it is only a special model situation.

When all greedy rules are applied, it is time for searching the optima within the rest of all possible solutions. After the greedy part the search space is much smaller thus it is possible to search it entirely. There is also an added pruning that decreases the number of the possible solutions that are needed to be searched. Then from all feasible solutions is chosen that one with the minimal number of line crossings. We will more deeply discuss this part of the program in section 4.6.

4.3 Greedy order

In this section we will state some mathematical theory that gives us sufficient background to create a set of rules that can be applied to the problem. These rules are such that there is not worry that the optimal solution will be sacrificed. We define some properties of the graph and we will show the following. If there is a graph in that these properties are fulfilled then this graph must be optimal.

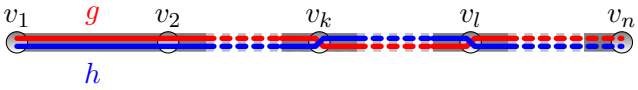
For the convenience of the reader we will repeat a theorem published in [Asquith et al., 2008], because our own propositions will be based on this theorem. But we need

[1] By a *sharp* lower bound we mean such a lower bound that there is no bigger.

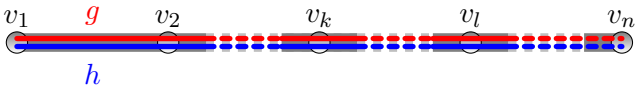
to present a new proof because that one written in [Asquith et al., 2008] requires the periphery condition that our input does not meet.

Theorem 4.1 (Asquith et al., 2008) *In an optimal solution every pair of lines in L will cross at most once along any common subpath.*

Proof The proof will be done by contradiction. Suppose that there are two lines g, h in an optimal solution S such that they cross each other at least twice on the common subpath $v_1v_2\cdots v_n$. Denote v_k and v_l two vertices where the lines cross. Let us denote S' solution where lines g and h are switched on the path $v_kv_{k+1}\cdots v_l$. Figure 4.1 is illustration to this situation.



(a) Two lines g and h cross each other twice in solution S. This is only illustrative example. Line g can be drawn under the line h at the vertex v_n. Lines g and h may also cross between v_k and v_l.

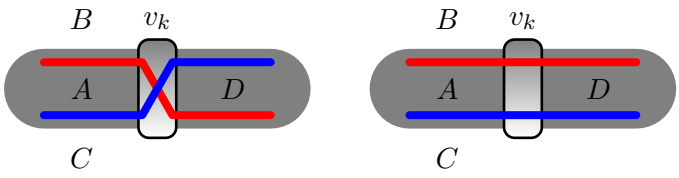


(b) Situation in the solution S' after switching lines g and h between v_k and v_l. There might be other lines as well. If a line was drawn between g and h it remains there after exchanging g and h.

Figure 4.1: Illustration to the proof of theorem 4.1

We will show that the number of crossings in S' is strictly less than the number of crossings in the solution S that was supposed to be optimal. Remind that all vertices v_i, where $i \in 1, 2, \ldots, n$, are inline vertices for both lines g and h as a path defined by them is a common subpath of the lines g and h. This follows the definition 2.10.

- At vertices $v_1, v_2, \ldots, v_{k-1}$ and $v_{l+1}, v_{l+2}, \ldots, v_n$ remained the order of all lines same. Hence the number of crossings of all lines did not change at all.
- At vertices $v_{k+1}, v_{k+2}, \ldots, v_{l-1}$ are in S' the drawings of lines g and h switched without affecting any other line. As a result all lines that crossed the line g in the solution S now cross the line h in S' and vice versa. At these vertices the number of crossings is same in both solutions.
- Now we will show that at the vertex v_k the number of crossings strictly decreases. The proof for the vertex v_l is same. We split the area into four regions named A, B, C, D as it is shown in the figure 4.2. This is similar to the split in the definition 2.6. Other lines can occupy any of these regions, but we will show that for any combination the number of crossings with g and h will not grow.
  - Lines that occupy only region D do not have any crossing with g or h at v_k in both solutions.
  - Lines that do not occupy region D have same number of crossings with g and h as D is the only area that moved from one region bounded by g (or h) on the other side of that line.
  - Lines that lie in D and B (the proof for the possibility D and C is similar, only g and h have to be switched) have one crossing with h and none with g in S. In S' they cross g but not so h.



(a) State in the solution S. (b) State in the solution S'.

Figure 4.2: Illustration to the proof of the theorem 4.1 that shows the neighborhood of the vertex v_k divided into four regions A, B, C and D.

- Lines that occupy A and D cross both lines in S, but neither of them in S'.
- Lines that lie in D, A and B (or similarly D, A and C) have one crossing with both of lines g and h in S, but in S' they cross only g. Hence the number of crossings is smaller.
- Lines that lie in D, B and C (or similarly in all four regions) have one crossing with g and one crossing with h in S as well as in S'. The number of crossings remained the same.

We see that the number of crossings between g, h and other lines did not increase by moving to S'. In some cases it could decrease, but such lines do not need to exist. But lines g and h crossed each other in S, which is no longer true in S'. Hence the number of crossings at v_k in S' is smaller than in S.

Summing up we have seen that the number of crossings is strictly smaller in S' than in S due to vertices v_k and v_l. But this is contradiction with the assumption that S is optimal. Hence the propositions holds. ■

4.3.1 Preferred order at common subpath end

As far we know that two lines can cross each other only once at each common subpath. This leads to many consequences, because we can reduce the number of feasible solutions. The reason is that most of the feasible solutions do not fulfill the condition 4.1 and hence they cannot be optimal. At first we will define the common subpath endpoint preferred order.

Let u be an endpoint of a common subpath of lines l_1 and l_2 and $e = (u, v)$ be the edge incident with u of this subpath. Now we can sort these lines (in other words their edges incident to u) in ascending order of $\prec$ ordering similarly as we did on figure 2.2e. Let us leave from this order the edge e.

Remind the definition 2.6 of the vertex crossing. Switching the order of lines l_1 and l_2 on the edge e influences only the size of two sectors neighboring e given by splitting the plane by line l_1 edges. This observation leads us to the idea of defining the *preferred order at vertex* u. It is such order that the number of crossings of l_1 and l_2 at u is minimized with respect of choosing order of lines l_1 and l_2 at e.

We will say that the vertex preferred order is precedes or follows if line order of l_1 and l_2 tends to be precedes or follows on e. We will say that the order does not matter if lines tend to both these orders equally. Now suppose only edges incident to u. The order to that lines l_1 and l_2 tend to can be seen from the closest edges to e that contain l_1 or l_2. This is more formally described in definition 4.2 and proposition 4.3.

Definition 4.2 (Vertex preferred order) *Let u be an endpoint of a common subpath of lines l_1 and l_2 and e be the corresponding last edge of this common subpath. We say that the preferred order of lines l_1 and l_2 at u with respect to e*

- ***does not matter** if u is a terminal vertex of l_1 or l_2.*

Otherwise we define l_c the first of lines l_1 and l_2 encountered clockwise from direction of edge e on any edge incident with u. Similarly let l_a be the first of these two lines in an anticlockwise direction. We say that the preferred order of lines l_1 and l_2 at u with respect to e

- *does **not matter** if $l_a = l_c$;*
- *is **follows** if $l_a = l_1$ and $l_c = l_2$;*
- *is **precedes** if $l_a = l_2$ and $l_c = l_1$.*

Figure 4.3 illustrates the term vertex preferred order. Without the first part the definition will not make sense, because the first encountered line would have been l_1 or l_2 on the edge e. But switching the line order will cause the switch of l_a and l_c. If it cannot lead to confusion we will say instead of "preferred order of lines l_1 and l_2 at u with respect to e" just "preferred order of lines l_1 and l_2". Sometimes we will also write for short $\mathrm{pord}(l_1, l_2, u, e)$. The following proposition shows why the names of preferred order were chosen in that way.

Proposition 4.3 (Vertex preferred order is optimal) *Let us have fixed the order of lines l_1 and l_2 on all edges incident with u except their common edge $e = (u, v)$. Then the number of crossings $\mathrm{cross}(l_1, l_2, u)$ is minimal for*

- $l_1 <^u_{(u,v)} l_2$ *if the preferred order of lines l_1 and l_2 is precedes,*
- $l_2 <^u_{(u,v)} l_1$ *if the preferred order of lines l_1 and l_2 is follows,*
- *both $l_1 <^u_{(u,v)} l_2$ and $l_2 <^u_{(u,v)} l_1$ if the preferred order of lines l_1 and l_2 does not matter,*

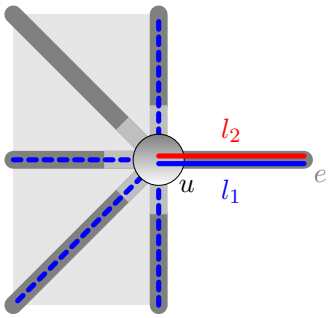
with respect to order of lines l_1 and l_2 on the edge e.

Proof The proof can be done by simply counting the number of crossings of l_1 and l_2 at u. As it was said in the introduction with having the order fixed on all edges incident to u except e, the number of crossings is influenced only at neighboring segments of the plane given by splitting it by l_1 edges.

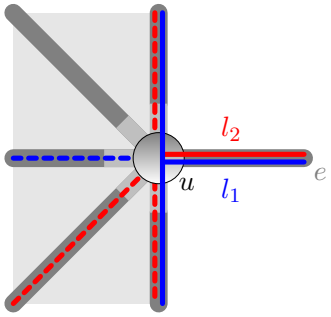
We split the area into three segments A, B and C similarly as we did in the proof of theorem 4.1. This is illustrated in figure 4.4. If $l_a = l_2$ and $l_c = l_1$ then the preferred order is precedes and l_2 lies in one segment (union of B and C) given by l_1 edges. Otherwise it lies in two. Similar situation arises in the symmetric case of preferred order follows.

Suppose that preferred order does not matter. If u is terminal for l_1 (or similarly l_2), then switching l_1 and l_2 order does not introduce any new crossing between l_1 and l_2. Same holds if $l_a = l_c = l_2$. ■

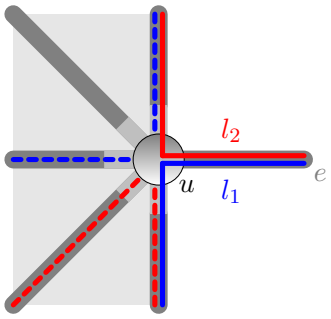
We can observe one interesting property. If the vertex preferred order is precedes or follows then by breaking this preferred order we may introduce new crossings between l_1 or l_2 and other lines. Recall the figure 4.4. If there was a third line l from sector D to sector B, then we introduced one crossing with l_1 and one with l_2 by exchanging l_1 and l_2. Other lines were not influenced. Also with not matter trail crossings between l_1, l_2 and other lines will not be same all the time.



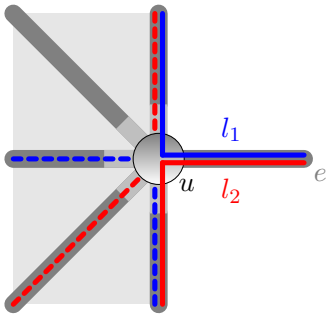
(a) Example of situation when the order of lines l_1 and l_2 does *not matter* on e. Reason is that u is a terminal station of l_2.



(b) Another example of order that does *not matter*. Line l_1 can be found in clockwise direction as well as anticlockwise. Hence if we switch the order of lines on e, the number of crossings of l_1 and l_2 will remain the same at u.



(c) The preferred order of lines l_1 and l_2 on e with respect to u is *precedes*. Here the clockwise line l_c is l_1 and the anticlockwise l_a is l_2. Changing the order of l_1 and l_2 on e is possible, but it will introduce one crossing at u.



(d) The preferred order of lines l_1 and l_2 on e with respect to u is *follows*. The situation is similar to that one on figure 4.3c, but the drawings of l_1 and l_2 are switched. Hence in clockwise direction is l_2 and in anticlockwise direction is l_1.

Figure 4.3: Illustration to the term *vertex preferred order*. Individual figures show examples of the preferred order of lines l_1 and l_2 on an edge e with respect to a vertex u. Lines in the light grey area can be drawn arbitrary, depending only on the color (i.e. whether it is line l_1 or l_2) of the first line l_c in clockwise direction and the first line in anticlockwise l_a direction from e such that $l_c, l_a \in \{l_1, l_2\}$.

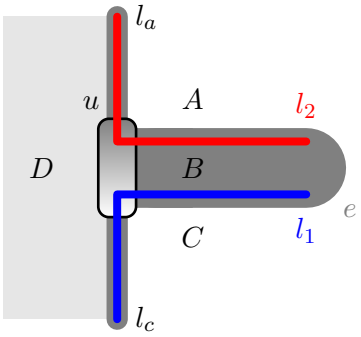
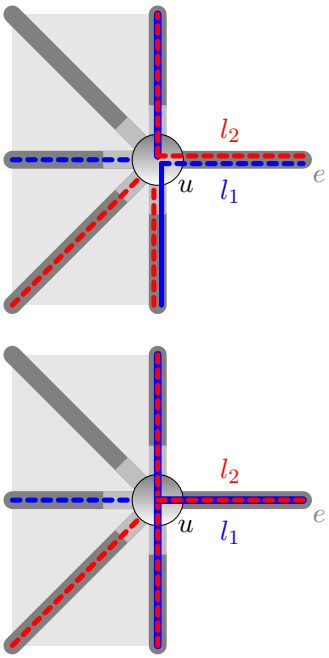


Figure 4.4: Illustration to the proof of proposition 4.3. Case when $l_a = l_2$ and $l_c = l_1$.



(a) The situation during calculating the preferred order. We do not know which line is first anticlockwise from e but we know the clockwise. In this particular situation the preferred order of l_1 and l_2 on e with respect to u is precedes or does not matter. Hence we can say that the preferred order is *weak precedes*.

(b) We do not know which line is clockwise and which clockwise. Hence we do not have *enough information* to decide the preferred order of l_1 and l_2 on e with respect to u.

Figure 4.5: When computing the preferred order of two lines l_1 and l_2 on a common edge e with respect to vertex u there may arise a situation when we do not know which line is clockwise and which anticlockwise. Then we may define terms weak precedes and follows.

But what is important for us is the fact that if there were only lines l_1 and l_2 (as they are drawn directly next to each other and no other line is between them) their number of crossings will be minimal as the vertex preferred order is reflected.

In our program the vertex preferred order does not have to be known yet, because both lines can be found on the same edge and their order does not have to be known at the moment. Hence we say that it is *weak precedes* if it is known to be either precedes or not matter and *weak follows* if it can be follows or not matter. Otherwise we say that we do not have *enough information* or order is *not known*. The figure 4.5 shows this situation.

4.3.2 Preferred order on common subpath

So far we know what is the preferred order of two lines on one endpoint of their common subpath. But there are two endpoints of each subpath. If we compare these preferred orders, we can end up with the following observation. If the preferred order on both

endpoints is the same, then in the optimal solution shall be these lines ordered by this preferred order. We can assume this from the fact that lines cannot cross each other twice on any common subpath.

There is one more thing to mention. If the order was different on whole common subpath or only at one terminal vertex of the common subpath, the lines will not cross twice, but the solution could not be optimal. This is straight consequence of proposition 4.3. For any possible order of all lines, the number of crossings is minimal if the end preferred order of the given pair of lines is respected. By breaking this rule we introduce two new crossings.

Also note that at the endpoints of a common subpath the preferred orders are switched similarly as it is with the $<$ ordering. That is caused by the fact that we change the direction of the edges in this ordering, instead of comparing lines with $<_{(u,v)}$, we compare them using $<_{(v,u)}$. Now we can formally define the preferred order ona common subpath.

Definition 4.4 (Common subpath preferred order) *Let $C = u \cdots v$ be a common subpath of lines l_1 and l_2 with the first and the last edge e and f respectively. Call* e-preferred order *the preferred order* $\mathrm{pord}(l_1, l_2, u, e)$ *of l_1 and l_2 at u with respect to e and* f-preferred order *the preferred order* $\mathrm{pord}(l_2, l_1, v, f)$ *of l_2 and l_1 at v with respect to f. Than we say that the* ***preferred order*** *of lines l_1 and l_2 on the common subpath C*

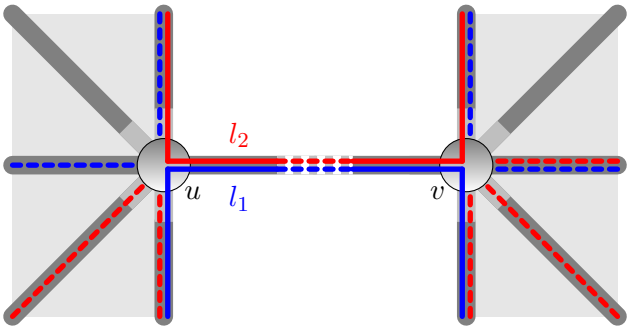
- *is* ***precedes*** *if e and f preferred orders are precedes or one of them is precedes and the other does not matter;*
- *is* ***follows*** *if e and f preferred orders are follows or one of them is follows and the other does not matter;*
- *does* ***not matter*** *if e and f preferred orders do not matter;*
- *and are* ***not comparable*** *if one of e and f preferred orders is follows and the other precedes.*

In cases when both e and f orders are precedes (follows) we say that the preferred order is ***sharp****. When only one of the e and f orders is precedes (follows) and the other does not matter we say that preferred order is* ***fuzzy****.*

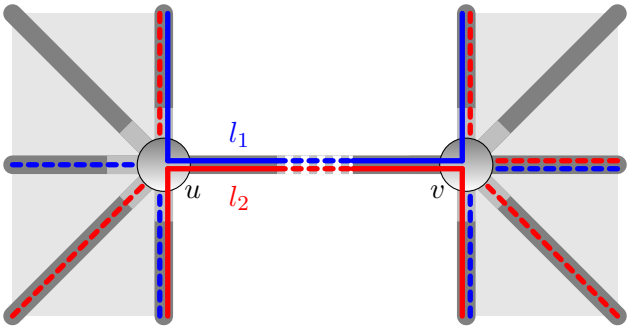
This definition covers all $3^2 = 9$ possible combinations of e and f preferred orders. Hence the common subpath preferred order is defined for every common subpath. Similarly as in section 4.3.1 we can again see that the optimal solution must respect the common subpath preferred order if there are only two lines. Common subpath preferred order is illustrated on figure 4.6.

Proposition 4.5 (Common subpath preferred order is optimal)
Let us have the order of all lines in the MLCMP fixed except the order of lines l_1 and l_2 on a common subpath $C = v_1 v_2 \cdots v_n$. Then the number of crossings between l_1 and l_2 is minimal on $v_1, v_2, \ldots, v_n$ with respect to this order when

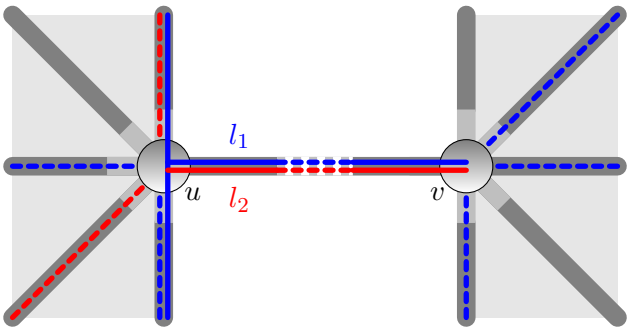
- *$l_1 <^{v_i}_{(v_i, v_{i+1})} l_2$ for all $i \in \{1, 2, \ldots, n-1\}$ if preferred order of lines l_1 and l_2 on C is precedes,*
- *$l_2 <^{v_i}_{(v_i, v_{i+1})} l_1$ for all $i \in \{1, 2, \ldots, n-1\}$ if preferred order of lines l_1 and l_2 on C is follows,*
- *lines l_1 and l_2 do not cross if preferred order of lines l_1 and l_2 on C does not matter,*
- *lines l_1 and l_2 cross one or zero times on C (with respecting the preferred order at v_1 and v_n) if lines l_1 and l_2 are not comparable on C.*



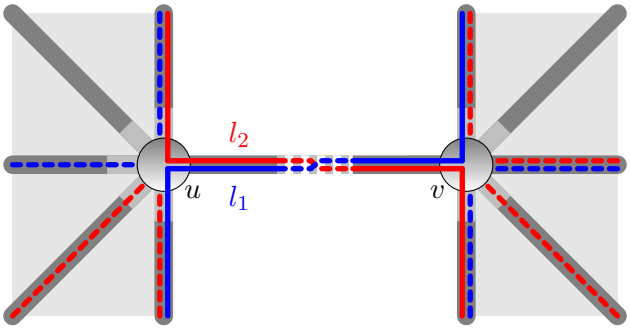
(a) The preferred order is *precedes* as on both ends l_1 tends to be drawn below l_2.



(b) Situation similar to case 4.6a. Role of l_1 and l_2 is only switched. Hence the preferred order of l_1 and l_2 is *follows*.



(c) The Order of lines l_1 and l_2 does *not matter* on C because the order of lines l_1 and l_2 does not matter on both endpoints of C.



(d) Lines l_1 and l_2 are *not comparable* on C. They tend to be drawn on opposite sides of C endpoints u and v.

Figure 4.6: Illustration to the term *common subpath preferred order* defined in 4.4. Similarly as in figure 4.3 it depends only on lines that are closest to the common subpath C. Suppose that the common subpath C is oriented from left to right.

Proof We will refer us to the theorem 4.1 that says that lines do not cross twice on any common subpath. Because the order of all lines is fixed and only lines l_1 and l_2 can be switched on C, we need to show that the number of crossings between l_1 and l_2 is minimal when we respect the preferred order of l_1 and l_2 on C.

Suppose that the preferred order of lines is precedes (or similarly follows) on C. Without loss of generality the preferred order of lines at v_1 is precedes (or follows) and the preferred order of l_2 and l_1 at v_n is not matter or precedes (or follows). Hence the number of crossings is minimal at v_1 and v_n and there is no crossing at vertices $v_2, v_3, \ldots, v_{n-1}$. As a result the number of crossings is minimal from all possible orders of l_1 and l_2 when line l_1 precedes (or follows) line l_2 on whole C.

If the preferred order of lines on C does not matter, then at both endpoints of C the order does not matter. Hence the number of crossings at v_1 and v_n is minimal if we choose any of the orders. At the remaining vertices $v_2, v_3, \ldots, v_{n-1}$ the number of crossings is minimal if l_1 and l_2 do not cross. Hence if we choose the order of l_1 and l_2 arbitrary on whole C, the number of crossings is minimal.

Suppose that the lines l_1 and l_2 are not comparable on the common subpath C. Without loss of generality the preferred order of l_1 and l_2 at v_1 is precedes and $\operatorname{pord}(l_1, l_2, v_n, (v_n, v_{n-1}))$ is follows. If lines l_1 and l_2 do not cross, then exactly one crossing is introduced at v_1 or v_n to the minimal number of crossings at these two vertices. Suppose that the lines cross once at some vertex v_i, where $i \in \{2, 3, \ldots, n-1\}$, and the preferred order at v_1 and v_n is respected, i.e. without loss of generality $l_1 <^{v_1}_{(v_1,v_2)} l_2$ and $l_2 <^{v_{n-1}}_{(v_{n-1},v_n)} l_1$. Then the number of crossings at v_1 and v_n is minimal and there is one crossing introduced at v_i. Two crossings cannot be on C and if the order was not respected at v_1 or v_n there would be at least two crossings. To conclude, the last point in the proposition 4.5 also holds and this finishes the proof. ■

Corollary 4.6 Suppose that the common subpath preferred order of lines l_1 and l_2 is given by the graph topology on all of their common subpaths. Then the minimal number of crossings between l_1 and l_2 among all possible solutions is the same as that one in the solution where the common subpath preferred order is respected.

Proof From proposition 4.5 we know that the number of crossings on each vertex of a common subpath is minimal when the common subpath preferred order is respected. By changing the order on one particular common subpath we cannot influence the preferred order on any other common subpath as we assumed that it is given by graph topology.

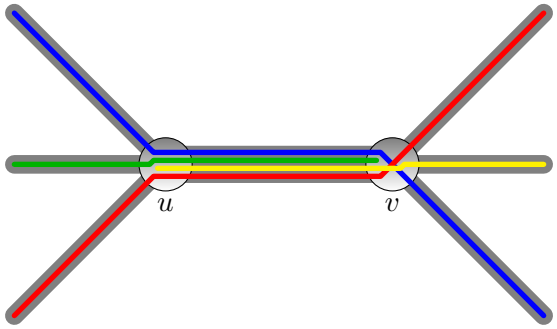
It remains us to prove that there is a solution where the common preferred order of lines l_1 and l_2 is respected. But this solution can be found easily. We may assume that on all common subpaths where the preferred order is not follows that it is precedes (with respect to a fixed direction of the current common subpath). This gives us a solution where are only lines l_1 and l_2 and their common subpath preferred order is respected. Now we can draw the remaining lines arbitrarily. ■

Corollary 4.7 Suppose that the common subpath preferred order is given by the graph topology on all common subpaths. If there exists a solution where the common subpath preferred order of all pairs of lines is respected, this solution is optimal.

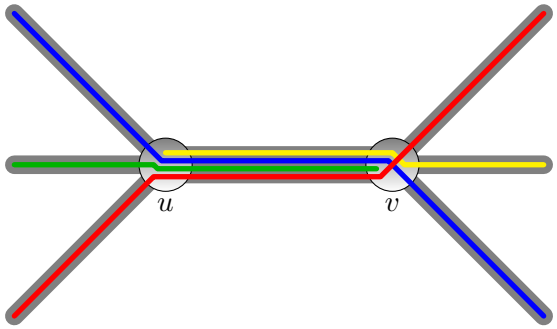
Proof The number of crossings in the whole problem is sum of crossings of each pair of lines. Hence if we find a solution where each of the terms in this sum is minimal, the sum must be minimal as well. Hence we have found the optimal solution. ■

It may seem that if we can find a solution where the common subpath preferred order is respected for a pair of lines then we can find a solution where the preferred order is

respected for any number of lines. This would be useful and it would simplify everything but it is false. Figure 4.7 shows a counter example. If this hypothesis held then in the path lines case (discusses in chapter 5) we would be able to find the optimal number of crossings in the graph. But as this does not need to hold always, we are able to do it only in the periphery assignment case.



(a) If preferred order of green, blue and red lines is respected, then blue and red line must cross at v. If we repeat this thought for red, blue and yellow line, we see that blue and red line cross at u. Hence we are not able to find solution with single crossing.



(b) One of the optimal solutions to the situation in figure 4.7a. We see that there is introduced one crossing between one pair of lines where the preferred order is fuzzy.

Figure 4.7: Counterexample to the hypothesis that we can always find a drawing of lines such that common subpath preferred order is respected.

Again as in the previous section the preferred order of lines at the endpoints does not need to be known at the moment. If we do not have enough information for one of the endpoints of C, we do not have *enough information* on whole common subpath C. But there might be the weak preference at the endpoints of C.

If on one endpoint of C is the preferred order weak precedes and on the other precedes, then the preferred order on whole common subpath can only be precedes. The same holds for weak follows and follows. If both ends have the same weak preference, assume that weak precedes, then the preferred order can be either precedes or not matter. Hence we can assign the line order to be precedes on whole common subpath C. The only thing that must be checked is whether the given solution is still admissible.

Finally if both e and f preferred order from definition 4.4 are weak and different, i.e. one endpoint has weak precedes and other has weak follows preference, we have to say that we do not have enough information on C. The final order can be precedes, follows, not matter as well lines might be not comparable. An example of the case when we do not have enough information is on figure 4.8.

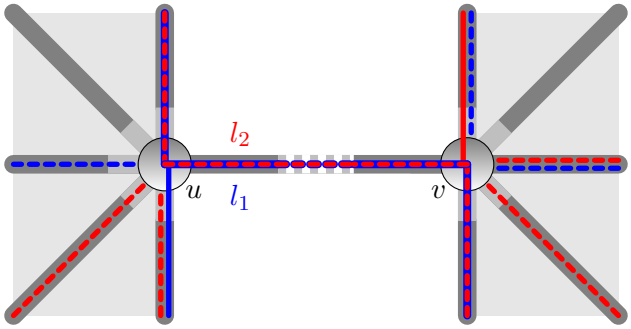


Figure 4.8: We do not have *enough information* to decide what is the common subpath preferred order on this path. It may be not matter, not comparable, precedes as well as follows.

4.3.3 Summing it up

The corollary 4.7 of proposition 4.5 gives us a strong tool for deciding the order of lines in the Metro-Line Crossing Minimization Problem. The common subpath can be clearly found in polynomial time, sorting of edges and lines is also polynomial. Afterwards deciding the preferred order of lines can be done in constant time. Hence for many pairs of lines in the graph it might be possible to know the preferred order on some of their common subpaths. This knowledge may be based only on the graph topology.

In our program we can remember the order $<$ of each pair of lines that go through the given edge. Once the order is known, it can be written to a data structure. There will be written all consequences of this order arising from the transitivity of the $<$ order. Once the order is known entirely, it cannot be changed and trying to overwrite it with an inadmissible order will tell us that the common subpath preferred order cannot be respected.

We also know that if the common subpath preferred order of lines l_1 and l_2 is known from the topology of the graph, in an optimal solution the preferred order is respected (if it is possible). Hence we can try to calculate this preferred order and then decide whether it is admissible or not.

4.3.4 Another greedy approach

There can be also found more greedy rules. For example suppose that we have a path $C = v_1 v_2 \cdots v_n$ such that for all edges $e_i = (v_i, v_{i+1})$, where $i \in \{1, 2, \ldots, n-1\}$, L_{e_i} contains both lines l_1 and l_2. Moreover suppose that all vertices $v_2, v_3, \ldots, v_{n-1}$ are inline vertices of the line l_1. Call this path the extended common subpath of lines l_1 and l_2 respectively. If we know the preferred order of lines on both endpoints of this common subpath then we can construct a set of rules that allows us to order these two lines on the edges e_i. Again we must be sure that assuming the order will not lead to an inadmissible state.

We will present these rules informally without any proof. But everyone can check that no redundant crossings will be introduced by following them. The conditions under which the solution will be admissible are same as those that we will present in section 4.5. At this time we will leave them unsaid and we can suppose that the lines are alone on their common subpath C.

Suppose that you already know the order of l_1 and l_2 on an edge e_i and you try to decide the order of those lines on e_{i+1}. In this situation we will call l_1 side the side of path C on that l_1 lies on e_i and l_2 side the other. Then we will say that there is an edge on the same site if there is an edge from v_{i+1} on l_1 side. This means exactly what it

looks like. Continuing e_i in direction of C we encounter an edge at the endpoint of e_i. Similarly we say that there will be edge at next vertex on same (other) side if there is an edge from v_{i+2} on l_1 (l_2) side. We will not introduce any redundant crossing on e_{i+1} if we apply the first of these rules that holds.

- The order of lines does not matter if there is an edge on the same side and there will be edges on both sides at the next vertex.
- If there is not an edge on the same side, continue with the current order.
- If there is an edge on the same side, but there will not be at the next vertex, continue on the same side.
- If there is an edge on the same side, and there will be on the same side on the next vertex as well, switch the order of lines l_1 and l_2 at e_{i+1}.

4.4 Dependecy graph

In section 4.3.2 we stated the preferred order of two lines on their common subpath. The rest of the problem will be about solving the common subpaths whose order is not known and deciding where to cross lines that are not comparable. But we may try to classify some of subpaths on that we do not have enough information yet.

Situation when the preferred order is not known arises when we do not know the order of lines l_1 and l_2 at some edge e' that is incident to an endpoint v of a common subpath C of these two lines. This edge e' must be the first edge incident to v in clockwise (or anticlockwise) direction from C that contains both lines l_1 and l_2. Denote the common subpath of lines l_1 and l_2, starting from e', as C'. Then we say that the common subpath C *depends* on the common subpath C'.

It may seem that this dependency is a symmetric relation but the opposite is true. If we do not know the order at one endpoint, the preferred order might be weak precedes (or follows). Afterwards we can decide under some conditions that the preferred order on C' is precedes. Hence C depends on C', but not vice versa. If our greedy algorithm encountered first C' and after that C, then the order on C might be resolved. But if it encountered C after C' it could not be solved. Figure 4.9 shows this situation.

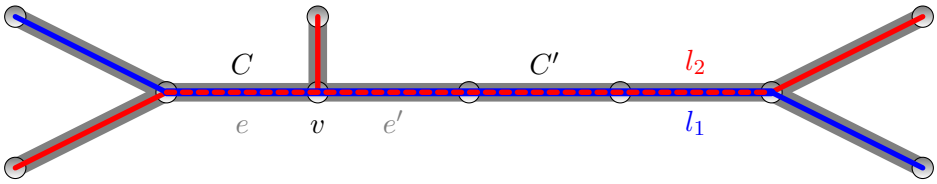


Figure 4.9: An example of a dependency of the common subpath preferred order. If we first encounter the common subpath C, we cannot decide what is the preferred order of l_1 and l_2 on this path as order of lines may not matter or lines might be non comparable. But if we resolved firstly the common subpath C', we see that the red line l_2 shall be drawn above the blue l_1 regardless of their order on C. After that we see that lines l_1 and l_2 are not comparable on C.

This may lead us to a simple solution based on idea of the common subpath dependency. If we run our greedy algorithm again, it will know for sure the order of C' and hence now it can resolve the preferred order of lines on C. The previous situation may arise again. So we need to run the algorithm again and again until nothing changes.

Such an approach will certainly work and allow us to decide preferred order of more lines than just one run of the program. However multiple restarts will be likely very ineffective. For each common subpath, whose order is not yet known, we can find out on which common subpaths it depends. For each pair of lines we can construct such a dependency graph. In this graph we can try to find sink vertices,[2] that do not depend on any other common subpath and resolve them first. If the graph does not contain a cycle we can construct a topological ordering of this dependency graph and resolve the subpaths in the given order. This approach will be more effective than relying on simple restarts of the greedy algorithm.

4.5 Deciding on not matter and not comparable common subpaths

Before searching all common subpaths by an exponential algorithm it might be useful to decide the order of some common subpaths with preferred order that does not matter or with not comparable lines. This can be done only if we know that we will not sacrifice the optimal solution (unless we did it already by assuming that there is a solution that respects the preferred order) and we will not do something that will lead us to an inadmissible solution.

Let us have the common subpath $C = v_1v_2 \cdots v_n$ of lines l_1 and l_2. In order to know that the order of lines does not matter or that l_1 and l_2 are not comparable on C we need to know the preferred order of lines l_1 and l_2 at both endpoints v_1 and v_2. If we did not know that, we will not have enough information on C.

Imagine that C is drawn horizontally. Some problem may arise when there is a line between lines l_1 and l_2 or if there is a line about that we do not know if it lies above or below l_1 or l_2. This can be formally expressed in proposition 4.9.

Definition 4.8 (Bundle common subpath) *Let $C = v_1v_2 \cdots v_n$ be a common subpath of lines l_1 and l_2. Then we say that C is a **bundle** if and only if for all edges $e_i = (v_i, v_{i+i})$ ($i \in \{1, 2, \ldots, n-1\}$) holds*

$$\forall l \in L_e \setminus \{l_1, l_2\} : ((l <_e^{v_i} l_1 \Rightarrow l <_e^{v_i} l_2) \wedge (l_1 <_e^{v_i} l \Rightarrow l_2 <_e^{v_i} l)) . \tag{4.1}$$

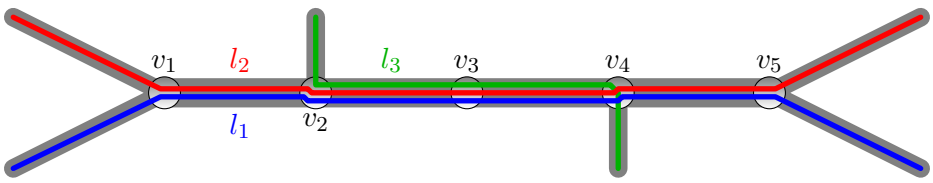


Figure 4.10: Illustration to the term *bundle common subpath*. The common subpath $v_1v_2v_3v_4v_5$ of l_1 and l_2 is a bundle as there is no line drawn between them. Same holds for the common subpath $v_2v_3v_4$ of lines l_2 and l_3. On the opposite $v_2v_3v_4$ is not a bundle common subpath of l_1 and l_3 as $l_1 <^{v_2}_{(v_2,v_3)} l_2$, but $l_2 <^{v_2}_{(v_2,v_3)} l_3$.

Proposition 4.9 *Let C be a bundle common subpath of lines l_1 and l_2 that are not comparable or their order does not matter. Suppose that l_1 and l_2 do not cross each other on C. Fix the order of all lines except l_1 and l_2 on C. Then the number of crossings among all vertices will remain same if we switch the drawings of l_1 and l_2 on C.*

[2] By a *sink vertex* we mean a vertex with outdegree zero.

If lines l_1 and l_2 are not comparable then the number of crossings will be same in those two possibilities as well as if they cross once and the preferred order at C endpoints is respected.

Proof Denote vertices of C as $v_1, v_2, \ldots, v_n$. Condition (4.1) tells us that the lines that are drawn right (or left) of l_1 are drawn right (or left) of l_2 on each edge $e_i = (v_i, v_{i+i})$ $(i \in \{1, 2, \ldots, n-1\})$. This means that the lines l_1 and l_2 are drawn directly next to each other. Consider any line l that cross l_1 at v_i m-times. As l cannot be drawn between l_1 and l_2 on C edges, the number of sectors with l edges (namely $m+1$), given by dividing the plane with l_1 edges, is same as if we split the plane with l_2 edges of C. Same holds for splitting plane with one l_1 edge and one l_2 edge. Hence the number of crossings between l and l_1 will remain m if we switch arbitrary l_1 and l_2 on C edges.

Now we reduced the proof to showing that the sum of the number of crossings of l_1 and l_2 around C vertices will remain the same. We may refer us to the proposition 4.5 from that follows that the number of crossings between l_1 and l_2 is the same as both possibilities (or n possibilities in case of non comparable l_1 and l_2) lead to the optimal number of crossings between l_1 and l_2. ■

Corollary 4.10 Suppose that the preferred order of lines l_1 and l_2 on C is known to be not matter or not comparable. If C is known to be a bundle due to the preferred order, we may assume that

$$l_1 <^{v_i}_{(v_i, v_{i+1})} l_2$$

for all $i \in \{1, 2, \ldots, n-1\}$, where $C = v_1 v_2 \cdots v_n$.

Proof If we know that the lines l_1 and l_2 are a bundle, we must know the order of all pairs of lines on each edge of C to be precedes or follows except l_1 and l_2. Lines l_1 and l_2 must be neighbors. It can be easily seen that assuming the order will lead to an admissible solution. Moreover this solution will be still within the optimal solutions (if the original one was) as a result of proposition 4.5 and proposition 4.9. ■

Now we know where we can decide the order of lines almost arbitrary. If it is done then the problem complexity decreases dramatically. There remains only some edges whose order must be found by searching the state space. This is content of the next section.

4.6 Searching the state space

All parts of the program already shown are polynomial. But now we need to search all possible remaining orders of lines that have not been decided yet. In order to minimize the number of encountered inadmissible solutions, it seems to be a good idea to decide the order of lines on a whole common subpath instead of just single edge.

There are two possible orders for a common subpath with an order that does not matter. If lines are not comparable, then there are n possibilities, where n is the number of vertices of the given common subpath. And the worst case is when we do not have enough information. For such a common subpath there are $2n-2$ possible orders. This follows from the proposition 4.5. If lines do not cross, there are two possibilities. If they cross once with the given order at endpoints, there is one possibility for each vertex of the common subpath except the terminals of the path.

Suppose that we have successfully assigned the order on some common subpaths of lines by our greedy algorithm. Now our *search space* is a set of all valid assignments of the lines order such that they are candidates to an assignment with the minimal number

of crossings among all stations. Hence from the search space are omitted the assignments where lines cross twice on a common subpath, the assignments where lines with not matter order cross each other and similar situations that contradict what we know as far due to propositions 4.1 and 4.5.

4.6.1 Splitting the search space

If we tried to assign all common subpaths in one moment, our program will not find the desired solution in plausible time. For a common subpath of length n there are 2, n or $2n-2$ possibilities. The size of the search space is then a product of all these possibilities. But many of them are inadmissible or they are duplicate.

If the underlying graph is not connected, the minimum can be found independently on each connected component and the result is given only be merging these optima. Now call the order $<$ on an edge e *definite* iff the preferred order of all pairs of lines in L_e is know to be precedes or follows on their common subpaths that contain e. Otherwise we say that e is currently *indefinite*. If we leave out indefinite edges from the graph, it may split into connected components. Following proposition tells us that we can solve these components independently.

Just remind from section 4.4 that the preferred order on C depends on the preferred order on C' iff we do not have enough information on C and after assuming the order on C' (and other common subpaths if needed) we can calculate the common subpath preferred order on C. This relation we will call *direct dependency*. *Indirect dependency* is transitive closure of the direct dependency relation.

Proposition 4.11 *Preferred line order on a common subpath C may depend directly only on the order of lines on indefinite edges incident to the endpoints of C.*

Proof The proof is done by simple contradiction. Suppose that the order of lines l_1 and l_2 on their common subpath C depends on order on an edge e' that is not definite. If e' is not incident to C endpoints, preferred order of C may not directly depend on it. If the order of lines l_1 and l_2 on e' is given, it can be seen which line is closer to C in (anti)clockwise direction. Hence the preferred order on C cannot depend on e'. ■

Corollary 4.12 Preferred order on a common subpath C may depend indirectly on the preferred order on another common subpath C' only if there is a path from a C vertex to a C' vertex that contains only indefinite edges.

Proof Suppose that C indirectly depends on C' and all paths between C and C' vertices contain at least one definite edge. Proposition 4.11 follows that C and C' are not weakly connected in the directed graph of these dependencies. But that is contradiction with the assumption that C indirectly depends on C'. Hence there must be at least one path with all edges indefinite. ■

We can utilize this corollary as follows. Suppose we split the problem into *isolated components* of indefinite tracks, such that these components are connected. Consider two of those components A and B. If we found the optimum on A it will not affect any solution of B. All assignments of B that were admissible before assigning A are still admissible. Hence we can find the local optima of A and B independently as well as the optima of all similar isolated components in the graph. The final solution will be given by pure merging of these local optima. We will use the term isolated components in next sections.

4.6.2 Recursive structure

Now we are given an isolated component and our task is to find the local optimum on this component. We may try to iterate over all possible orders of all common subpaths whose preferred order is not precedes nor follows. But many of those solutions will not be admissible as there can be created cycles in $<$ order on some edge.

Our idea may be based on the following observation. If there is a set of definite edges that splits the isolated component in two, we may solve these components independently and merge these optima. This is an approach typical for the recursive algorithms based on the *divide an conquer* paradigm. So suppose that we have chosen a set T of edges that splits the isolated component into two. None of those edges has a definite order at the moment.

Let us denote U the set of all common subpaths of an arbitrary pair of lines that contains any edge of T. If we want the edges of T to be definite, we have to set the order on each edge of all subpaths in U. Hence we need to iterate the product of 2 for each not matter common subpath, n for each not comparable common subpath and $2n - 2$ for each common subpath on that we do not have enough information.

After this is done, we may run the greedy part of our algorithm on both sub components. Hence we can utilize the information given by the current assignment of orders on T tracks. Now we can again split these components recursively until we get a trivial case.

4.6.3 How to speedup the recursion

The dummy selection of the edges set T that splits the isolated component may work as well, but we may try to do better. We may to choose tracks in T such that there is a minimal number of possible valid assignments. This corresponds to the *minimum remaining values* heuristic. Purpose of this selection is that we want to minimize the branching of the recursive search.

On the opposite we want to have the isolated components divided into two sub components of approximately same size. This will minimize the depth of recursive calls.

In our algorithm, there is a compromise of these two contradictory approaches. We choose the edges from T such that they form a minimum cut of the current isolated component. But the minimum cut does not need to be the only one in the graph. Hence from those we choose that one with the minimal number of common subpaths (that do not have the preferred order precedes or follows) that cross edges in T. In case of both characteristics being equal we take that one that divides the problem more uniformly.

We do not need to be sure for 100 % that the cut we found is really a minimal cut, so we may use a faster heuristic named *Karger's algorithm*. It is a randomized algorithm that contracts the edges until there remain only two vertices. The remaining edges (trivial cycles are removed during the run) represent the given cut. It can be shown that the found cut is a minimal cut with probability big enough that several restarts of the algorithm will probably find the desired minimal cut. More information about this algorithm can be found in [Karger, 1993].

Due to the transitivity of the order $<$ we are given a simple *forward checking*. When the cycle in the $<$ order is created, the program may skip all assignments that contain this invalid order. We do not need to assign it again and find out that it still leads to an error.

In order to be able to iterate all these possible assignments we need an ordering structure that allows us backtracking. But copying the whole order of all lines on all edges will be a little bit expensive. Hence we use a *copy-on-write* structure that remembers

only the changes to the order and in case of backtracking it forgets only corresponding changes.

In the Metro-Line Crossing Minimization Problem we have to find the optimum, not only a valid assignment. Hence some of the optimizations commonly used in the Constraint satisfaction problems will not work. For example the assignments are usually sorted in the probability order, but we need to search them all. Hence the running time does not depend on the order of the possible assignments.

4.7 Pseudocode

Now we have described all parts of our algorithm so far. Hence it is time to present the pseudocode. We split the code into two parts, the greedy part and the exponential part where we search the state space. The algorithm 1 shows the pseudocode.

```
function SolveMLCMP(problem)
    ContractEdges(problem)                                              ▷ 4.2.1
    ordering ← EmptyOrdering(problem)
    if Failed(GreedyPart(problem, ordering)) then
        AssignNotEnoughInformation(problem, ordering)
    end if

    for each subproblem in IsolatedComponents(problem, ordering) do     ▷ 4.6.1
        SearchStateSpace(problem, ordering)
    end for
    return ordering
end function

function GreedyPart(problem, ordering)                                  ▷ 4.3
    if FailedOnWholeProblem() then return
    AssignPreferredOrderOnCommonSubpaths(problem, ordering)             ▷ 4.3.2
    ResolveDependencyGraph(problem, ordering)                           ▷ 4.4
    DecideOnBundleSubpathsIfPossible(problem, ordering)                 ▷ 4.5
end function

function SearchStateSpace(problem, ordering)                            ▷ 4.6
    cut ← FindMincut(problem)                                           ▷ 4.6.3
    (subproblem1, subproblem2) ← SplitProblem(problem, cut)             ▷ 4.6.1
    found ← ∞

    for each assignment in ValidAssignments(problem, order, cut) do     ▷ 4.6
        NewBreakpoint(ordering)
        Assign(assignment, order, cut)

        GreedyPart(subproblem1)
        SearchStateSpace(subproblem1, ordering)
        GreedyPart(subproblem2)
        SearchStateSpace(subproblem2, ordering)

        if CurrentNumberOfCrossings(problem, ordering) < found then
            found ← CurrentNumberOfCrossings(problem, ordering)
            optimum ← assignment
        end if
        UndoBreakpoint(ordering)
    end for

    Assign(ordering, optimum)
end function
```

Algorithm 1: The pseudocode of the algorithm

Chapter 5 Discussion of path lines case

Now we will spend some more time discussing the original problem introduced by [Benkert et al., 2007] and then solved by the other authors. It consisted of an underlying graph and a set of lines on that graph. Lines were only single paths. But a condition of *admissibility* of all vertices was added. From all possible solutions was chosen that one with the minimal number of crossings that had admissible vertices.

The condition of admissibility was simple. Lines l_1 and l_2 were not allowed to change their relative order at any vertex of their common subpath, including both endpoints. Hence the lines were allowed to cross only at vertices that are not part of any of their common subpaths and at edges of common subpaths. The solutions of our problem and the original are equivalent except some marginal cases.

There were several reasons why we did not consider the admissibility earlier. At first we supposed that the lines were general graphs. If there were three or more edges of one line incident with a vertex v, we could not talk about a relative order of that line with others. Moreover our motivation was to draw a tourist trail map with minimized number of crossings. Tourist trails map would look a little bit weird if the lines crossed each other somewhere in the middle of a track instead of at a crossroad.

5.1 Common subpath preferred order

The admissibility of many common subpaths was ensured by the common subpath preferred order as we defined it in section 4.3.2. The following proposition shows some interesting facts about the common subpath preferred order when the lines are paths not matter whether the admissibility condition is required or not.

Proposition 5.1 *Suppose that all lines in L are simple paths. Then the common subpath preferred order is always known to be precedes, follows, not matter or not comparable.*

Proof Suppose any pair of lines l_1 and l_2 and their common subpath C on that we do not have enough information. Choose one of C terminals on that we do not have enough information and call it v. Denote the corresponding C edge e. Vertex v cannot be a terminal station of l_1 nor l_2 as the order will not matter otherwise.

Without loss of generality we can suppose that the nearest edge with l_1 or l_2 in clockwise direction from e is an edge incident with v on that we do not have enough information. Denote it e'. If there is in the anticlockwise direction any other edge that contains l_1 or l_2 it contradicts that l_1 and l_2 are paths. Hence the first edge that contains one of lines l_1 or l_2 in the anticlockwise direction from e must be again e'. This is a contradiction to the fact that v was a terminal of the common subpath C. Hence we must know what the preferred order of lines on C is. ■

Corollary 5.2 We can find a nontrivial lower bound of the number of crossings in a Metro-Line Crossing Minimization Problem instance with path lines in polynomial time.

Proof From proposition 5.1 we know that there is no path on that it is not enough information. Corollary 4.6 tells us that in an optimal solution lines cannot have smaller number of crossings than in that solution where the common subpath preferred order is respected. Hence we know that on all common subpaths with preferred order precedes, follows or order that does not matter lines may not cross. Lines that are not comparable cross each other exactly once on the common subpath or at common subpath ends. We can calculate the lower bound for each common subpath and then we sum all these lower bounds. Finding the common subpath is polynomial for all pairs of lines and there exists a polynomial number of pairs of lines. The common subpath preferred order can be decided in polynomial time too. Hence we are able to calculate this lower bound in polynomial time. ■

After getting this lower bound, we do not need to search the whole state space, but we can search it only until we get some solution that reaches the calculated lower bound. In the problem as we presented it before this lower bound does not have to be sharp. We have shown such an example in figure 4.7. But in many of possible inputs it will be sharp. This can speedup our algorithm if we know the preferred order of all common subpaths in the current isolated component. If not we can get a weaker lower bound by assuming the following. If we do not have enough information on a common subpath, lines cross at least zero times. This may be sometimes useful as well.

This lower bound would hold for the Metro-Line Crossing Minimization Problem with the vertex admissibility condition as well. But proofs in section 4.3.2 required that we may cross lines at all vertices. Hence it is more likely that this bound is not sharp. We can modify the problem by adding dummy vertices into the edge midpoints, in order to get solutions where lines cross at edges. But by forbidding crossings at original vertices, we restrict the solutions so that the current optimal number of crossings will no longer be reachable. Figure 5.1 illustrates this problem. Hence if we allow only edge crossings then we may introduce some new avoidable (in meaning of common subpath preferred order) crossings into the graph. Our recursive algorithm for searching the state space will work, but sometimes it can return the answer that the optimum does not exist when it actually exists. Then it needs to be run again without the greedy part. On the opposite situations like that one shown in figure 5.1 can be detected.

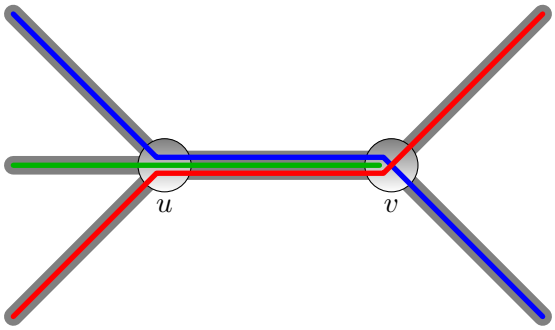


Figure 5.1: Figure that illustrates the fact that if we allow crossings of lines only on edges, we may introduce new avoidable crossings. As the blue and red line cannot cross at v, the green line must cross one of the others.

5.2 Metro-Line Crossing Minimization Problem-FixedSE

Despite [Nöllenburg, 2010] presented a polynomial algorithm for MLCMP-FixedSE, we will spend some moments on this problem variant. We will show that our theory is consistent with that in [Nöllenburg, 2010]. Note that theorem 4.1 holds even if we limit lines to cross only on edges. The proof is almost the same, or reader may look into [Asquith et al., 2008].

Return to the definition 4.2 of the vertex preferred order. We supposed that the preferred order always does not matter if the corresponding vertex was an endpoint of one of the lines. But in the Metro-Line Crossing Minimization Problem-FixedSE we may know the order of those two lines as it is a part of the input. We will loose some information if we assume that the order does not matter. We will redefine this preferred order on MLCMP-FixedSE to be as follows.

Definition 5.3 (Vertex preferred order in the MLCMP-FixedSE)
Let us consider two lines l_1 and l_2 and their common edge $e = (u, v)$ in the MLCMP-FixedSE. If u is a terminal station for l_1 or l_2 we define the preferred order of l_1 and l_2 at u with respect to e to

- *be **precedes** if l_1 is drawn right of l_2 at the point u with respect to direction of edge $e = (u, v)$ due to the terminus side assignments,*
- *be **follows** if l_1 is drawn left of l_2 at the point u with respect to direction of edge $e = (u, v)$ due to the terminus side assignments,*
- ***does not matter** if l_1 and l_2 both end at u and they terminate in the same station end.*

There are no other possibilities for lines. The order is know due to the terminus side assignment or both lines must end at the same station end. Examples of this term are shown on figure 5.2.

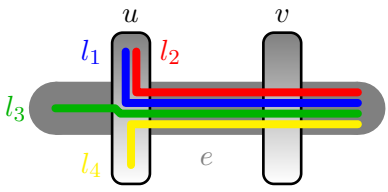


Figure 5.2: An illustration to the *vertex preferred order* definition 5.3 in the case of the MLCMP-FixexSE. The vertex preferred order of l_3 and l_1 (or l_2) at u on e is precedes as l_3 is drawn right of l_1 (or l_2). Similarly l_4 precedes l_1, l_2 and l_3 or vice versa l_1, l_2 and l_3 follow l_4. The preferred order of l_1 and l_2 does not matter as l_1 and l_2 terminate both on same station end of u.

Proposition 5.4 *Any MLCMP-FixedSE instance can be transformed into a MLCMP-FixedSE instance where are only common subpaths with the preferred order precedes, follows or not comparable. Moreover this order is always known to be one of those three.*

Proof The order is always known as we proved in proposition 5.1. It remains to prove that there is no common subpath with preferred order that does not matter unless one of the lines can be omitted. Consider one common subpath C with order that does not matter. Denote the corresponding lines l_1 and l_2. Then the preferred order of l_1 and l_2 on both its endpoints must be the one that does not matter.

When l_1 and l_2 order is known to be not matter at vertex u with respect to a common edge e, then l_1 and l_2 must terminate at u at same station end. This is a straight consequence of definition 5.3. If this holds for both C endpoints then l_1 and l_2 represent exactly the same path on the same set of tracks. Moreover they end in the same station ends. Their preferred order to all other lines on all possible common subpaths is equal. Lines that cross l_1 must cross l_2 and vice versa. Hence we may assume that lines l_1 and l_2 are a bundle, i.e. they are drawn next to each other. We can omit one of these lines (suppose l_2) and in final solution split the other (l_1) into two lines l_1 and l_2 that are drawn side by side with no crossings. ■

Corollary 5.5 We may assume that there is no common subpath on that the preferred order of lines does not matter.

Proof The corollary is a straightforward result of the proposition 5.4. Instead of the original problem we may consider an equivalent one. ■

Corollary 5.6 The vertex preferred order does not matter only for a pair of lines that terminate in the same station end of the same station. The preferred order on the other common subpath end is precedes or follows.

Proof We have already proven that if the lines preferred order on a common subpath end does not matter then lines terminate at the same station and at the same end of this station.

Now consider the other endpoint of an incident common subpath of these two lines. If the preferred order did not matter, the preferred order on the whole common subpath would not matter. That contradicts 5.5. Hence the preferred order on the second common subpath endpoint must be precedes or follows. ■

Yet we are ready to prove a theorem presented in [Nöllenburg, 2010]. There is defined the crossing between lines l_1 and l_2 on their common subpath C to be *unavoidable* if lines l_1 and l_2 cross each other on C in any possible solution of the problem. Our goal is not to present some "new" or "better" proof but to show that our theory leads to the same results and is consistent with other authors. Just remind that two lines are allowed to cross only on the edges of common subpaths or vertices that are not part of any common subpath of those lines.

Theorem 5.7 (Nöllenburg, 2010) *Let $G = (V, E)$ be an underlying graph and L a set of lines on that graph. Suppose that a terminus side assignment for all lines in L is given. Then there are only unavoidable crossings in any optimal solution of the Metro-Line Crossing Minimization Problem-FixedSE.*

Proof Consider a common subpath C of lines l_1 and l_2. There are three possible preferred orders on this common subpath as a result of proposition 5.4 and its corollary 5.5

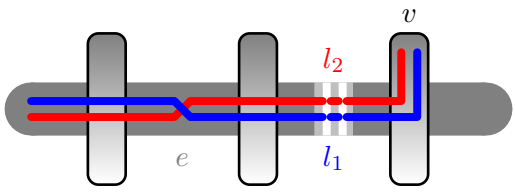
1. If lines l_1 and l_2 are not comparable on C then they have to cross once, but not twice or more times as a result of theorem 4.1. Hence all crossings on not comparable common subpaths are unavoidable.

2. If the lines preferred order of l_1 and l_2 on C is precedes, then there are two possibilities for the vertex preferred orders on the ends of C.

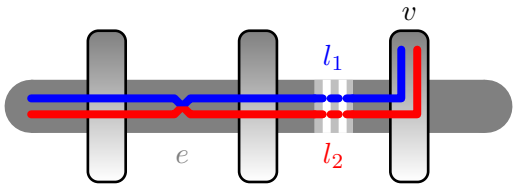
    - Suppose that the preferred order of l_1 and l_2 at the first vertex of C with respect to the first edge of C is precedes and preferred order of l_2 and l_1 at the last C vertex with respect to the last C edge is precedes as well. This

is a case of a sharp precedes order. Then lines l_1 and l_2 can cross only even number times on C edges. As two or more crossings are not possible in an optimal solution, lines l_1 and l_2 will not cross in any optimal solution. Hence no avoidable crossing can be introduced.

- Suppose without loss of generality that the preferred order of l_1 and l_2 at the first vertex of C with respect to the first edge of C is precedes and the preferred order of l_1 and l_2 at the last C vertex with respect to the last C edge does not matter. This is a case of a fuzzy precedes order. Denote the last vertex of the common subpath C as v. Lines l_1 and l_2 cannot cross twice. Suppose that they cross once on an edge e in an optimal solution S. Then we can switch l_1 and l_2 drawings on the path between their crossing as the vertex v because v is a terminal of both lines. Denote this solution S'. The total number of crossings between all lines remained the same except one crossing of l_1 and l_2 at e. Lines that crossed l_1 on path from e to v now cross l_2 and vice versa. Anywhere else nothing changed. Hence the number of crossings in S' is one less than in S what contradicts to the optimality of S. Hence l_1 and l_2 do not cross on C and there is no avoidable crossing in the optimal solution.



(a) The solution S with redundant crossing between l_1 and l_2.



(b) The solution S' where this crossing is removed.

Figure 5.3: An illustration to the exchange argument in the proof of theorem 5.7. As the preferred order of two lines does not matter on one endpoint and they cross, we can switch their drawings and find a solution with smaller number of crossings.

In this case there cannot be no avoidable crossing on C in an optimal solution.

3. Last case when the lines preferred order is follows is similar to the case 2. The proof is the same except that we need to use the word follows instead of precedes.

We see that in all three possible cases there are no avoidable crossings which completes the proof. ■

The proof may look like that it holds even for situation when the terminus side assignment is not given. But in figure 5.1 we saw an counter example. Reading again the proof we see that the only part that used the periphery assignment condition was that one when we discussed the fuzzy common subpath. From not matter order we deduced that lines l_1 and l_2 both terminate at v.

Now what is wrong in case without a periphery assignment? One of the lines (without loss of generality suppose that l_1) ended at v and the other may have continued. Now

we could not have switched the l_1 and l_2 drawings as v is a terminal for l_1 and an inline vertex for l_2. Our exchange argument cannot work and we are not able to find a better solution than S. So S may be optimal. Also we see that this case is the only where the Metro-Line Crossing Minimization Problem-FixedSE with and without the terminus side assignment differ.

5.3 Path lines terminating in leaves

Now return to the original problem that we solved in chapter 4. We will show that we are able to solve it in polynomial time if lines are just single paths that terminate in leaves of degree one with respect to the underlying graph $G = (V, E)$. This is a problem variant called Metro-Line Crossing Minimization Problem-T1.

In [Asquith et al., 2008] is proved that the Metro-Line Crossing Minimization Problem-FixedSE is polynomial time solvable. [Nöllenburg, 2010] presents that the MLCMP-T1 and the MLCMP-FixedSE are equivalent as we can can replace the station ends by dummy edges or vice versa the edges with terminals with the stations ends. Similarly as in section 5.2 we will present a new proof of the fact that the MLCMP-T1 can be solved in polynomial time. This proof is the next example of consistency of our theory and the previous one as both lead to the same results. The following proposition holds even if we allow lines to cross at vertices (or not). The proof is for both variants same.

Proposition 5.8 *Given a MLCMP instance where all lines are simple paths, that terminate in vertices of degree one, we can find a solution in polynomial time.*

Proof Due to proposition 5.1 and due to corollary 5.2 we know a lower bound of crossings. We will show that we can find a solution that reaches this lower bound in polynomial time.

Pick any arbitrary solution S_0 with a random order of lines on their common edges. If there are two lines l_1 and l_2 that cross twice on their common subpath C switch their drawings between any two crossings on C (as we did in the proof of theorem 4.1). By this exchange we did not introduce any new crossing between l_1 (or l_2) and any other line as lines that crossed l_1 cross now l_2 and vice versa. The number of crossings between l_1 and l_2 has decreased by two. Repeat this until we get a solution S_1 where each pair of lines cross at most once on any common subpath. This repetition will end because the number of crossings in the graph after each step strictly decreases. Such a sequence of natural numbers must be finite.

For each common subpath $C = v_0v_1 \cdots v_n$ of lines l_1 and l_2 with endpoints e and f take the e and f preferred order from definition 4.4. Now for C do following.

1. If lines on C are not comparable, we already reached the optimum of crossings on C. Lines that are not comparable have to cross at least once and we found a solution where they cross no more than once.

2. If both e and f preferred orders are precedes (or similarly both are follows), lines cannot cross odd number of times as the graph is planar.[1] This follows that in S_1 lines on such common subpaths do not cross.

[1]This fact can be clearly seen. Assume without loss of generality that C is horizontal and line l_1 lies below l_2 at v_0 and v_n. If they crossed once then l_2 must be drawn below l_1 right (without loss of generality) from this crossing. But this contradicts that l_1 is below l_2 at v_n. If l_1 and l_2 crossed odd number of times, we could find a drawing where l_1 and l_2 cross once by switching their drawings between a pair of crossings. But we have already shown that this is impossible.

3. Suppose that one of e and f preferred orders is precedes (or similarly follows) and the other is not matter. This is a case of a fuzzy preferred order. If lines do not cross, we cannot do better. If they cross once we can change their drawings between the crossing and the not matter end of C. This can be done same as in the proof of theorem 5.7. Suppose without loss of generality that it is v_n. Then both lines l_1 and l_2 terminate at v_n. If this was false then one of the lines terminated in an inline vertex which contradicts the fact that we have a MLCMP-T1 instance. Exchanging the lines we did not introduce any new crossing and lines l_1 and l_2 do not cross any more.

4. If the order of the lines does not matter on C then if they cross we may switch their order on one of C endpoints similarly as we did in 3. Hence we can find a solution where no new crossings are introduced and l_1 and l_2 do not cross on C.

To summarize, we have found a solution where all lines cross exactly n times where n is a lower bound of crossings given by 5.2. All steps were clearly polynomial as each step removed at least one crossing and there is at most a polynomial number of crossings in the problem. ■

For completeness we will calculate how much time will consume this algorithm. A random ordering can be found in $\mathcal{O}(|E|)$ time. In the graph it can be at most $\mathcal{O}(|L|^2|V|)$ crossings if we allow lines to cross at vertices or $\mathcal{O}(|L|^2|E|)$ if we allow lines to cross at edges. Now the calculation of one common subpath takes us $\mathcal{O}(l)$ time where l is the length of that particular subpath. For one pair of lines the maximal length of all common subpaths is $\mathcal{O}(|V|)$ as they are simple paths. Hence all common subpaths can be calculated in $\mathcal{O}(|L|^2|V|)$. For one common subpath the preferred order can be found in constant time as we only need to compare two angles for each common subpath endpoint. There is at most $\mathcal{O}(|L|^2|E|)$ common subpaths and this is time for calculation of preferred order.

Now each step can be done in $\mathcal{O}(l)$ time as it exchanges the drawings between the crossings. For each pair of lines this step decreases the number of crossings that is at most $\mathcal{O}(|V|)$. Hence all iterations to find the optimal solution from that random one take $\mathcal{O}(|V||E|)$ for each pair of lines. As the graph is planar then $|E| = \mathcal{O}(|V|)$. We can conclude that total time is

$$\mathcal{O}\left(|E|\right) + \mathcal{O}\left(|L|^2|E|\right) + \mathcal{O}\left(|L|^2|E|\right) + \mathcal{O}\left(|L|^2|V||E|\right) = \mathcal{O}\left(|L|^2|V|^2\right).$$

We see that the algorithm from our proof is not much fast. This is result of the fact that we pick at beginning any arbitrary ordering of lines. We may have chosen better but our goal was to show a proof, not some new algorithm.

Chapter 6 Evaluation

In this chapter we will use our program to solve the problem on real data provided by the OpenStreetMap. We will use two types of input. First there will be the tourist trails. They can contain few lines but with complicated structure. The other input will contain tram lines in bigger Czech cities. Lines will be simpler (trees or in some inputs only paths), but there will be greater amount of lines.

We will try to measure the time required by the program to the calculate whole problem and to calculate the isolated components. We will investigate how this time depends on the number of unsolved tracks and on number of unsolved common subpaths. After that we will try to analyze the memory requirements for the program. Despite the problems with memory measurements in Java, we will try to measure the approximate memory requirements.

6.1 Algorithm implementation and used hardware

The provided implementation of the algorithm shown in chapter 4 is in the programming language Java. The main class of the program is in archive `MLCMP.jar`. The whole program can be run from a command line using the `java` program as following.

```
java -jar -Xmx512m MLCMP.jar <program_arguments>
```

The program arguments are described in help of the program that will be shown after running the program with `-h` argument. The Java Virtual Machine argument `-Xmx512m` only sets the maximum of memory provided to the program to 512 MB. The required memory is approximately ten times less, but running with small maximum of allocated memory can cause errors. More memory is also required for reading the input directly from OpenStreetMap XML file.

All datasets were tested on Java Standard Edition (version 7u21) platform. The operating system was Windows 7 Professional 64-bit system on an Intel i7-860 with 2.8 GHz and 8 GB RAM. For the time statistics each input was run 50 times to smooth random effects caused by the operating system and the Java environment. The memory statistics were collected during one run, but after several measurements. The time spent by Garbage Collector is separated from the pure run statistics. Also the time required for reading the input is not included into the time measurements.

In the graphs were used two data sets downloaded from the OpenStreetMap on 14th Match 2013 and 19th May 2013. In the graphs we do not distinguish between these two datasets. Half of data marked as the tourist trails data comes from the March input and half comes from the May input.

6.2 Time spent by the program

In this section we present and evaluate some graphs of running time of the program. The program was run in its own thread and another thread was monitoring whether the program is not running longer than it is allowed to. Also it paused the program for a while in a cycle and ran the garbage collector outside. This minimizes the randomization of the running time cased by the garbage collector. Even so the differences were minimal. The running time with garbage collector time included is never two times greater than without it. Most of the data do not differ or they differ minimally.

The overall running time on the tourist trails input is approximately 4 s plus 10 s for reading the graph from the plain text input.[1] The overall running time on the trams map is approximately 3.5 min on the data from March and 1 min on the data from May (if we omit the biggest isolated component). This graph can be loaded within 3 s.

6.2.1 The greedy part running time

Figure 6.1a shows the relation between the running time of the greedy part and the number of edges in the problem. The required time is measured in milliseconds. As the running time is only a few milliseconds, the running time is rounded for small data sets. The fact that the measurements are in milliseconds does not play role, it only shows how the time grows with the input size.

We see that in the case of four lines the values seem to be almost linear, but in the public transportation maps case the input grows with some higher polynomial. But a bigger number of edges means a bigger city and probably a greater number of lines. Hence one more plot 6.4b of the running time of the greedy part is provided. It shows the dependency of running time on the product $|E| \cdot |L|$. Now we see the running times from data sets mixed. The data show a linear dependency on the product $|E| \cdot |L|$.

6.2.2 The CSP part running time

The second part of our program is the searching of the state space. The input of this part is an isolated component. Then the state space of this component is searched. Figure 6.2 shows the running time dependency on the number of edges in the isolated component (figure 6.2a) and the number of common subpaths (figure 6.2b).

Surprisingly, we see that in both figures the data from both input types are mixed. They can be fitted with a straight line. As the vertical axis is logarithmic, the running time grows exponentially. That is what we have expected. The biggest component that has 72 tracks[2] (and represents the Prague tram lines network) was not solved within the time limit of 10 hours. But in this case the graph does not contain only paths but it is more complicated as some lines use two different rails at some areas.

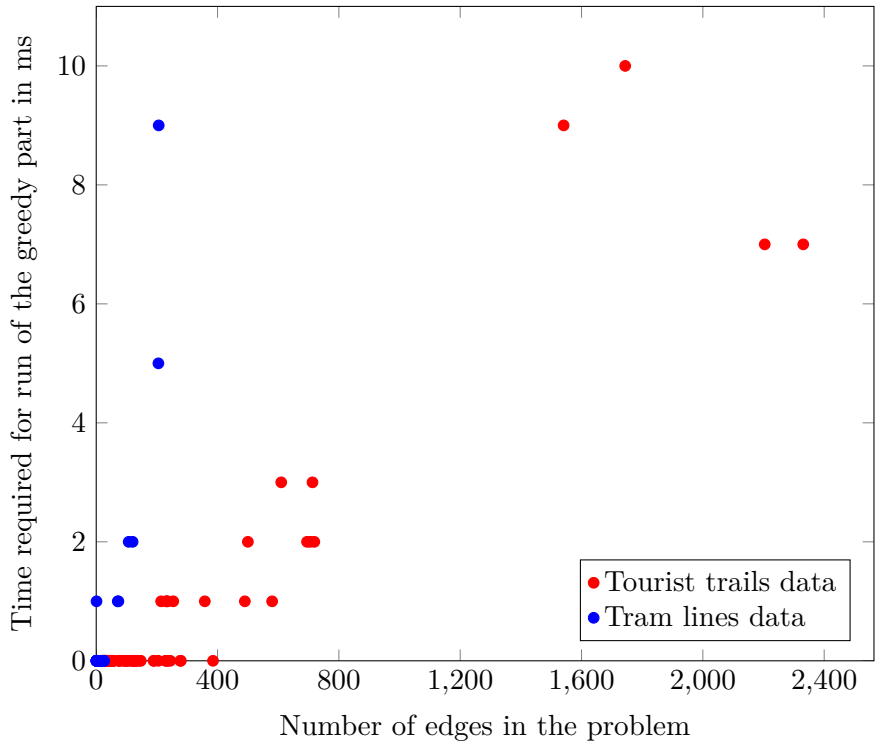
6.3 Memory requirements

In this section we try to analyze memory requirements of the program. As the provided implementation is in Java it is a little bit problematic. The reason is that Java uses the *garbage collecting* for automatic memory management. As a result programmer has limited possibilities how to control the memory.

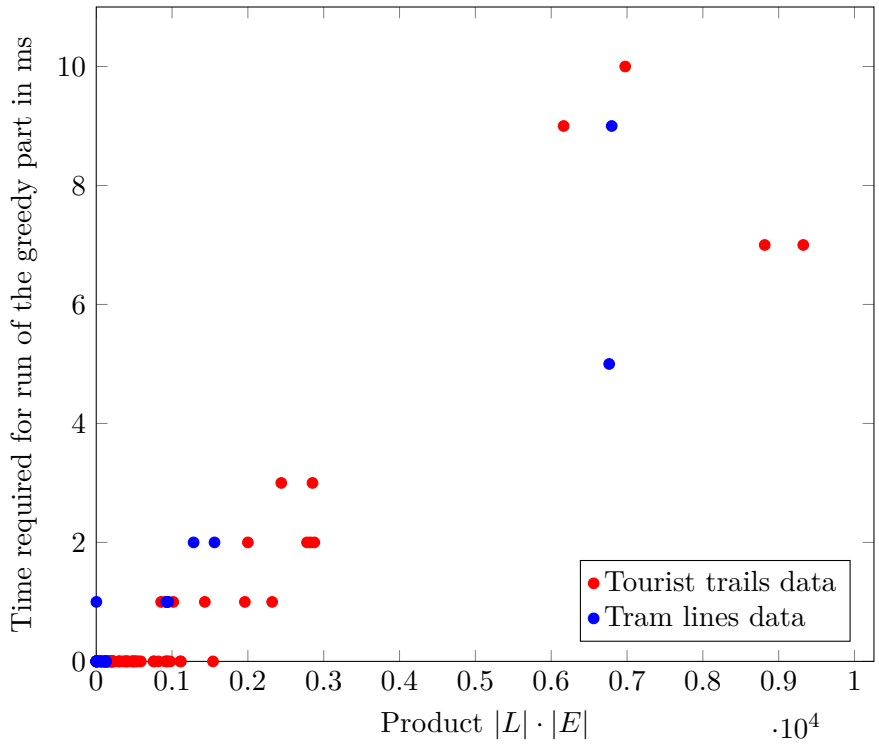
The memory usage was measured similarly as the required time. The program was run in one thread that was paused periodically after some amount of time. Then the garbage

[1] Reading the OpenStreetMap XML file takes approximately 10 min. The 7 GB XML file that contains data from the Czech Republic has to be read three times.

[2] Or 73 in the newer input.

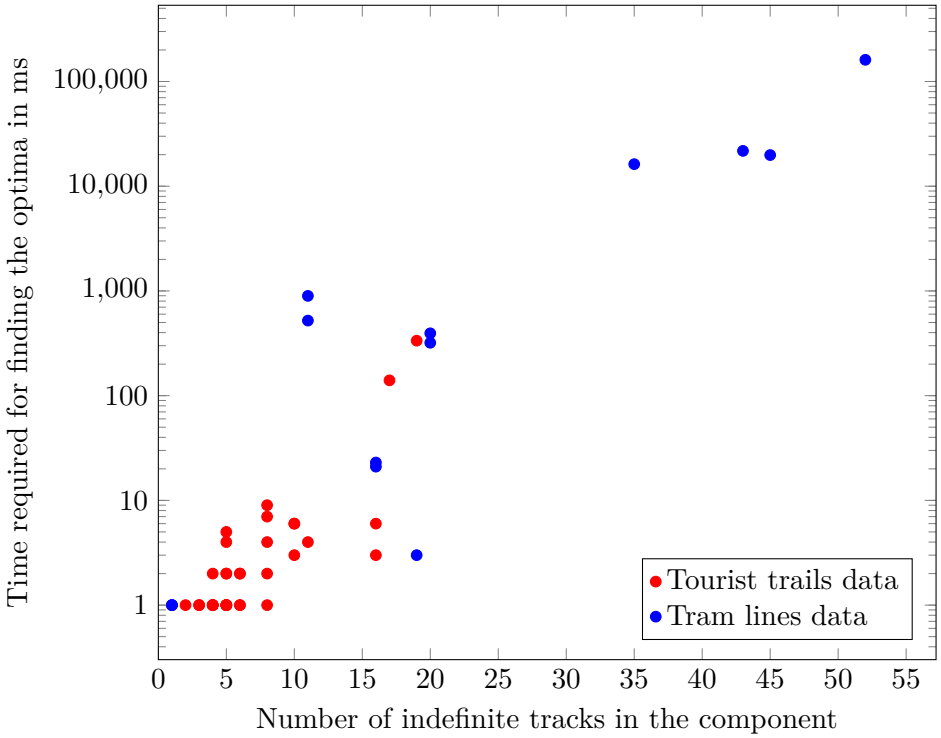


(a) A plot of the time required for computing the greedy part of the algorithm.

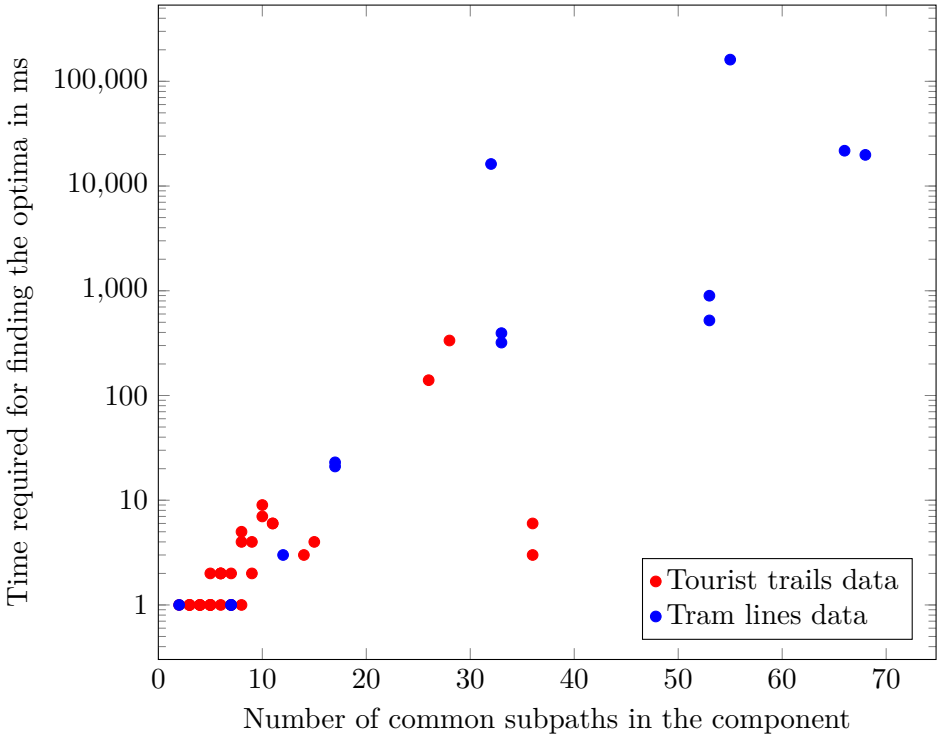


(b) Another plot of the time required for computing the greedy part of the algorithm.

Figure 6.1: Plot of time required for computing the greedy part of the algorithm. The data are rounded to the milliseconds.



(a) Dependency of the time required for solving the isolated component on the number of unsolved tracks in this component.



(b) Dependency of the time required for solving the isolated component on the number of common subpaths with preferred order that is not precedes nor follows in this component.

Figure 6.2: Plots of the time required for computing one isolated component. Note that the vertical axis are in logarithmic scale.

collector was called and the used memory was measured. After that the original thread with the program was resumed. This approach allows us to approximately measure the memory requirements.

From the data we see that the most of memory is required for the model representation. Small isolated components do not require much memory to be allocated additionally to the memory required by the model. As the state space is searched in a depth first order, the memory requirements are minimal. The required memory is linearly dependent on the depth of the search space i.e. the number of cuts that we do in the graph.

The greedy part requires some additional memory. It is caused by building structures like the dependency graph or lists that are caching some additional data. An example of such a list is a list that caches the unsolved tracks. All of the required memory is released when the greedy part is done.

We see that for the tourist trails data is required approximately 21 MB of RAM and for the trams case approximately 1.7 MB. The tourist trails data (from the March input) contain 12128 edges[3] what means 1.73 kB per one edge. In the case of trams in Czech cities we have 489 tracks.[4] That is 3.5 kB per one edge. The reason for bigger amount of memory is a bigger amount of lines in L_e. Memory required for storing the ordering of lines is $O(|L_e|)^2$ and hence the amount of the required memory is greater.

6.4 Success rate of the greedy part

In this section we closely look on the greedy part and compare how it was successful for both input types. By the success rate we mean the relative part of tracks that are resolved as definite after the greedy part. The higher count of definite tracks for a given input may mean many times smaller time of finding the solution. As the greedy part is polynomial and the second part exponential, we want the most of the work to be done in the greedy part.

The figure 6.4 shows this success rate. We see that the greedy part solves (except one case) more than 80 % of edges in the tourist trails input and less than 60 % (again except one case) of edges in the transportation maps input. We see that the topology of the transportation maps is more complex than in the case of tourist trails. Hence the program solves much more faster the tourist trails data than the tram lines data.

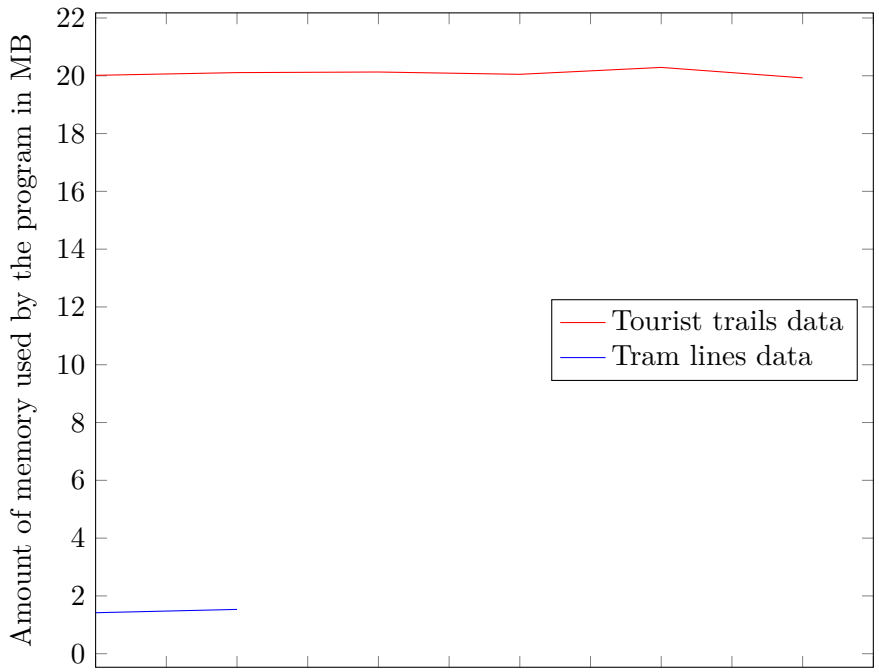
6.5 Summary

The used data were downloaded from the OpenStreetMap. But the OpenStreetMap is community based and hence the data might not be complete. In the case of tourist trails one can easily see that large areas are still missing in the data. The graph of Czech tourist trails is more or less one big connected graph with only a few small connected components outside this graph. But in our data from March 2013 we can see that there are two big components with 2204 edges and 1541 edges. The whole graph has 12 thousand edges.[5] Most of the remaining edges are part of the remaining 18 components with size between one hundred and one thousand. The rest are small components that contain for example five edges. There are about 9 hundred of such components. Anyone who knows tourist tracks in any area can find that some of them are missing.

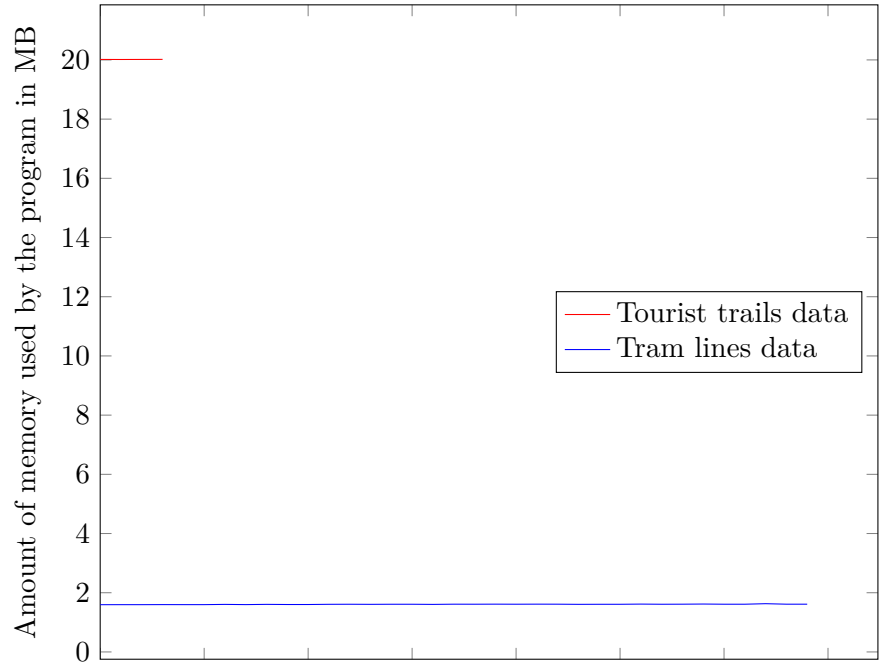
[3] The number of vertices is approximately the same 11493. The number of edges $m = \mathcal{O}(n)$ (and vice versa) as the graph is planar. The maximal planar graph of n vertices contains $3n - 6$ edges ($n > 2$) and the minimal contains $\frac{n}{2}$ edges (for n even). Hence the asymptotic memory and time requirements are the same for edges as well as vertices.

[4] The number of vertices is 406.

[5] All numbers are in the contracted graphs. The original graph has more than 450 thousand edges.

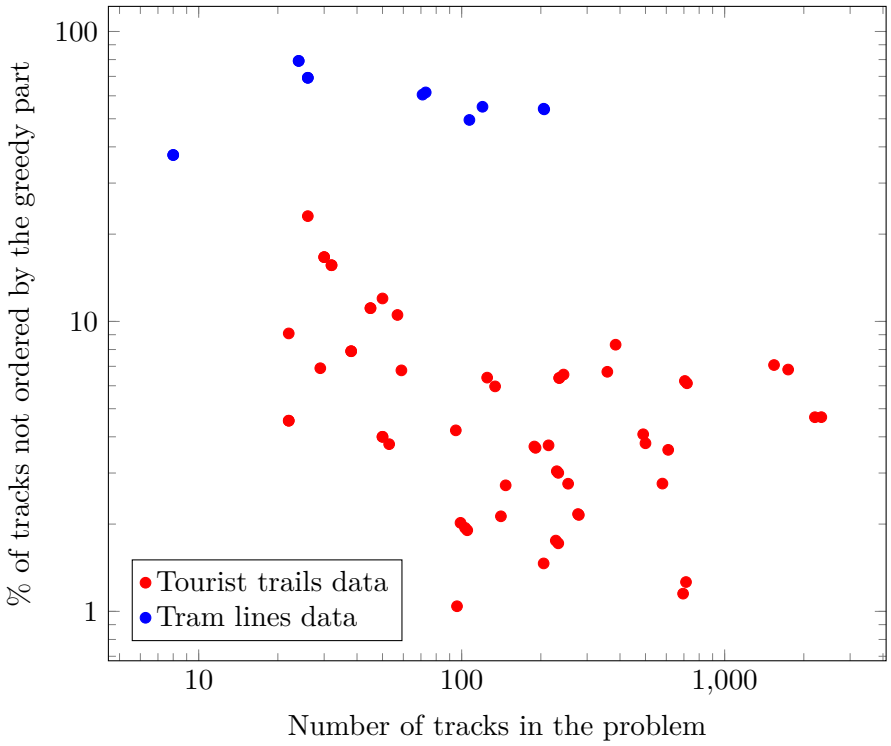


(a) Memory requirements of the greedy part over time.

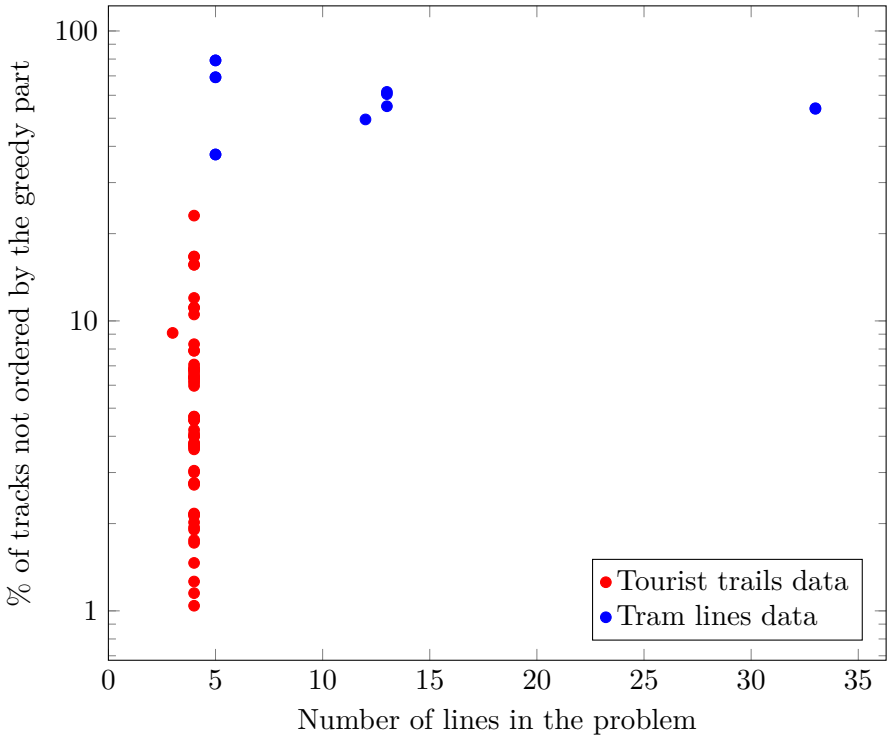


(b) Memory requirements of the CSP part of the algorithm over time. Figure shows the cases of two biggest (in meaning of the number of tracks) isolated components that are in each data set.

Figure 6.3: Plot of the memory requirements of the program over time.



(a) Dependency of relative count of indefinite tracks on the number of tracks in the problem. Note that both axis are in a logarithmic scale.



(b) Dependency of relative count of indefinite tracks on the number of lines in the problem.

Figure 6.4: Plots of dependency of the relative part of tracks that were solved on the input size. Note that the vertical axis are in a logarithmic scale.

On the opposite the OpenStreetMap is developing fast as the users are uploading new and new data. In May, approximately two months later, the tourist trails graph had 627 new edges and the two biggest components had 2331 and 1744 edges.

On the other side the tram lines are complete as many of the data were automatically captured from the public transportation databases. What messes up the input is that the crossroads are stored as form of the physical tracks. Hence the in case of a crossroad where trams go in three directions we can find a triangle in the middle. That makes the input more complex that it needs to be. Other example are the tram turntables. They are stored in the map with all the lines on each track of the turntable. This is another part of the input that increases the complexity. These turntables were replaced by the periphery assignment in the Metro-Line Crossing Minimization Problem-FixedSE.

In conclusion, for the tourist trails the input was less complex than the real data. On the opposite the trams input was more complex than real data despite the fact that it was 25 times smaller in the number of tracks in the problem.

Chapter 7 Conclusion

In this work we have introduced a new general version of the Metro-Line Crossing Minimization Problem, where the lines can be arbitrary graphs even with cycles. We have set up the notation and developed a new algorithm that solves this problem. We also touched the previous works and pointed out the important differences between the given models.

We are able to find an optimal drawing of Czech tourist trails within 20 s and an optimal solution of tram transportation maps in all Czech cities except Prague within 5 min.

7.1 Open problems

There are still big leaks in theory that covers this problem. We will show some hypotheses without a proof that remains to be shown or disproved. At first it still remains open question whether the Metro-Line Crossing Minimization Problem is NP-complete or not. Even in the case of path lines, we are not able to find a sharp lower bound of crossings. On the other side in the case of the MLCMP-FixedSE we know this lower bound for sure and hence we can find the optimal solution in polynomial time. Figures 4.7a and 5.1 show the difference from the case when we have the periphery assignment.

Both avoidable crossings in figures 4.7a and 5.1 are in the case of the preferred order fuzzy precedes (or symmetrically follows). We already know that lines with the preferred order sharp precedes or sharp follows do not cross and lines with not comparable lines cross exactly once. One hypothesis is that the lines whose order is not matter do not cross at all. The proof would be simple if we have shown that there is no line drawn between them, but in some cases in all optimal solutions there may be such a line. As both endpoints can be arbitrarily long, we may introduce a new crossings if we use an exchange argument as we did in the proofs of optimality.

If we were able to show that the situation on figures 4.7a and 5.1 is the only one where we can find a redundant crossing, we could catch those cases and solve them alone.

7.2 Future work

Despite of having presented an algorithm for solving the Metro-Line Crossing Minimization Problem in its most general form, there still remains lot of work to be done.

The output from our algorithm can be used in future with the OpenStreetMap to improve the visualization of maps. But there is still space for changes. For example if we have several possibilities where to cross the lines, we may choose some solution instead of the another under some criteria. Consider a pair of two lines that are not comparable.

We may choose any vertex where they cross. Then we may prefer for example crossing in the middle of the common subpath if it is possible.

There is one great possibility to speed up the program. If we know the lower bound of crossings given by the common subpath preferred order, we may try to develop some heuristic that tries to find a solution that reaches this bound. Then if this heuristic fails, we may search the state space as it is.

Other possibility to future speedup is when the algorithm encounters the situation when the common subpath preferred order is not respected. That is the case shown on figure 4.7. At this moment the program throws away all work already done and searches the whole state space. Better case would if it could only adjust the final solution. On the other side the situation on figure 4.7a is not realistic, it is only a model. In the OpenStreetMap tourist trails and tram lines data for whole Czech Republic it did not appear at all.

Appendix A

The contents of the CD

- `/`
  - `Bachelors_thesis.pdf` ..... this text
  - `Bachelors_thesis_pdf.pdf` ..... this text intended for reading on screen
  - `Bachelors_thesis_print.pdf` ..... this text as it was printed
  - `build` ..... the compiled program
  - `docs` ..... the source code documentation
    - `index.html` ..... the index of the documentation
  - `docs_private` ... the source code documentation including private API members
    - `index.html` ..... the index of the documentation
  - `input`
    - `trails_March.in` .. OpenStreetMap tourist trails data from 14th March 2013
    - `trails_May.in` ..... OpenStreetMap tourist trails data from 19th May 2013
    - `trams_March.in` ..... OpenStreetMap tram lines data from 14th March 2013
    - `trams_May.in` ..... OpenStreetMap tram lines data from 19th May 2013
  - `nbprojects` ..... projects for NetBeans with the program
    - `graphalgos` ..... implementation of the used graph algorithms
    - `MLCMP` ..... main class and the evaluation
    - `MLCMP_input` ..... input and output of the data
    - `MLCMP_model` ..... model of the underlying graph and lines
    - `MLCMP_order` ..... implementation of the algorithm
    - `MLCMP_ordermodel` ..... structures responsible for the ordering storage
    - `MLCMP_preprocess` ..... implementation of the preprocessing step
    - `utilities` ..... various utility classes
  - `sources` ..... the source codes of the program
  - `stats` ..... rough statistical data output of the program
  - `stats_dat` ..... statistical data used for plotting the graphs
    - `trails_csp.dat` ... the results of running the exponential part on the tourist trails input
    - `trails_greedy.dat` the results of running the greedy part on the tourist trails input
    - `trams_csp.dat` the results of running the exponential part on the trams input
    - `trams_greedy.dat` . the results of running the greedy part on the trams input

Appendix B

Program arguments

The program `MLCMP.jar` can be run from a command line as follows.

```
java -jar [-Xmx512m] MLCMP.jar
        [-solve|-stats|-toPlain|-toTEX]
        <-plain|-xml> <inputfile>
        [-trails <intval>]
        [-limit <intval>]
        [-n <intval>]
        [-vlb <intval>]
        [-split <boolval>]
        [-help]
```

`-solve`
: Solves the problem and prints out the optimal number of crossings.

`-stats`
: Solves the problem the given number of times and saves the statistics to a files. There will be two output files with the same file name as the input. One with the extension `.log` for the same output as is in the command line and one with the extension `.out` for the measured time requirements and the memory usage.

`-toPlain`
: Outputs the loaded graph to a plain file. The new file will be named with the same name as the input graph followed by `_plain.in`.

`-toTEX`
: Outputs the solution to a file that can be compiled with `pdflatex`. Note that the output is only approximative and may contain errors or may be unreadable.

`-plain`
: Specifies that the input is loaded from a plain file. See the source code documentation for more information about this file format. This command must be followed by a file name.

`-xml`
: Specifies that the input is loaded from an OpenStreetMap XML file. This command must be followed by a file name.

`-trails`
: This option specifies the type of the data loaded from an OpenStreetMap XML file. This command must be followed by `0` (default) for tourist trails data or by `1` for trams network. This option works only with the `-xml` option.

`-limit`
: A time limit for the greedy part of each problem and for each isolated component. If on the greedy part or on an isolated component is spent more time than this limit, the corresponding thread will be stopped. This option requires an integer value as the next argument and is ignored in case of the `-toPlain` option.

`-n`
: This command specifies the number of iterations of the statistics run. An integer value is required after this command. The `-stats` option is required otherwise this option will be ignored.

`-vlb`
: A lower bound of vertices is specified by an integer that is required after this option. After finding the connected components some of them may be too small. If they have less than the specified number of vertices they will not be solved. This is handy for the tourist trails input from the OpenStreetMap as there is a lot of components that have only a few vertices. This option works only with the `-solve` or `-stats` option.

`-split`
: Defines whether the whole problem from the input shall be divided into the connected components. This may be useful as solving the whole problem in one run may be sometimes faster. This option works only with the `-solve` and `-stats` options. After this option is required `t` for the connected components and `f` for one big graph.

`-help`
: Prints the help.

Appendix C

Program output

The output file produced by the program with the `-stats` option contains a list of the MLCMP instances in the loaded data set. The first line of an MLCMP instance contains the measured values from the greedy part in order as follows. All time measurements are in msand all memory measurements are in MB.

1. Number of tracks in the problem.
2. Number of vertices in the problem.
3. Number of lines in the problem.
4. Number of crossings in the optimal solution or -1 if it was not found within the time limit.
5. Time required for the calculation of the greedy part.
6. Time required for the calculation of the greedy part including the garbage collector time.
7. Used memory at the beginning.
8. Used memory at the end.
9. Maximal amount of used memory over time or -1 if the measurement did not occur.
10. Average amount of used memory over time or -1 if the measurement did not occur.
11. Maximal amount of used memory over time minus the memory at the beginning or -1 if the measurement did not occur.
12. Average amount of used memory over time minus the memory at the beginning or -1 if the measurement did not occur.
13. Time interval between the measurements of the memory.
14. Number of indefinite edges in the problem.
15. Number of common subpaths with different preferred order than precedes or follows.

On the next line follow the measurements of used memory over time. After one blank line follow the measurements of the particular isolated components. Each isolated component is on one line with measurements from following list.

1. Number of indefinite edges in the isolated component.
2. Number of common subpaths with different preferred order than precedes or follows.
3. Time required for finding the optimum.
4. Time required for finding the optimum including the garbage collector.
5. Used memory at the beginning.

6. Used memory at the end.
7. Maximal amount of used memory over time or -1 if the measurement did not occur.
8. Average amount of used memory over time or -1 if the measurement did not occur.
9. Maximal amount of used memory over time minus the memory at the beginning or -1 if the measurement did not occur.
10. Average amount of used memory over time minus the memory at the beginning or -1 if the measurement did not occur.
11. Time interval between the measurements of the memory.

After a blank line follows a list with the measurements of the memory usage over time. If the measurement did not occur, there is `-` symbol. The memory measurements are sorted in the same order as the isolated components.

List of Figures

Bibliography

[1] Argyriou, E., Bekos, M. A., Kaufmann, M., and Symvonis, A. (2009). Two polynomial time algorithms for the metro-line crossing minimization problem. http://www.math.ntua.gr/~mikebekos/pubs/GD2008.pdf.

[2] Argyriou, E., Bekos, M. A., Kaufmann, M., and Symvonis, A. (2010). On metro-line crossing minimization. http://jgaa.info/accepted/2010/ArgyriouBekosKaufmannSymvonis2010.14.1.pdf.

[3] Asquith, M., Gudmundsson, J., and Merrick, D. (2008). An ilp for the metro-line crossing problem. In *Proceedings of the fourteenth symposium on Computing: the Australasian theory - Volume 77*, CATS '08, pages 49–56, Darlinghurst, Australia, Australia. Australian Computer Society, Inc. http://crpit.com/confpapers/CRPITV77Asquith.pdf.

[4] Bekos, M. A., Kaufmann, M., Potika, K., and Symvonis, A. (2007). Line crossing minimization on metro maps. Technical report. http://www.math.ntua.gr/~mikebekos/pubs/GD2007.pdf, longer version at http://www.researchgate.net/publication/30509011_line_crossing_minimization_on_metro_maps/file/79e415129139525f5d.pdf.

[5] Benkert, M., Nöllenburg, M., Uno, T., and Wolff, E. (2007). Minimizing intra-edge crossings in wiring diagrams and public transport maps. In *Proc. 14th Int. Symposium on Graph Drawing (GD'06), volume 4372 of Lecture Notes in Computer Science*, pages 270–281. Springer-Verlag. http://i11www.iti.uni-karlsruhe.de/~awolff/pub/bnuw-miecw-07.pdf.

[6] Dopravní podnik hlavního města Prahy (2012). Tramvaje a metro. http://www.dpp.cz/download-file/5376/05_metro_tram_daily_stops.pdf.

[7] Foursquare (2013). About. http://foursquare.com/about/#maps [accessed 16-May-2013].

[8] Hong, S.-H., Merrick, D., and do Nascimento, H. A. D. (2004). The metro map layout problem. In *Proceedings of the 12th international conference on Graph Drawing*, GD'04, pages 482–491, Berlin, Heidelberg. Springer-Verlag. http://sydney.edu.au/engineering/it/~dmerrick/metromap/metromap.pdf.

[9] Jonathan M. Stott, P. R. (2004). Metro map layout using multicriteria optimization. http://www.cs.kent.ac.uk/pubs/2004/1925/content.pdf.

[10] Karger, D. R. (1993). Global min-cuts in $\mathcal{RNC}$, and other ramifications of a simple min-cut algorithm. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 21–30. Society for Industrial and Applied Mathematics. http://people.csail.mit.edu/karger/Papers/mincut.ps.

[11] Kartographie Berliner Verkehrsbetriebe (BVG) (2012). Berlin Liniennetz Routemap. http://www.s-bahn-berlin.de/pdf/VBB-Liniennetz.pdf.

[12] Nöllenburg, M. and Wolff, A. (2006). A mixed-integer program for drawing high-quality metro maps. In *IN PROC. 13TH INTERNATIONAL SYMPOSIUM ON GRAPH DRAWING*, pages 321–333. Springer-Verlag. http://i11www.ira.uka.de/~awolff/pub/nw-mipdh-06.pdf.

[13] Nöllenburg, M. (2010). An improved algorithm for the metro-line crossing minimization problem. http://i11www.iti.uni-karlsruhe.de/extra/publications/n-iamlc-10.pdf.

[14] Nollenburg, M. and Wolff, A. (2011). Drawing and labeling high-quality metro maps by mixed-integer programming. *IEEE Transactions on Visualization and Computer Graphics*, 17(5):626–641. http://i11www.iti.uni-karlsruhe.de/extra/publications/nw-dlhqm-10.pdf.

[15] OpenStreetMap (2013a). Openstreetmap. http://www.openstreetmap.org/.

[16] OpenStreetMap (2013b). Planet osm. http://planet.openstreetmap.org/ [accessed 16-May-2013].

[17] OpenStreetMap Wiki (2012). About — openstreetmap wiki. http://wiki.openstreetmap.org/w/index.php?title=About&oldid=838461 [accessed 16-May-2013].

[18] OpenStreetMap Wiki (2013). History of openstreetmap — openstreetmap wiki. http://wiki.openstreetmap.org/w/index.php?title=History_of_OpenStreetMap&oldid=897370 [accessed 16-May-2013].

[19] RailCorp (2012). Sydney city rail map. http://www.cityrail.info/stations/pdf/CityRail_network_map.pdf.

[20] Wikipedia (2013a). Harry beck — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Harry_Beck&oldid=552927810 [accessed 16-May-2013].

[21] Wikipedia (2013b). History of cartography — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=History_of_cartography&oldid=554081626 [accessed 16-May-2013].

[22] Wolff, A. (2007). Drawing subway maps: A survey. http://i11www.iti.uni-karlsruhe.de/~awolff/pub/w-dsms-07.pdf.