Master Thesis

Visualization of Event Graphs for Train Schedules

Samuel Wolf

Date of Submission:	October 2, 2024
Advisors:	Prof. Dr. Alexander Wolff
	Prof. Dr. Marie Schmidt
	Dr. Johann Hartleb

Julius-Maximilians-Universität Würzburg
Lehrstuhl für Informatik I
Algorithmen und Komplexität

Contents

Abstract

Software that is used to compute or adjust train schedules is based on so-called event graphs. The vertices of such a graph correspond to events; each event is associated with a point in time, a location, and a train. A train line corresponds to a sequence of events that are associated with the same train. The event graph has a directed edge from an earlier to a later event if they are consecutive along a train line.

We present a way to visualize such graphs. We propose a straight-line drawing of the event graph with the additional constraint that all vertices that belong to the same location lie on the same horizontal line and that the x-coordinate of each vertex is given by its point in time. Hence, it remains to determine the y-coordinates of the locations. A good drawing of a time-space diagram supports users (or software developers) when creating (software for computing) train schedules.

To enhance readability, we define two aesthetic criteria: the number of crossings and the number of turns. We prove that minimizing the number of crossings or minimizing the number of turns is NP-hard, and even NP-hard to approximate. To tackle these challenges, we develop exact reduction rules to reduce the instance size. Additionally, we propose exact algorithms, including integer linear programs and parameterized algorithms, along with heuristics for minimizing the number of crossings and the number of turns. Finally, we experimentally evaluate the performance of these algorithms using real-world test data.

Zusammenfassung

Software, die zur Berechnung oder Anpassung von Zugfahrplänen verwendet wird, basiert auf sogenannten Event-Graphen. Die Knoten eines solchen Graphen entsprechen Ereignissen; jedes Ereignis ist mit einem Zeitpunkt, einem Ort und einem Zug verknüpft. Eine Zuglinie entspricht einer Abfolge von Ereignissen, die demselben Zug zugeordnet sind. Der Event-Graph enthält eine gerichtete Kante von einem früheren zu einem späteren Ereignis, wenn diese entlang einer Zuglinie aufeinanderfolgen.

Wir stellen eine Methode zur Visualisierung solcher Graphen vor. Diese Darstellung ist eine geradlinige Zeichnung unter der zusätzlichen Bedingung, dass alle Knoten, die zu demselben Ort gehören, auf derselben horizontalen Linie liegen und dass die x-Koordinate jedes Knotens durch seinen Zeitpunkt bestimmt ist. Somit bleibt die Bestimmung der y-Koordinaten der Orte als Problem übrig. Eine gute Darstellung des Event-Graphen in Form eines Zeit-Raum-Diagramms unterstützt Benutzer (oder Softwareentwickler) bei der Erstellung (von Software zur Berechnung) von Zugfahrplänen.

Zur Verbesserung der Lesbarkeit definieren wir zwei Kriterien: die Anzahl der Kreuzungen und die Anzahl der Richtungsänderungen. Wir beweisen, dass sowohl die Minimierung der Anzahl der Kreuzungen als auch die Minimierung der Anzahl der Richtungsänderungen NP-schwer ist und zeigen außerdem, dass es sogar NP-schwer ist diese Probleme zu approximieren. Um diese Herausforderungen zu bewältigen, entwickeln wir exakte Reduktionsregeln zur Verkleinerung der Problemgröße. Zusätzlich schlagen wir exakte Algorithmen vor, darunter ganzzahlige lineare Programme und parametrisierte Algorithmen, sowie Heuristiken zur Minimierung der Anzahl der Kreuzungen und Richtungsänderungen. Abschließend evaluieren wir diese Algorithmen experimentell anhand von realen Testdaten hinsichtlich ihrer Laufzeit und Qualität.

1 Introduction

Every day the German train infrastructure needs to accommodate over 50,000 passenger and freight trains used by over 450 companies on a network spanning over 33,000 kilometers according to DB InfraGO AG [AG], a subsidiary of Deutsche Bahn AG. The development of schedules that manage all trains in a fair and efficient way often requires months of intensive work involving manual and automated processes. Unfortunately, no schedule can fully anticipate disturbances or disruptions due to technical outages, weather, or accidents, causing delays and a need for an immediate adjustment to minimize the consequences of these events. These adjustments are localized in the area where the disturbance or disruption occurred and involve re-routing or track changes of trains among other things. However, changing the schedule locally for a small set of trains creates a cascading effect to a broader area and a larger time frame, as the adjustments need to abide to technical constraints such as headway times, which makes changes even more complex.

Historically, immediate adjustments have been done by an experienced team of dispatchers. With the advancements in computer technology and optimization theory in the last decades, computer-aided rescheduling has become possible. Due to the sheer magnitude of the network and the real-time requirements, the scalability of existing algorithmic solutions remains a major obstacle of a full automation. Nonetheless, the integration and further development of autonomous systems is steadily making progress.

DB InfraGO uses an iterative approach consisting of multiple optimization algorithms to quickly find real-time adjustments within a schedule, using an underlying structure called *event graph* that encodes the necessary information for producing a schedule. This event graph also encodes the solution produced by the optimization algorithms and can be turned into a schedule. Roughly speaking, the event graph contains *events* such as railway switches, or start and stop events of trains. Often each event is associated with a place and a time interval. In an event graph, two events can be connected with each other by an edge, which means that they depend on each other in some way. For example, an edge can model the minimum required time that needs to pass between two events. During the usage and the development of the iterative approach certain questions arise, for instance:

- The system has found a new timetable with some delay. Where are the delays, and can we improve?
- Between two iterations: What is the difference between the timetables of the previous and the current iteration?
- The current timetable within the iterative process is not feasible yet. Where are the violations that make the timetable infeasible?

At the point where the questions arise no finished timetable is available yet, and a normal timetable would not be able to provide enough information to answer these questions. Thus, in order to get answers to the questions above, the event graph has to be considered. However, this graph is a large abstract structure that cannot be understood by a human easily. An intuitive way to quickly convey the information of the event graph to a human, and to offer the opportunity to gain insights into the structure of the event graph, is to visualize the event graph. Such a visualization should be able to provide information about the points in time and the locations at which delays and violations occur as well as to provide information about technical constraints. However, while the time dimension should be displayed accurately, the geographical location of events is not inherently necessary since it is more important whether events are at the same location but not where the location is geographically.

Related work. To the best of our knowledge, no previous work has been done on the visualization of event graphs. However, similar research has been done, which we discuss here.

Closely related to event graphs are *alternative graphs*, which model a generalization of the job shop scheduling problem. This generalization includes considerations such as maximum storage times for jobs or a pair of two consecutive jobs, or limited buffer capacities where currently idling jobs must be stored. Alternative graphs have been introduced by Pacciarelli [Pac02] and are widely used for train scheduling problems according to Qu et al. [QCL15], such as the work by D'Ariano et al. [DPP07] or Liu and Kozan [LK09]. Notably, every alternative graph is an event graph but not all event graphs are alternative graphs. Despite the prevalence of alternative graphs, we are not aware of any algorithms for the visualization of alternative graphs.

A visualization technique for the visualization of similar structures is the *time-distance diagram* (or also known as *time-chainage diagram*). This type of diagram displays (linear) time schedules in the context of project management in a variety of use cases [Emm07]. Usually, the time-distance diagram is a 2D-Cartesian coordinate system, where one axis corresponds to the spatial dimension and the other axis corresponds to the time dimension within the project. These types of visualizations are often used in the context of linear construction work such as the construction of tunnels, bridges, or streets. Time-distance diagrams are also used for dispatching trains. In particular, these diagrams are used on rail corridors to schedule trains on a linear piece of infrastructure such as a specific railway connecting two different cities. All named usages have in common that they depict linear processes where the mapping of entities on an axis of a diagram is straightforward. This is not the case when visualizing event graphs since event graphs generally model infrastructure that is larger than a single corridor. Hence, there is no straightforward linear ordering to the entities of the event graph. This poses an algorithmic challenge that the use cases of time-distance diagrams do not possess.

Contribution. For these reasons, we establish a drawing style for the visualization of event graphs that we call *time-space diagram*, which is similar to the previously mentioned time-distance diagram. In a time-space diagram, the events are drawn in a 2D-coordinate system, where the y-coordinate corresponds to the location of an event and the x-coordinate corresponds to the time of an event. We propose algorithms for drawing time-space diagrams with a focus on events belonging to trains. However, we also briefly discuss a way to add additional information such as violations and delays that are compatible with our algorithms. In particular:

- We propose two optimization objectives that may lead to readable drawings. These objectives are the number of crossings and the number of turns.
- We prove that optimizing each of the objectives is NP-hard, and even NP-hard to approximate within a constant factor of the optimal solution.
- We propose a 0–1 integer linear program for optimizing the number of crossings.
- We introduce reduction rules that help to decrease the size of event graphs for the problem of finding time-space diagrams with the minimum number of turns.
- We propose two integer linear program formulations for minimizing the number of turns. The first formulation is an integer linear program that can be combined with the integer linear program for the minimization of the number of crossings. The second formulation is a binary integer linear program.
- We devise exact algorithms working on decompositions and separations for minimizing the number of turns.
- We propose heuristic algorithms for optimizing the number of turns, and experimentally analyze a selection of the proposed heuristic and exact algorithms on real-world data.

Content. We formally define terms such as the time-space diagram and the corresponding optimization criteria, and go through prerequisite knowledge in Chapter 2. We start with investigating the number of crossings as an optimization criterion. In Chapter 3 we show that this optimization problem is NP-hard and even NP-hard to approximate, and propose a 0–1 integer linear program in Chapter 4. Afterwards, we investigate the number of turns as an optimization criterion and prove in Chapter 5 that this optimization problem has a similar complexity compared to the previous criterion. We propose two integer linear programs for minimizing the number of turns in Chapter 7, and two algorithms based on decompositions and separators in Chapter 8. Further, we propose five heuristics that aim to find readable time-space diagrams in Chapter 9. For an experimental analysis of several of the proposed algorithms with respect to quality and runtime, see Chapter 10. Finally, we conclude with recommendations and open problems in Chapter 11.

2 Prerequisites and Problem Definitions

In this chapter we formally define event graphs, time-space diagrams, and two optimization objectives that aim to produce readable diagrams. We start with introducing notation and concepts that are used throughout this thesis.

2.1 Basic Concepts and Notation

Throughout this work, for any positive integer n we write $[n]$ as a shorthand for $\{1, \ldots, n\}$.

Strict total orders. Given a set S and a relation $\prec$ on S, we say that $\prec$ is a *strict total order* if the following properties hold for all $a, b, c \in S$:

1. Asymmetry: if $a \prec b$, then not $b \prec a$
2. Transitivity: if $a \prec b$ and $b \prec c$, then $a \prec c$
3. Connectivity: if $a \neq b$, then $a \prec b$ or $b \prec a$

Given a strict total order $\prec$ on a finite set S, the $\operatorname{rank}(b)$ of an element $b \in S$ is the position in the unique enumeration of S such that for each pair $a \prec b$, a is enumerated before b. Thus, $\operatorname{rank}(b) = |\{a \in S \mid a \prec b\}| + 1$. In the following we assume an *order* to be a strict total order unless specified otherwise.

Graphs. We assume that the reader is familiar with fundamental concepts of graph theory. For a comprehensive reference, we recommend Diestel's textbook on graph theory [Die17]. Here, we introduce only the specific notation used throughout this thesis.

Given a graph G, we denote the set of vertices as $V(G)$ and the set of edges as $E(G)$. If the graph is clear from the context, we simplify these to V and E, respectively. Unless stated otherwise, the number of vertices in the graph is denoted by $n = |V(G)|$, and the number of edges by $m = |E(G)|$. The *neighborhood* of a vertex v is denoted by $N(v)$, and the *closed neighborhood* $N(v) \cup \{v\}$ is denoted by $N[v]$. The maximum vertex degree of a graph G is denoted by $\Delta(G)$.

We write G/S for a graph in which the vertex set $S \subseteq V$ is *contracted* into a single vertex S. Similarly, we write G/e for a graph in which the edge $e \in E$ is contracted into a single vertex e.

We call (A, B) with $A \cup B = V(G)$ a *cut* of a graph G. A set $X \subseteq V(G)$ is a *separator* of (A, B), if on every a–b path, with $a \in A$ and $b \in B$, at least one vertex is in X. If a single vertex acts as a separator in G, we call this vertex a *cut vertex*. If two distinct vertices $s, t \in V(G)$ act as a separator, we call them a *separating pair*. Further, if the vertices of an edge $\{u, v\} \in E$ form a separating pair in G, we refer to the edge $\{u, v\}$ as a *bridge*.

A *rooted tree* is a tree T in which one vertex has been designated as the *root* of T. Note that a rooted tree admits a natural orientation of the edges E either towards or away from the root.

A *tournament graph* G is a directed graph such that for every pair $u, v \in V(G)$ with $u \neq v$ the graph G contains either the edge (u, v) or the edge (v, u).

Given a graph G, a path is called *chain* if all its vertices have degree 2 in G. Note that we consider a single vertex of degree 2 as a chain.

Drawing of a Graph. A *drawing* (*node-link drawing*) Γ of a given graph G is a function Γ, mapping each vertex v in G to a distinct point $\Gamma(v)$ in the plane and each edge $\{u, v\} \in E$ to $\Gamma(\{u, v\}) = \Gamma_{\{u,v\}}([0, 1])$, where $\Gamma_{\{u,v\}}([0, 1])$ is a simple open Jordan curve in the plane such that $\Gamma_{\{u,v\}}(0) = \Gamma(u)$ and $\Gamma_{\{u,v\}}(1) = \Gamma(v)$. For convenience, we often use the same terminology and refer to v and $\Gamma(v)$ as vertex and $\{u, v\}$ and $\Gamma(\{u, v\})$ as edge, if it is clear from the context which mathematical object is meant.

We say that two different edges e, f in a drawing Γ of a graph G *cross* if there is a point x such that $x \in \Gamma_e([0, 1]) \cap \Gamma_f(]0, 1[)$.

A *drawing convention* is a rule that a drawing Γ must satisfy in order to be admissible. Drawing conventions often dictate the shape of the geometric representation of vertices and edges. For example, a drawing convention can require that every edge is drawn as a straight-line segment.

A *drawing aesthetic* is an optimization criterion that drawing algorithms aim to optimize, for example, in order to achieve readability. Common aesthetics include (the minimization of) the number of crossings, (the minimization of) the area of the drawing, and (the minimization of) the maximum edge length.

Parameterized Complexity and Fixed Parameter Tractability (FPT). Problems that can be solved in time $f(k) \cdot n^c$, where $f(k)$ is a computable function, $c > 0$ is a constant, n is the input size, and k is a parameter, are known as *fixed-parameter tractable* (parameterized by the parameter k). The complexity class FPT contains precisely all such fixed-parameter tractable problems with the respective parameter k. Algorithms with such a runtime are commonly referred to as FPT algorithms parameterized by k.

The study of FPT algorithms is particularly motivated by scenarios where certain instance classes have parameters k that are small or constant, making FPT algorithms efficient for these instances, even though the problem is hard for general instances. This efficiency can have significant practical implications, especially when dealing with problems where the input is known to possess a specific structure captured by a bounded parameter k.

However, some problems remain NP-hard, when parameterized by a certain parameter k. We say that these problems are para-NP-hard for this parameter k. For instance, consider the graph coloring problem, where we want to determine whether the vertices of a given graph G can be colored with k colors such that no two vertices that are connected by an edge are colored with the same color. Since it is well known that this problem is NP-hard for $k = 3$, there cannot be an FPT algorithm parameterized by k, assuming $\mathsf{P} \neq \mathsf{NP}$.

The Big-M Method. A core technique used in integer linear formulations is a variant of the "big-M method". With this method we can switch constraints on or off, or count constraints that are violated. We do this by introducing a sufficiently large constant M and change a given constraint $a_1x_1 + \cdots + a_nx_n \geq b$ to the constraint

$$a_1x_1 + \cdots + a_nx_n + Mz \geq b,$$

where z is a binary variable. By setting $z = 0$, the original constraint takes effect and any feasible solution must satisfy the original constraint. However, if we want this constraint to be turned off we can set $z = 1$, with the consequence that the constraint is satisfied due to the sufficiently large constant M regardless of the assignments of the variables $x_1, \ldots, x_n$. Whenever we use this method, we need to make sure to find an appropriate value for M which has to constitute a reasonable bound for $b - a_1x_1 - \cdots - a_nx_n$.

2.2 Event and Location Graphs

Event Graphs. An event graph models trains running on specific routes on an infrastructure via events. This graph includes information about the location at which an event occurs and in which time interval the event *should* occur, but also other technical information such as headway constraints between two different trains.

Definition 1. *An* event graph $\mathcal{E}$ *is an undirected graph. Each vertex e of $\mathcal{E}$, called* event*, is associated with a location $\ell(e)$, a time interval $\tau(e) = [t_1, t_2]$, a point in time $t(e)$ at which the event is scheduled, and a train* $\mathrm{train}(e)$ *to which the event belongs. Each edge in $\{e_1, e_2\}$ of $\mathcal{E}$ is annotated with a minimum time difference $\hat{t}(\{e_1, e_2\})$ necessary between events e_1 and e_2.*

The events in $\mathcal{E}$ are of two types: *real events* corresponding to trains modelling *passing*, *arrival*, or *departure* events and *artificial* events without real locations that can enforce specific requirements to real events within the model. There are also different types of edges that differ in their semantic meaning: There are *running* and *stopping* edges that connect two events of the same train, and *same point* edges that connect two events of different trains at the same railway switch, as well as edges that are incident to artificial events.

An event e is *delayed* if $t(e) > t_2$ with $\tau(e) = [t_1, t_2]$. Further, there is a *violation* of an edge $\{e_1, e_2\}$ contained in $\mathcal{E}$ if $|t(e_1) - t(e_2)| < \hat{t}(\{e_1, e_2\})$. Since the event graph models train schedules, we can construct paths of events $e_1, \ldots, e_j$ in $\mathcal{E}$ for each train k, where $\mathrm{train}(e_1) = \cdots = \mathrm{train}(e_j) = k$ such that $t(e_1) < \cdots < t(e_j)$. We call this path the *train line* of train k.

Since artificial events have no real location associated with it, and different edge types will not influence the time-space diagram, we omit artificial events entirely and do not distinguish between edge types in the rest of this work.

Location Graphs. Based on an event graph $\mathcal{E}$, we define the *location graph* of $\mathcal{E}$ that captures the connections between locations:

Definition 2 (Location Graph)**.** *Let $\mathcal{E}$ be an event graph. The* location graph $\mathcal{L}$ *of $\mathcal{E}$ is an undirected weighted graph, whose vertices are the locations of $\mathcal{E}$. Two locations u, v in the location graph $\mathcal{L}$ are connected with an edge $\{u, v\}$, if there is at least one edge $\{e_1, e_2\}$ in the event graph $\mathcal{E}$ with $\ell(e_1) = u$ and $\ell(e_2) = v$ and* $\text{train}(e_1) = \text{train}(e_2)$*. The weight $w(\{u, v\})$ of an edge in the location graph $\mathcal{L}$ corresponds to the number of edges $\{e_1, e_2\}$ in the event graph $\mathcal{E}$ which are incident to events with location $\ell(e_1) = u$ and $\ell(e_2) = v$ and* $\text{train}(e_1) = \text{train}(e_2)$*.*

Note that a location graph can contain self-loops. Further, note that the train line of a train k in the event graph corresponds to a not necessarily simple path of vertices in the location graph. Abusing the notation of a train line, we also call this path in the location graph a *train line* of k, and assume that this path is consistently directed with the train line of k in the event graph. We call a vertex $v \in V(\mathcal{L})$ a *terminal* vertex, if a train starts or ends at location v.

2.3 The Time-Space Diagram

We formally define a time-space diagram as a drawing style of an event graph, where the x-coordinate of an event corresponds to the time and the y-coordinate to the location of an event.

Definition 3 (Time-Space Diagram)**.** *Let $\mathcal{E}$ be an event graph and let Y be a positive integer. A* time-space diagram *of $\mathcal{E}$ is a drawing Γ of the event graph $\mathcal{E}$ in the plane, where for every event e in $\mathcal{E}$ the x-coordinate corresponds to the time $t(e)$ and there is an injective function $y\colon \ell(V) \to [Y]$ such that the y-coordinate of e is $y(\ell(e))$. The edges of the event graph are drawn as straight-line segments.*

Unless stated otherwise, we assume $Y = |V(\mathcal{E})|$. Further, we call $y(u)$ the *level* of location u in a drawing Γ of $\mathcal{E}$. Note that we have not defined any specific drawing conventions for the visualization of violations and delays. One possible way of visualizing this information is the following: A violated edge in $\mathcal{E}$ is drawn with a distinct color c and a delay of an event e in $\mathcal{E}$ is drawn as a horizontal straight-line segment starting at the event e and ending at t_2 of the time interval $\tau(e) = [t_1, t_2]$.

Note that by Definition 3, two different time-space diagrams for the same event graph only differ in the mapping y. Figure 2.1 shows two different time-space diagrams of the same event graph, where we can see that there are certain properties of mappings y that are more desirable than others.

According to Purchase et al. [PCJ96] the number of crossings is a key factor in the readability of graph drawings. This seems also to be the case in time-space diagrams, as Figure 2.1 suggests, where Figure 2.1a only has one crossing, while Figure 2.1b has three. For this reason, we introduce the following drawing aesthetic:

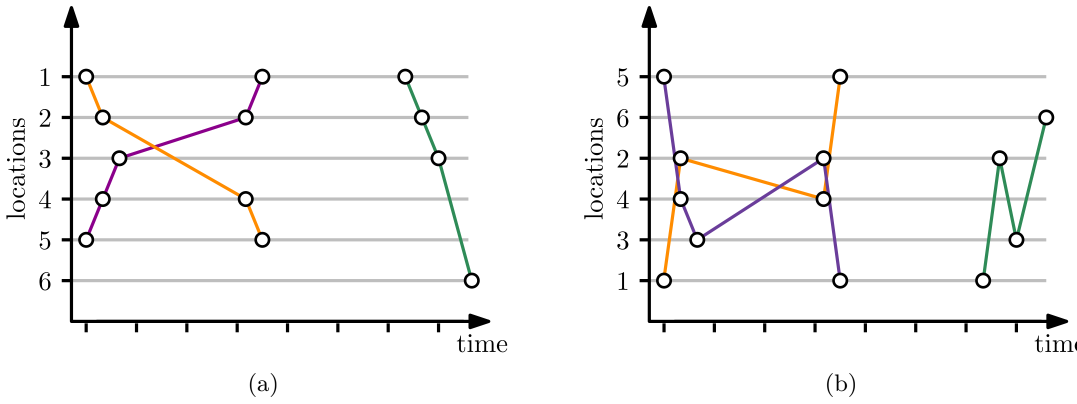

Fig. 2.1: Two different time-space diagrams of the same event graph $\mathcal{E}$ with locations $\{1, 2, \ldots, 6\}$. In Figure 2.1a, the locations are arranged such that there is exactly one crossing and the train lines are y-monotone polylines. The time-space diagram shown in Figure 2.1b has three crossings and the train lines are not y-monotone.

Problem 4 (Time-Space Crossing Minimization (TSCM))**.** *Let $\mathcal{E}$ be an event graph and let Y be a positive integer. Find a time-space diagram Γ of $\mathcal{E}$ such that the number of crossings between the edges of $\mathcal{E}$ is minimized and such that there is no event e_1 such that $\Gamma(e_1) \in \Gamma(\{e_2, e_3\})$ for any $\{e_2, e_3\}$ in $\mathcal{E}$ with $e_1 \neq e_2$ and $e_1 \neq e_3$.*

Additionally, Purchase et al. identified the number of bends in a drawing as a key factor for the readability of graph drawings, where bends refer to changes in the direction of a polyline representing an edge in the graph drawing. Although time-space diagrams require straight-line segments as edges, we observe that a polyline representing a train line is easier to follow when it consistently moves upward or downward, rather than zig-zagging. Since a change in the direction of a polyline can only occur at a vertex corresponding to an event, we distinguish this concept by referring to such a directional change at a vertex as a *turn*.

There are two types of turns that can appear in a time-space diagram, depending on the type of edges that are involved. The first type is a *normal turn*, illustrated in Figure 2.2a, where for three consecutive locations (u, v, w) on a polyline, $y(v)$ is strictly the largest/smallest level. The second type is a *saddle turn*, depicted in Figure 2.2b, where for a k-tuple of events of a train line $(e_1, e_2, \ldots, e_{k-1}, e_k)$ with $k > 3$, the locations $\ell(e_2) = \cdots = \ell(e_{k-1})$, and the levels of $\ell(e_1)$, $\ell(e_k)$ are either both larger or both smaller than l.

The distinction between these turn types yields the following formal definition for the minimization of turns in a time-space diagram:

Problem 5 (Time-Space Turn Minimization (TSTM))**.** *Let $\mathcal{E}$ be an event graph and let Y be a positive integer. Find a time-space diagram Γ of $\mathcal{E}$ that minimizes the number of normal or saddle turns along the train lines in the drawing Γ.*

We can make several observations with respect to the two different optimization problems and their interactions:

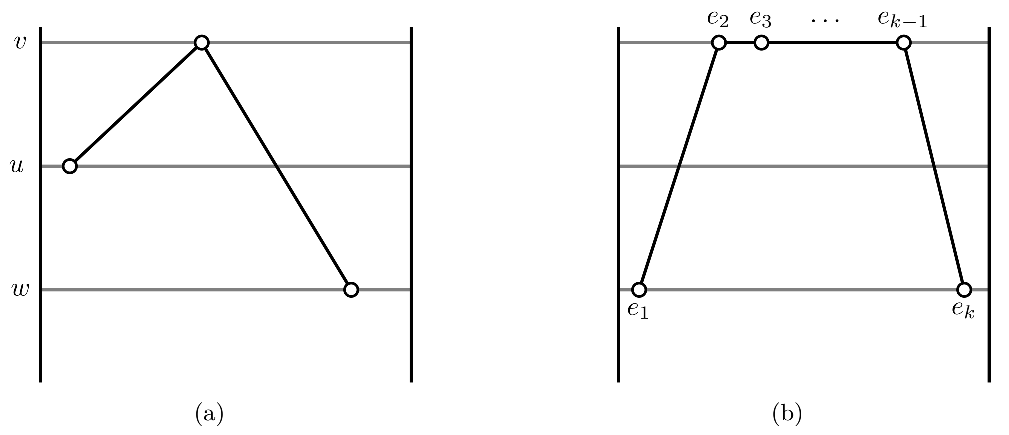

Fig. 2.2: The two types of turns in a time-space diagram. Figure 2.2a shows a normal turn and Figure 2.2b shows a saddle turn. While both figures depict turns, where v or $\ell(e_2), \ldots, \ell(e_{k-1})$ have the largest level, the opposite case is also possible.

First, note that the visualization of delays as described above can create crossings in the drawing. However, when minimizing for the number of crossings, the crossings of edges with delay-edges can also be considered by adding an artificial event e' for each delayed event e at location $\ell(e)$ and at time $t(e') = t_2$, where $\tau(v) = [t_1, t_2]$, and by connecting events e and e'.

Second, note that we can avoid even more crossings if we relax the restriction that the levels have to be integer coordinates and allow that levels are in the range $[1, Y]$. Due to this relaxation, solutions might contain levels that are arbitrary close. In order to restrict this, it is possible to enforce a minimum distance between two levels. We call this relaxation the *slack-level model*. In the following, we also mention how to adapt the algorithm for finding the minimum number of crossings in time-space diagrams such that it can also find such a drawing in the slack-level model.

Third, note that a turn-optimal drawing of an event graph is determined solely by the order implied by y. Consequently, this problem only concerns the strict ordering of the locations, and the specific value of $y(v)$ for a location v is not crucial. For this reason, we will often not state concrete y-coordinates but only an ordering of locations.

Part I

Minimizing the Number of Crossings

3 The Complexity of TSCM

In this chapter we show that TSCM is NP-hard by showing that the corresponding decision version of this problem is NP-hard. We reduce from Betweenness which was shown to be NP-hard by Opatrny [Opa79]. The problem is defined as follows:

Problem 6 (Betweenness [Opa79])**.** *Let S be a finite set and let $R \subseteq S \times S \times S$ be a finite set of ordered triplets. Is there a total ordering $\prec$ of S such that for each triplet $(a, b, c) \in R$ either $a \prec b \prec c$ or $c \prec b \prec a$ holds?*

Theorem 7 (NP-hardness of TSCM)**.** *Given an event graph $\mathcal{E}$ and a positive integer k, it is NP-hard to decide whether there is a time-space diagram of $\mathcal{E}$ with at most k crossings.*

Proof. We show that the problem is NP-hard via a reduction from Betweenness. Let (S, R) be an instance of the Betweenness problem. We construct a TSCM instance $(\mathcal{E}, Y)$ corresponding to (S, R).

We set $Y = |S|$ and consider the set S as the set of locations of the event graph we construct. In order to construct the event graph $\mathcal{E}$, we consider R in an arbitrary but fixed order, where we let the i-th element in the order be denoted by (a_i, b_i, c_i). For each triplet $(a_i, b_i, c_i) \in R$ we construct a gadget in $\mathcal{E}$ which is composed of three paths $\Lambda_i^1 = \langle u_1, u_2, u_3 \rangle$, $\Lambda_i^2 = \langle v_1, v_2, v_3 \rangle$, and $\Xi_i = \langle w_1, w_2, w_3 \rangle$, where we call Λ_i^1, Λ_i^2 *spikes* and Ξ_i *ordering-lines.* Since Λ_i^1, Λ_i^2, and Ξ_i are individual paths in $\mathcal{E}$, they can be seen as trains. We assign the locations of the events of a gadget such that the events u_1, u_3, w_2, v_1, v_3 take place at location b_i, the events w_1, v_2 take place at location a_i, and the events u_2, w_3 take place at location c_i. For an example of this gadget, see Figure 3.1a.

We now assign times to each event in a gadget in a way that ensures that there is a crossing between a spike and the ordering line if and only if the level of the location b_i is not between the levels of locations a_i and c_i.

The gadget of each triplet $(a_i, b_i, c_i) \in R$ is allocated a time interval of $(i, i+1)$. Within this time interval the two spikes and the ordering-line are placed such that their events are ordered

$$t(u_1) < t(w_1) < t(u_2) < t(u_3) < t(w_2) < t(v_1) < t(v_2) < t(w_3) < t(v_3)$$

along the time axis.

Claim 7.1. *There is a solution to (S, R) if and only if there is a time-space diagram Γ of $(\mathcal{E}, Y)$ with an ordering of the locations such that there is no crossing in Γ.*

Proof. Let $\prec$ be a valid ordering for the instance (S, R). According to the transformation, the locations of the event graph $\mathcal{E}$ correspond to the elements of S. By ordering the

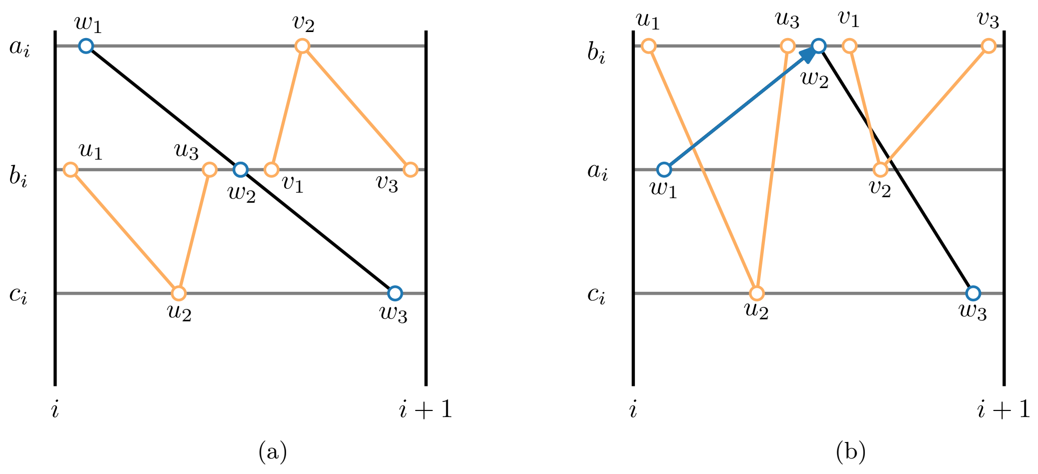

Fig. 3.1: The gadget for the reduction from BETWEENNESS to TSCM. On the left, the constraint (a_i, b_i, c_i) is satisfied and therefore there is no crossing in the gadget. On the right, the constraint is violated, as the level b_i is not between a_i and c_i. In this case, $\{w_1, w_2\}$ and $\{u_2, u_3\}$ must cross, but more crossings are possible.

locations of $\mathcal{E}$ with respect to $\prec$, where the smallest element in $\prec$ is assigned the top most level, we obtain a crossing-free time-space diagram.

First, note that there can be only crossings between edges within the same gadget, since the time intervals of the gadgets do not overlap. Therefore, consider the gadget of a triplet $(a_i, b_i, c_i) \in R$ and assume that, without loss of generality, $a_i \prec b_i \prec c_i$ (the case where $c_i \prec b_i \prec a_i$ is symmetric). Since the time interval of the edge $\{w_1, w_2\}$ in Ξ_i only intersects the time intervals of the edges of spike Λ_i^1, the edge $\{w_1, w_2\}$ can only cause crossings with the edges of Λ_i^1. By construction $\ell(w_1) = a_i$, $\ell(w_2) = \ell(u_3) = \ell(u_1) = b_i$, and $\ell(u_2) = c_i$. This means that the edges of Λ_i^1 are drawn between the levels of b_i and c_i and the edge $\{w_1, w_2\}$ is drawn between the levels of a_i, and b_i. Thus, Λ_i^1 and $\{w_1, w_2\}$ are horizontally separated by the level of location b_i, therefore preventing any crossings. With the same arguments there can be no crossings between the edges of Λ_i^2 and the edge $\{w_2, w_3\}$. Hence, if there is a valid ordering $\prec$ of the BETWEENNESS instance (S, R), there is a drawing of $(\mathcal{E}, Y)$ such that no edges cross.

Conversely, let $(\mathcal{E}, Y)$ be a TSCM instance constructed according to the transformation described above and let Γ be a crossing-free time-space diagram of $\mathcal{E}$. We show that the ordering obtained from the mapping y of locations to levels constitutes a valid ordering $\prec$ for the BETWEENNESS instance (S, R). Suppose there was a triplet (a_i, b_i, c_i) that is violated by $\prec$, i.e. b_i is not between a_i and c_i with respect to $\prec$. Without loss of generality, let $b_i \prec a_i \prec c_i$ as in Figure 3.1b (all other cases are symmetric). Consider the directed line spanned by the vector $\vec{w_1 w_2}$ of the edge $\{w_1, w_2\}$ in Ξ_i and consider the edge $\{u_2, u_3\}$ in Λ_i^1. By construction, the events occur in the order

$$t(w_1) < t(u_2) < t(u_3) < t(w_2). \tag{3.1}$$

Since the events w_2 and u_3 share the same location b_i, the event u_3 is located on the left side of the line through $\vec{w_1 w_2}$. Similarly, as event u_2 has location c_i, u_2 must be located on the right side of $\vec{w_1 w_2}$, as suggested in Figure 3.1b. Thus, u_2 and u_3 lie on different sides of $\{w_1, w_2\}$ and together with inequalities 3.1 this implies that a crossing between the edges $\{w_1, w_2\}$ and $\{u_2, u_3\}$ occurs. This is a contradiction to the assumption that Γ is crossing-free. Hence, a crossing-free drawing Γ of the constructed instance $(\mathcal{E}, Y)$ yields a valid ordering $\prec$ for the BETWEENNESS instance (S, R). $\lhd$

Note that this proof only requires that inequality 3.1 holds. Therefore, we omitted precise time values. Since a gadget of a triplet has constant size, the reduction can be performed in polynomial time. Note that the NP-hardness of the decision version immediately implies that the optimization version of TSCM is NP-hard as well. $\square$

Based on this reduction, we can observe some implications on the complexity of other algorithmic approaches assuming $\mathsf{P} \neq \mathsf{NP}$. In the reduction we showed that we can decide BETWEENNESS by testing whether there are *no* crossings on the transformed instance. This immediately eliminates the possibility of an FPT algorithm parameterized by the number of crossings for TSCM, due to the fact that any FPT algorithm whose only parameter is either the number of crossings would be able to decide BETWEENNESS in polynomial time.

Corollary 8. *TSCM is* para-NP*-hard when parameterized by the number of crossings.*

Unfortunately, approximation algorithms with a multiplicative or a constant additive factor are also unlikely for similar reasons. A multiplicative α-approximation algorithm for TSCM is ruled out by the fact that such an approximation algorithm would have to solve the cases with $0 = 0 \cdot \alpha$ crossings optimally.

If there was an additive β-approximation algorithm for a constant β, we could copy each gadget $\beta + 1$ times. By duplicating the gadgets $\beta + 1$ times for a triplet (a_i, b_i, c_i), a crossing in one gadget causes all other copies of the gadget of the triplet to have a crossing as well. Thus, the number of crossings is divisible by $\beta + 1$. If there was an optimal mapping from locations to levels without crossings, the additive algorithm would also have to return the optimal value, since the only integer in the range $\{0, \beta\}$ that is divisible by $\beta + 1$ is 0, implying that we could again use this additive algorithm to decide BETWEENNESS.

Corollary 9. *Assuming* $\mathsf{P} \neq \mathsf{NP}$*, there is neither a multiplicative α-approximation with $\alpha \in \text{poly}(n)$ nor an additive β-approximation with $\beta \in \mathcal{O}(1)$ for TSCM.*

4 A 0–1 Integer Linear Program

In this chapter we propose a binary integer linear program for TSCM. This formulation can also be transformed into a mixed-integer linear program to provide crossing-optimal time-space diagrams, if the requirement of integer levels is relaxed and a slack-level model (as described in Chapter 2) is used. We start with modelling the levels. Since our formulation for TSCM and our formulation for the slack-level model only differs in the modelling of the levels, we also describe how to formulate the levels in the slack-level model.

Modelling the Levels. Let $\mathcal{E}$ be a given event graph and let Y be a positive integer. We begin with formulating the levels for the TSCM. We introduce binary variables y_v^l to denote that $y(v) = l$, and introduce constraints that ensure that exactly one level l is picked for exactly one location v:

$$\sum_{v \in \ell(V)} y_v^l \leq 1 \qquad \forall\, l \in \{1, \dots, Y\} \tag{4.1}$$

$$\sum_{l=1}^{Y} y_v^l = 1 \qquad \forall v \in \ell(V) \tag{4.2}$$

We can then access the level of a location v with the following expression, due to constraint 4.2:

$$\sum_{l=1}^{Y} l \cdot y_v^l. \tag{4.3}$$

For the slack-level model which requires that two levels $y(u)$, $y(v)$ have a distance $|y(u) - y(v)| \geq \delta$ we propose the following formulation: We introduce continuous variables $y_v \in [1, Y]$ that represent the levels of locations $v \in \ell(V)$. In order to model the requirement that the levels of two distinct locations u, v are at least δ apart we use the big-M method with the auxiliary binary variable L_{uv} to enforce that either $y_u - y_v \geq \delta$ or $y_v - y_u \geq \delta$ is true:

$$y_u - y_v + (Y + \delta - 1) \cdot L_{uv} \geq \delta \qquad \forall \{u, v\} \in \binom{\ell(V)}{2} \tag{4.4}$$

$$y_v - y_u + (Y + \delta - 1) \cdot (1 - L_{uv}) \geq \delta \qquad \forall \{u, v\} \in \binom{\ell(V)}{2}. \tag{4.5}$$

Since the distance relation between u and v is symmetric we need the sets of constraints 4.4 and 4.5 only for each unordered pair u, v. Note that the constant $Y + \delta - 1$ is indeed sufficiently large and can not be made smaller since the maximum distance between two levels is $Y - 1$.

Modelling the Crossings. In the following, we use the variable y_v to denote the level of location v. Recall that for the case of the original TSCM, where integer levels are required, the variable y_v corresponds to the expression in Equation (4.3).

In order to determine whether two edges $\{u, v\}$, $\{w, s\}$ cross, we consider the orientation of the vertices of one edge $\{u, v\}$ from the perspective of the other edge $\{w, s\}$. For this reason, we direct an edge $\{u, v\}$ such that u points to v if $t(u) < t(v)$, as Figure 4.1a illustrates. For simplicity, we write uv instead of the directed edge (u, v), and we write $\vec{uv}$ for the vector corresponding to the directed edge uv. Further, let $t(u, v) = [t(u), t(v)]$ be the time interval of an edge uv.

A necessary condition for a crossing between uv and ws is that the vertices w and s lie on different sides of the line spanned by vertices of uv. The orientation of three points u, v, and w can be determined with the sign of the determinant $\det([\vec{uv}\,\vec{uw}]) = \|\vec{uv}\| \cdot \|\vec{uw}\| \sin\theta$, where $[\vec{uv}\,\vec{uw}]$ is a 2×2 matrix with column vectors $\vec{uv}$, $\vec{uw}$, $\|\vec{uv}\|$, $\|\vec{uw}\|$ is the magnitude of the vectors, and θ is the angle between both vectors. If w is to the left of $\vec{uv}$, then $\det([\vec{uv}\,\vec{uw}]) > 0$, as the angle θ is in the range $(0, \pi)$ and therefore $\sin\theta > 0$. Conversely, if θ is in the range $(\pi, 2\pi)$, i.e. w is to the right of $\vec{uv}$, $\sin\theta$ is negative and consequently $\det([\vec{uv}\,\vec{uw}]) < 0$. Although, testing this condition for one edge $\vec{uv}$ is not sufficient, as Figure 4.1b shows, testing the condition for both edges uv, ws is sufficient to detect a crossing.

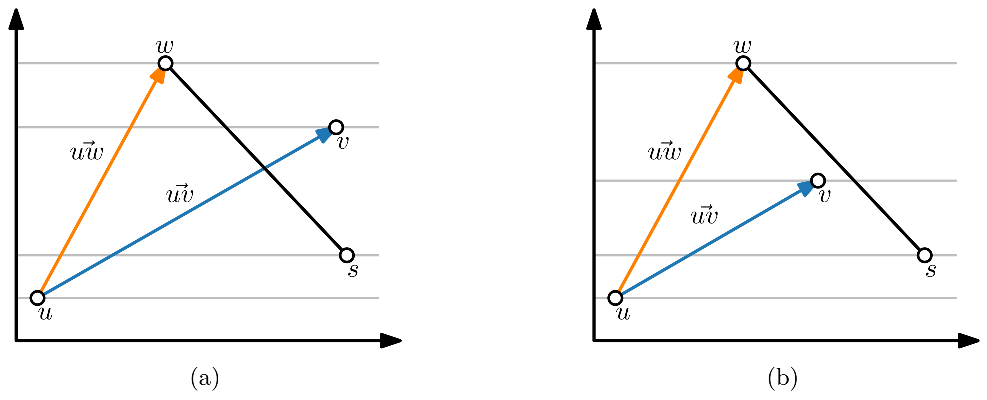

Fig. 4.1: Figure 4.1a shows how we direct an edge $\{u, v\}$ according to the time of u and v, and consider the orientation (left/right) of w and s with respect to $\vec{uv}$. Figure 4.1b illustrates that the orientation with respect to only one edge is not enough to determine a crossing between $\{u, v\}$ and $\{w, s\}$ as w is to the left of $\vec{uv}$ and s to the right of $\vec{uv}$ even though both edges do not cross. But when determining the orientation from both edges, a crossing can be detected.

We calculate the determinant of an edge uv with respect to a vertex w via

$$\begin{aligned}\det([\vec{uv}\,\vec{uw}]) &= \begin{vmatrix} t(v)-t(u) & t(w)-t(u) \\ y_{\ell(v)}-y_{\ell(v)} & y_{\ell(w)}-y_{\ell(u)} \end{vmatrix} \\ &= \underbrace{(t(v)-t(u))}_{\text{const.}}\cdot(y_{\ell(w)}-y_{\ell(u)}) - \underbrace{(t(w)-t(u))}_{\text{const.}}\cdot(y_{\ell(v)}-y_{\ell(u)}). \end{aligned} \tag{4.6}$$

Since the time coordinate of all events is fixed, we obtain a linear expression of the determinant. Note that the sign of the determinant is not always unambiguous since $|\det([\vec{uv}\,\vec{uw}])| = 0$ is possible. However, in the definition of TSCM (see Problem 4), it is prohibited that any valid drawing Γ contains a vertex u that is drawn on another edge $\{v,w\}$, where $u \neq v$ and $u \neq w$. Further, note that we can restrict the computation of the determinant to pairs of edges uv, ws, whose time interval overlap, as otherwise no crossing is possible. Therefore, we can always assume that $|\det([\vec{uv}\,\vec{uw}])| \geq \varepsilon$ for some $\varepsilon > 0$, which means that the sign of the determinant is always well-defined.

Thus, we can use Equation (4.6) to determine whether a crossing exists between two edges uv, ws, whose time intervals overlap by verifying if the vertices w, s lie on different sides of uv and the vertices u, v lie on different sides of ws.

Formally, we define the set $\mathcal{S}$ of an event graph $\mathcal{E}$ containing edge pairs uv, ws whose time intervals overlap as

$$\mathcal{S} = \{\{uv, ws\} \mid \{u,v\},\{w,s\} \in E, \{u,v\} \neq \{w,s\}, t(u,v) \cap t(w,s) \neq \varnothing\}. \tag{4.7}$$

Moreover, we define the set $\mathcal{R}$ as the set containing an edge-vertex pair whose orientation must be computed in order to determine a crossing:

$$\mathcal{R} = \{(uv,w),(uv,s),(ws,u),(ws,v) \mid \{uv,ws\} \in \mathcal{S}\}. \tag{4.8}$$

For each edge pair $\{uv, ws\}$ in $\mathcal{S}$ we define binary variables x_{uv}^{ws} that indicate a crossing between uv and ws. Further, we define a binary orientation variable o_{uv}^{w} for each $(uv,w) \in \mathcal{R}$ denoting the sign of $\det([\vec{uv}\,\vec{ws}])$, i.e. $o_{uv}^{w} = 1$ if and only if w is to the left of $\vec{uv}$ and 0 otherwise. We can compute the value of x_{uv}^{ws} according to our previous considerations by

$$x_{uv}^{ws} = (o_{uv}^{w} \oplus o_{uv}^{s}) \wedge (o_{ws}^{u} \oplus o_{ws}^{v}),$$

where $\oplus$ denotes the exclusive-or-operator. Consequently, we need to introduce variables and constraints expressing the logical statements $\oplus$ and $\wedge$, as well as a representation of the sign of the determinant via the binary orientation variables.

We use the big-M method to map the sign of the determinant to the orientation variables by introducing the following constraints for each orientation variable o_{uv}^{w}:

$$\det([\vec{uv}\,\vec{uw}]) + D_{uv}^{w}\cdot(1-o_{uv}^{w}) \geq \varepsilon \tag{4.9}$$
$$-\det([\vec{uv}\,\vec{uw}]) + D_{uv}^{w}\cdot o_{uv}^{w} \geq \varepsilon, \tag{4.10}$$

where D_{uv}^{w} is a sufficiently large upper bound. If $\det([\vec{uv}\,\vec{uw}])$ is positive, i.e. w is to the left of $\vec{uv}$ and is at least ε, constraint 4.10 requires o_{uv}^{w} to be 1, while constraint

4.9 is indifferent towards the assignment of o^w_{uv}. Conversely, if w is to the right of $\vec{uv}$, and therefore $\det([\vec{uv}\,\vec{uw}])$ is negative, constraint 4.9 forces o^w_{uv} to be 0, while constraint 4.10 is automatically fulfilled assuming that $|\det([\vec{uv}\,\vec{uw}])| \geq \varepsilon$. Thus, we have linear constraints, stating that an orientation variable o^w_{uv} is equal to 1, if w is to the left of $\vec{uv}$ and 0 otherwise.

For each orientation variable o^w_{uv} we can construct a sufficiently large upper bound by considering Equation (4.6):

$$\begin{aligned}\det([\vec{uv}\,\vec{uw}]) &= (t(v) - t(u)) \cdot (y_{\ell(w)} - y_{\ell(u)}) - (t(w) - t(u)) \cdot (y_{\ell(v)} - y_{\ell(u)}) \\ &\leq (t(v) - t(u)) \cdot Y + (t(w) - t(u)) \cdot Y \\ &\leq 2 \max\{|t(v) - t(u)|, |t(w) - t(u))|\} \cdot Y =: D^w_{uv}\end{aligned}$$

The vertical distance between two levels is trivially bounded by Y. Since each factor can be negative, an upper bound is the sum of the absolute values of each term.

Now, we formulate constraints modelling the logical expressions, starting with the xor-expression. Let c^{ws}_{uv} be a binary variable equal to the logical expression $c^{ws}_{uv} = o^w_{uv} \oplus o^s_{uv}$. The following constraints ensure this equivalence:

$$c^{ws}_{uv} \leq o^w_{uv} + o^s_{uv} \tag{4.11}$$
$$c^{ws}_{uv} \geq o^w_{uv} - o^s_{uv} \tag{4.12}$$
$$c^{ws}_{uv} \geq o^s_{uv} - o^w_{uv} \tag{4.13}$$
$$c^{ws}_{uv} \leq 2 - o^w_{uv} - o^s_{uv} \tag{4.14}$$

Constraints 4.12 and 4.13 force c^{ws}_{uv} to be 1 if $o^w_{uv} \oplus o^s_{uv} = 1$. Similarly, constraints 4.11 and 4.14 require $c^{ws}_{uv} = 0$, whenever $o^w_{uv} = o^s_{uv}$. Therefore, these constraints suffice to ensure that $c^{ws}_{uv} = o^w_{uv} \oplus o^s_{uv}$. With the help of the binary variables c^{ws}_{uv}, we can formulate constraints such that $x^{ws}_{uv} = c^{ws}_{uv} \wedge c^{uv}_{ws}$. We need to force $x^{ws}_{uv} = 0$ if either c^{ws}_{uv} or c^{uv}_{ws} are zero, and make sure that $x^{ws}_{uv} = 1$, if both c^{ws}_{uv} and c^{uv}_{ws} are equal to 1. Hence, the following constraints suffice:

$$x^{ws}_{uv} \leq c^{ws}_{uv} \tag{4.15}$$
$$x^{ws}_{uv} \leq c^{uv}_{ws} \tag{4.16}$$
$$x^{ws}_{uv} \geq c^{ws}_{uv} + c^{uv}_{ws} - 1. \tag{4.17}$$

The Complete ILP Formulation. In summary, the following integer linear program models TSCM:

$$\begin{aligned}
\text{minimize} \quad & \sum_{\{uv,ws\}\in\mathcal{S}} x_{uv}^{ws} && && (4.18)\\
\text{subject to} \quad & \sum_{v\in\ell(V)} y_v^l \le 1 && \forall\, l \in [Y] && (4.19)\\
& \sum_{l=1}^{Y} y_v^l = 1 && \forall v \in \ell(V) && (4.20)\\
& \varepsilon \le \det([\vec{uv}\,\vec{uw}]) + D_{uv}^w \cdot (1 - o_{uv}^w) && \forall (uv, w) \in \mathcal{R} && (4.21)\\
& \varepsilon \le -\det([\vec{uv}\,\vec{uw}]) + D_{uv}^w \cdot o_{uv}^w && \forall (uv, w) \in \mathcal{R} && (4.22)\\
& c_{uv}^{ws} \le o_{uv}^w + o_{uv}^s && \forall \{uv, ws\} \in \mathcal{S} && (4.23)\\
& c_{uv}^{ws} \ge o_{uv}^w - o_{uv}^s && \forall \{uv, ws\} \in \mathcal{S} && (4.24)\\
& c_{uv}^{ws} \ge o_{uv}^s - o_{uv}^w && \forall \{uv, ws\} \in \mathcal{S} && (4.25)\\
& c_{uv}^{ws} \le 2 - o_{uv}^w - o_{uv}^s && \forall \{uv, ws\} \in \mathcal{S} && (4.26)\\
& x_{uv}^{ws} \le c_{uv}^{ws} && \forall \{uv, ws\} \in \mathcal{S} && (4.27)\\
& x_{uv}^{ws} \le c_{ws}^{uv} && \forall \{uv, ws\} \in \mathcal{S} && (4.28)\\
& x_{uv}^{ws} \ge c_{uv}^{ws} + c_{ws}^{uv} - 1 && \forall \{uv, ws\} \in \mathcal{S} && (4.29)\\
& y_v^l \in \{0, 1\} && \forall v \in \ell(V) \forall l \in [Y] && (4.30)\\
& x_{uv}^{ws}, c_{uv}^{ws} \in \{0, 1\} && \forall \{uv, ws\} \in \mathcal{S} && (4.31)\\
& o_{uv}^w \in \{0, 1\} && \forall (uv, w) \in \mathcal{R} && (4.32)
\end{aligned}$$

By definition, we know that $|R| = 4|S|$ and since the formulation only uses binary variables in order to determine whether a crossing between to edges $\{uv, ws\} \in \mathcal{S}$ exists, we conclude the integer linear formulation for TSCM with the following theorem:

Theorem 10. *Let $\mathcal{E}$ be an event graph and let $\mathcal{S}$ the set of relevant edge pairs as defined in Equation* (4.7)*. The 0–1 integer linear program described in Equations* (4.18) *to* (4.32) *models the TSCM problem. This formulation uses $\mathcal{O}(Y \cdot |\ell(V(\mathcal{E}))| + |\mathcal{S}|)$ binary variables, and $\mathcal{O}(Y + |\ell(V(\mathcal{E}))| + |\mathcal{S}|)$ constraints.*

Turns vs. Crossings. Consider Figures 4.2 and 4.3. Both figures show a time-space diagram of the same real-world instance, where the time-space diagram shown in Figure 4.2 is optimal with respect to the number of turns and where the time-space diagram shown in Figure 4.3 is optimal with respect to the number of crossings. In particular, the drawing in Figure 4.2 contains one turn and the drawing in Figure 4.3 contains zero crossings. Evidently, the turn-optimal drawing is more readable than the crossing-optimal drawing, since the crossing-optimal drawing contains 71 turns. Interestingly, the turn-optimal drawing only contains five crossings, and is therefore close to the optimal number of crossings as well. We argue that this is most likely often the case and not a particularity of this instance for two reasons. First, a crossing-optimal drawing will most likely contain many turns, since turns of train lines can be exploited in order to “evade” other train lines such that many turns are a natural byproduct of a crossing-optimal drawing. Second, a turn-optimal drawing will most likely not contain many crossings, since few turns imply that the polylines in a turn-optimal drawing are as y-monotone as possible, thereby decreasing the “chance” that two polylines cross with each other. Note that we can construct worst-case instances, where two y-monotone polylines of size n cross $\mathcal{O}(n)$ times, which indicates that this reasoning can only be an explanation of a natural tendency of the readability of turn-optimal and crossing-optimal time-space diagrams but no general rule. However, because of this reasoning, the remainder of this thesis will focus on the minimization of the number of turns.

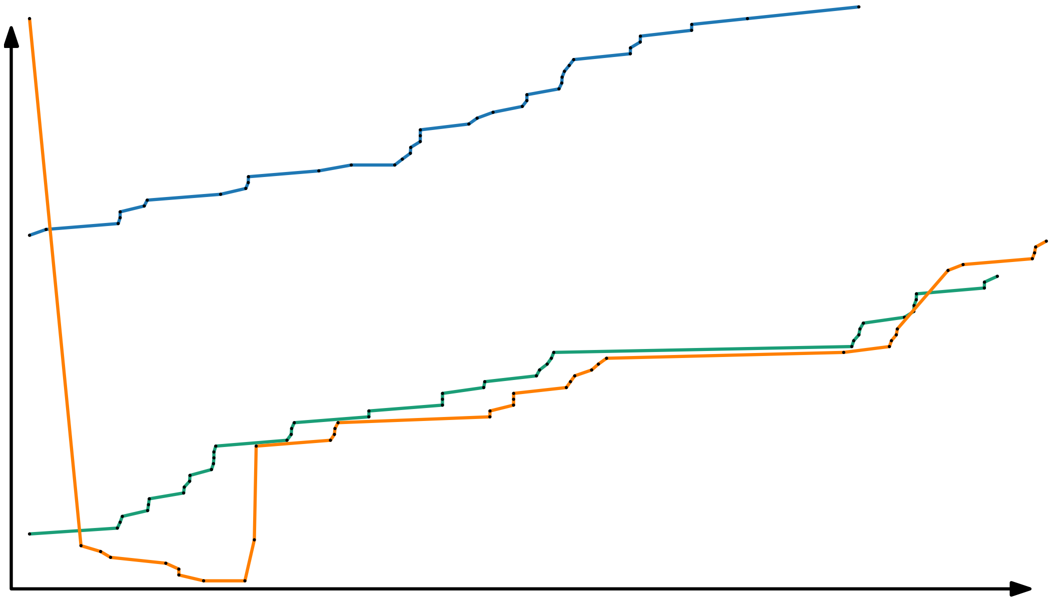

Fig. 4.2: A turn-optimal time-space diagram with one turn and five crossings.

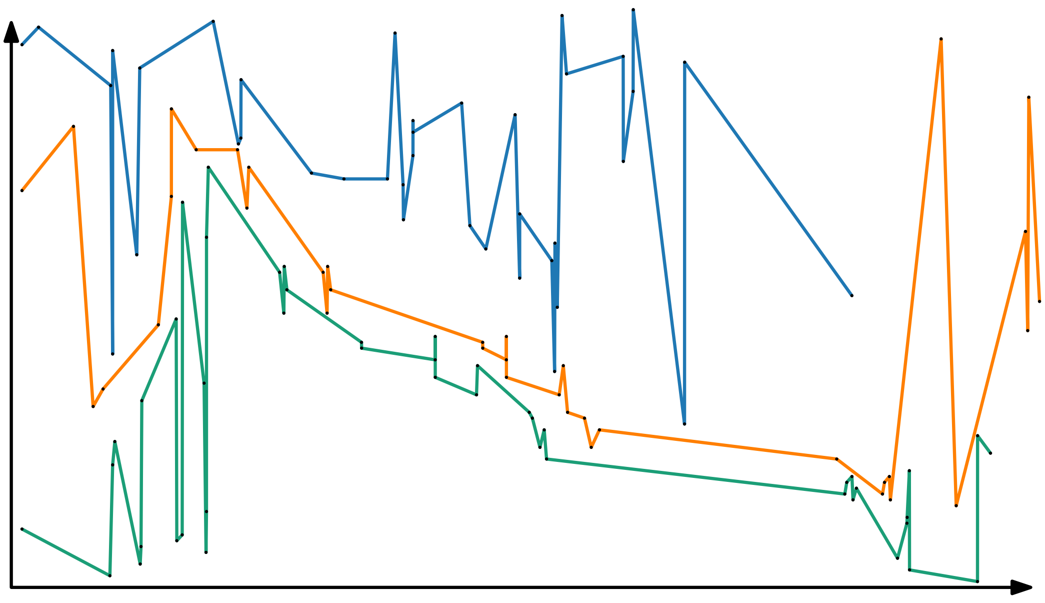

Fig. 4.3: A crossing-optimal time-space diagram of the same instance shown in Figure 4.2 with zero crossings and 71 turns.

Part II

Minimizing the Number of Turns

5 The Complexity of TSTM and Related Work

Unfortunately, finding turn-optimal time-space diagrams is also NP-hard. This follows from an alternative formulation of this problem as a BETWEENNESS problem which we propose below. However, we start with an alternative NP-hardness proof from which we can derive implications about the hardness of certain parameterized algorithms and about the inapproximability of this problem.

Theorem 11 (NP-hardness of TSTM)**.** *Given an event graph $\mathcal{E}$ and a positive integer k, it is* NP*-hard to decide whether there is a time-space diagram of $\mathcal{E}$ with at most k turns.*

Proof. Let (S, R) be a BETWEENNESS instance. We apply a simplified transformation of the instance (S, R) as shown in Theorem 7 with the difference that the gadget for a triplet $(a_i, b_i, c_i) \in R$ only needs an ordering-line Ξ_i but no spikes Λ_i^1, Λ_i^2. That is, for each triplet $(a_i, b_i, c_i) \in R$ we allocate a time interval $(i, i+1)$, in which we place a path $\Xi_i = \langle w_1, w_2, w_3 \rangle$ such that the events of the path Ξ_i appear in chronological order from w_1 to w_3 and such that w_1, w_2, w_3 occur at locations a_i, b_i, and c_i, respectively.

Claim 11.1. *Using this transformation, (S, R) has a valid ordering $\prec$ if and only if the transformed instance $(\mathcal{E}, Y)$ can be drawn as a time-space diagram Γ without any turns.*

Proof. Let (S, R) be a BETWEENNESS instance with a valid ordering $\prec$. If arranging the locations of $\mathcal{E}$ on levels from top to bottom according to $\prec$, for each triplet $(a_i, b_i, c_i) \in R$ it holds that either $a_i \prec b_i \prec c_i$ or $c_i \prec b_i \prec a_i$. By construction the ordering-line Ξ_i is a path of events with three consecutive locations a_i, b_i, c_i, thus Ξ_i is either monotonically increasing or decreasing in Γ. Therefore, no turn occurs.

Conversely, assume there is a turn-free drawing Γ of the transformed instance $(\mathcal{E}, Y)$. Let $\prec$ be the ordering of S implied by the mapping y of Γ.

Now, assume that $\prec$ is invalid. Thus, there is a triplet $(a_i, b_i, c_i) \in R$ for which b_i is not between a_i and c_i in $\prec$. Then, in the corresponding path Ξ_i the location b_i is also not between a_i and c_i in the mapping y. Therefore, $y(b_i)$ is the smallest/largest level of the three levels $y(a_i)$, $y(b_i)$, $y(c_i)$, implying a turn in Γ; a contradiction. $\triangleleft$

This shows that we can decide BETWEENNESS by testing if an optimal solution to the TSTM problem contains no turns. $\square$

Since this proof has a similar structure to the NP-hardness proof in Chapter 3, the same reasons for the para-NP-hardness and inapproximability described in Chapter 3 translate to the problem of minimizing the number of turns.

Corollary 12. *TSTM is* para-NP*-hard when parameterized by the number of turns.*

Corollary 13. *Assuming* P $\neq$ NP*, there is neither a multiplicative α-approximation with $\alpha \in \text{poly}(n)$ nor an additive β-approximation with $\beta \in \mathcal{O}(1)$ for TSTM.*

TSTM as a Betweenness Problem. The reduction shows that TSTM is very closely related to BETWEENNESS. In fact, we can also take a different perspective on the TSTM problem by formulating it in the following way:

Let $\mathcal{E}$ be an event graph. Similar to BETWEENNESS, we define a set S containing every location of $\mathcal{E}$ and for each train line $\langle e_1, \ldots, e_k \rangle$ in $\mathcal{E}$, we construct triplets $(\ell(e_{i-1}), \ell(e_i), \ell(e_{i+1}))$ for each $1 < i < k$ that correspond to BETWEENNESS constraints. We want to find an order $\prec$ of S such that a *minimum* number of constraints is violated, i.e. such that there is a minimum number of constraints (u, v, w) for which neither $u \prec v \prec w$ nor $w \prec v \prec u$ holds.

A closely related problem to this formulation is known as MAXIMUM BETWEENNESS, where the goal is to find an ordering such that the *maximum* number of constraints is satisfied. In fact, the only difference is that this problem is a maximization problem and the other problem is a minimization problem. Thus, the optimal objective function value and optimal solutions of both problems correspond to each other as the number of violated constraints is the number of constraints minus the number of satisfied constraints. Since BETWEENNESS was shown to be NP-hard by Opatrny [Opa79], MAXIMUM BETWEENNESS is also NP-hard. Further, Papadimitriou and Yannakakis [PY91] were able to show the existence of a fixed-ratio approximation algorithm among other complexity classifications, which is a notable difference to TSTM (see Corollary 13). To illustrate this difference, consider an instance with n constraints which admits an ordering of elements that satisfies all constraints. In this case, a 2-approximation algorithm for MAXIMUM BETWEENNESS can return a solution that only satisfies $n/2$ of the constraints, as the optimal value is n and therefore this solution is within a factor of 2 from the optimal value. However, in the minimization version of the problem, the optimal objective value would be 0 and therefore any multiplicative approximation algorithm for this problem would have to return an optimal solution.

A bound for an approximation algorithm for MAXIMUM BETWEENNESS was proposed by Chor and Sudan [CS98] who showed that it is NP-hard to find solutions that satisfy at least 47/48 of the constraints even if there is a solution that satisfies all constraints. Several constant factor approximations are known. For example, there is a trivial randomized algorithm that simply returns a random order as solution. In such a solution we can expect that at least 1/3 of the constraints are satisfied, since in a permutation of the elements, there are only six possible orders in which the elements of a constraint (u, v, w) can occur, where two of them satisfy the constraint. Chor and Sudan [CS98] proposed an algorithm that finds a solution that satisfies at least half of the constraints or determines that it is not possible in polynomial time via a relaxation of a semidefinite program. Later, Makarychev [Mak12] proposed a combinatorial algorithm with the same approximation factor running in linear time.

Moreover, heuristic approaches for MAXIMUM BETWEENNESS have been studied. For example, Savić [Sav09] proposed a genetic algorithm, where the genome is an integer with $n-1$ digits that encodes a possible ordering of the n elements in an instance. Their encoding enables them to use standard crossover and mutation operators that guarantee that the resulting genome remains a valid solution. Another heuristic was proposed by Filipović et al. [FKM13] who use a force-based approach, where they simulated a system of particles subject to attractive and repulsive forces whose state represents a possible solution.

An exact algorithm via a mixed-integer linear program was proposed by Savić et al. [SKMD10] and was tested on instances with up to 30 elements and 300 constraints. This formulation implicitly encodes an ordering with continuous and integer variables.

Note that there is a significant difference in perspective between TSTM and the isolated Betweenness formulation that minimizes the number of violated constraints. When only considering this isolated Betweenness formulation, important structural information of the event graph $\mathcal{E}$ and its corresponding train lines is lost, as the train lines guarantee a specific connectedness of the corresponding constraints that a general consideration of the Betweenness formulation cannot offer. But for the application of the drawing of time-space diagrams, these special cases can be used. For example, it is expected that in a TSTM instance the number of terminal vertices is low which can be exploited (for instance in Reduction Rule 6.2). For this reason we continue to consider this problem from the perspective of the TSTM problem.

6 Exact Reductions

In this chapter we describe two reduction rules to reduce the size of a given event graph $\mathcal{E}$ without sacrificing optimality specifically for the problem of finding turn-minimal time-space diagrams. These reductions have two purposes. The first reduction normalizes the event graph in order to facilitate the development of algorithms. The second reduction contracts parts of the event graph that are easy to solve independently such that only a hard kernel remains. Finally, we provide details on how the second reduction rule can be applied exhaustively.

The first reduction rule transforms saddle turns into normal turns:

Reduction Rule 6.1 (Turn Compression)**.** Let $\mathcal{E} = (V, E)$ be an event graph and let $(e_1, e_2, \dots, e_{k-1}, e_k)$ be consecutive events in a train line of z in $\mathcal{E}$ with locations $(v_1, v_2, \dots, v_k)$, where $v_2 = \dots = v_{k-1}$. Then the events $e_2, \dots, e_{k-1}$ can be contracted into a single vertex e' with location $\ell(e') = v_2$ and $t(e') = t(e_2)$.

Lemma 14. *Let $\mathcal{E}$ be an event graph and let $\mathcal{E}'$ be an event graph that resulted from applying Reduction Rule 6.1 on $\mathcal{E}$, then $\mathcal{E}$ and $\mathcal{E}'$ have the same minimum number of turns.*

Proof. Let k be the number of turns in a turn-optimal drawing Γ of $\mathcal{E}$ and let k' be the number of turns in a turn-optimal drawing Γ' of the event graph $\mathcal{E}'$. We can revert $\mathcal{E}'$ back to $\mathcal{E}$ and Γ' into a drawing of $\mathcal{E}$ by reinserting the contracted events, and by adding event vertices on the level corresponding to the locations $v_2 = \dots = v_{k-1}$. Since the reinserted vertices have the same location as e', no turn new turn can occur due to the reinsertion. Thus, the newly generated drawing of $\mathcal{E}$ based on Γ' has the same number of turns k'. Therefore, it holds that $k \leq k'$.

Similarly, contracting $e_2, \dots, e_{k-1}$ in $\mathcal{E}$ and Γ cannot introduce any turns in the resulting drawing, since $v_2 = \dots = v_{k-1}$ and the contraction leaves e' at location v_2. Consequently, a saddle turn $(v_1, v_2, \dots, v_k)$ in Γ remains a normal turn after the contraction, leading to $k' \leq k$.

As a result, the minimum number of turns in $\mathcal{E}$ is equal to the minimum number of turns in $\mathcal{E}'$.

□

With this reduction rule and the fact that it can be exhaustively applied in linear time, a case distinction between saddle and normal turns is not necessary when devising algorithms for finding turn-optimal time-space diagrams. Therefore, when designing algorithms for TSTM, we assume that the given event graph only contains normal turns.

The second reduction rule removes substructures in the instance that are easily and independently solvable. Specifically, in the location graph $\mathcal{L}$ of a given event graph $\mathcal{E}$, we can identify cases where it is straightforward to find an ordering of locations that avoids any turns. For example, consider the case depicted in Figure 6.1, where a set of trains move from some source location s to some sink location t via different routes such that there is no edge in the location graph that is traversed in opposing directions by any pair of trains and the corresponding directed component is acyclic. In this case, the subgraph between s and t can be solved independently of the rest of the graph, as the levels of s and t naturally define the space where the locations of the subgraph can be drawn. Further, since every train moves in one direction between s and t, we can use the topological ordering induced by the directions of the trains of the subgraph to obtain an ordering of the locations that avoids turns.

There are still cases, where we can easily find an ordering of the locations that avoids any turns, even though an edge e in the location graph might be traversed from both directions. If a train z moves from the sink t to the source s, its direction at an edge e might be the opposite of some other train traversing e from s to t. However, we can "change the direction" of z by letting the train move backwards in time to make its direction consistent with the direction of all other trains. This change of direction does not affect the drawing in a significant way since turns are independent of the direction of z. We capture these observations with the following terminology:

We call a component C in $\mathcal{L}$ that can be separated by a separating pair (s, t) a *transit component*, if C does not contain any terminal vertex and if the trains passing through C over s also pass over t before possibly passing s again. In particular, no train starts or ends in C and no train loops in C such that it only passes through one of the separating vertices.

A transit component C is *contractible*, if there is an acyclic ordering of the vertices in C from s to t such that for each maximal connected component of a train line of z in C, the path corresponding to the consecutive locations visited by a train is consistently directed. This allows us to formulate the following reduction rule:

Reduction Rule 6.2 (Transit Component Contraction)**.** Let $\mathcal{L}$ be the location graph of a given event graph $\mathcal{E}$ and let C be a contractible transit component that is separated by the separating pair (s, t). For each train z traversing C, contract the events in the train line of z such that the events that belong to location s, t are connected with an edge.

The following lemma shows that the number of turns in a turn-optimal drawing of a given instance graph $\mathcal{E}$ is equal to the number of turns in a turn-optimal drawing, where a contractible transit component C was contracted in $\mathcal{E}$.

Lemma 15. *Let $\mathcal{E}$ be an event graph and let $\mathcal{E}'$ be an event graph that resulted from applying Reduction Rule 6.2 on $\mathcal{E}$, then $\mathcal{E}$ and $\mathcal{E}'$ have the same minimum number of turns.*

Proof. Let k be the number of turns in a turn-optimal drawing Γ of an event graph $\mathcal{E}$ and let k' be the number of turns in a turn-optimal drawing Γ' of an event graph $\mathcal{E}$

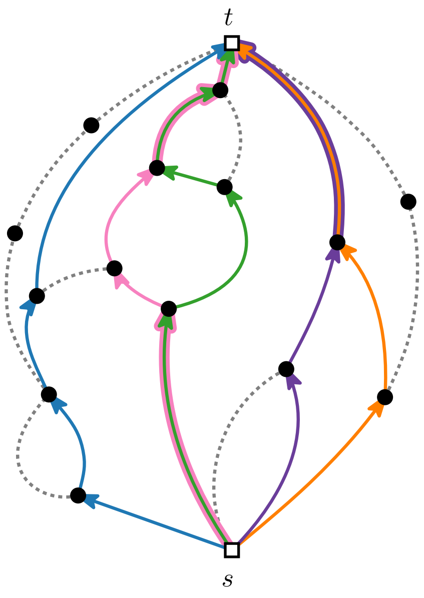

Fig. 6.1: A transit component C that is part of a larger location graph $\mathcal{L}$. A set of trains traversing C are illustrated, where some trains share an edge in C. This component C is contractible for the illustrated trains, since all trains are directed from s to t and there is an acyclic ordering.

after Reduction Rule 6.2 was applied on the contractible transit component C. Further, for the sake of simplicity, assume that any train traversing C only traverses C once. If a train z traverses C multiple times, the same arguments apply for each connected component of the train line of z going through C.

We can transform Γ' into a drawing of $\mathcal{E}$ containing C without increasing the number of turns in the following way: Let (s,t) be the separating pair that separates C with levels $y'(s)$ and $y'(t)$ in Γ' such that, without loss of generality, $y'(s) < y'(t)$. Since C is contractible, C admits an acyclic (topological) ordering of the locations in C such that the train line of every train traversing C is consistently directed from s to t. Let $\prec_C$ be such an ordering and let z be a train traversing C, where $Z' = \langle e_1, \dots, e_j \rangle$ is the component of the train line of z that traverses C such that $\ell(e_1) = s$ and $\ell(e_j) = t$. Since $\prec_C$ is a valid acyclic topological ordering, it holds that $\ell(e_i) \prec_C \ell(e_{i+1})$ for each $1 \leq i < j$. Thus, by extending y' so that each vertex v in C is assigned a level $y'(v) \in]y'(s), y'(t)[$, with $u, v \in V(C)$ and $y'(u) < y'(v)$ if and only if $u \prec_C v$, no additional turns are introduced in the transformed drawing. As a result, we obtain a drawing of $\mathcal{E}$ that includes C with the same number of turns as Γ', implying that $k \leq k'$.

Conversely, we can transform Γ into a drawing with a contracted component C with at most k turns. Again let (s,t) be the separating pair of C with levels $y(s)$ and $y(t)$ in Γ such that $y(s) < y(t)$.

First, assume that for each $v \in V(C)$ it holds that $y(s) < y(v) < y(t)$. Clearly, the contraction of C into a single edge transforms Γ into a drawing of the reduced instance with at most k turns.

Now, assume that there are vertices $v \in V(C)$ whose level is smaller (larger) than s (t) and let $L \subseteq V(C)$ be the set of vertices below s and let $U \subseteq V(C)$ be the set of vertices above t. We only describe how to handle U since L can be done analogously. Let Δ be the number of train lines with at least one vertex in U. We reorder the levels of the vertices in U according to a topological ordering of C restricted to U and move all vertices between $y(t)$ and the largest level $y(v') < y(t)$, $v' \in V(C)$. Each train line with at least one vertex in U corresponds to at least one turn in U, namely a turn at the vertex of a train line with the largest level. Therefore, moving and reordering U removes at least Δ turns. Also, the movement and reordering of U results in at most Δ turns more at t, since the only vertices that were moved are vertices in U. After moving and reordering U and L, we are in the first case and can therefore contract C.

This means that we can transform a turn-optimal drawing Γ of an event graph $\mathcal{E}$ into a drawing with a contracted component C without changing the number of turns, implying that $k' \leq k$. This concludes that Reduction Rule 6.2 is sound. $\square$

Testing for Contractible Transit Components. Given a separating pair (s, t) and a component C that is separated by (s, t), it is easy to verify if C is a transit component. Further, we can test if C is also contractible with the following algorithm that tests for the properties of a contractible transit component:

Algorithm 1: Algorithm for testing if a transit component is contractible

```
Input: A transit component $C$ with separation pair $s$, $t$, and a list
       $L = \langle E_1, \dots, E_l \rangle$ containing the subpaths of trains $z_1, \dots, z_l$ traversing
       $C$.
Output: true if $C$ is a contractible transit component false otherwise.
1 $C' \leftarrow (V(C), \varnothing)$                    // $C'$ is a directed graph.
  /* For each path $E_i$ we direct the path from $s$ to $t$ and set
     $E(C') = \bigcup E_i$.                                          */
2 foreach $E_i \in L$ do
3     if $\exists (s, v) \in E_i$ then
4         $E(C') \leftarrow E(C') \cup E_i$
5     else
6         $E(C') \leftarrow E(C') \cup \{(v, u) \mid (u, v) \in E_i\}$
  /* If the constructed graph $C'$ is acyclic, the transit component $C$
     is contractible otherwise it is not.                            */
7 return isAcyclic($C'$)
```

Applying Transit Component Contractions. In order to apply Reduction Rule 6.2 not only once but multiple times on the entire graph, a suitable set of separation pairs must be found. One possible approach is to employ SPQR-trees. Informally, an SPQR-tree represents a decomposition of a biconnected graph G alongside separation pairs in a tree-structure. The nodes of an SPQR-tree are distinguished into the types Series, Parallel, Q, and Rigid, that each correspond to a different subgraph of G which can be split by a corresponding separation pair. Thus, a SPQR-tree can be viewed as a hierarchical ordering of subgraphs of G that can be separated by a separating pair. SPQR-trees were first introduced by Di Battista and Tamassia [BT89] for representing the set of all planar embeddings of a planar biconnected graph. SPQR-trees can be computed in linear time with an algorithm by Gutwenger and Mutzel [GM00]. Given an SPQR-tree, we can test the components that are separated by the separating pairs in post-order, where we continue to test the larger component if its children are contractible transit components.

Since this approach is difficult to implement, we implemented an alternative way, where we restrict transit components to the special case of maximal chains. Chains can be found easily and have the additional advantage that they naturally contain separating pairs. Note that a focus on chains is not necessarily too restrictive since the train infrastructure naturally facilitates the formation of chains in the location graph as many events occur along a track limiting the interaction between multiple locations.

7 Integer Linear Programs

In this chapter we propose two different integer linear program formulations for finding turn-optimal time-space diagrams. The first formulation is an integer linear program that works on a concrete modelling of the levels and has the advantage that it can be combined with the formulation for minimizing the number of crossings described in Chapter 4 for a multi-objective optimization (even if the slack-level model is used). The second formulation is a 0–1 integer linear program that exploits the observation that only the ordering of the locations is important when minimizing turns but not the precise coordinate. For both formulations we assume that the event graph $\mathcal{E}$ has already been reduced according to the rules described in Chapter 6. Importantly, we assume that there are no saddle turns.

7.1 An Integer Linear Program with Concrete Levels

For this formulation we employ the same modelling of the levels as described in Chapter 4. Again, we let the level of the location v be denoted by y_v, which represents the expression 4.3. In Chapter 2 we defined that a (normal) turn occurs in a drawing Γ of $\mathcal{E}$ with respect to a tuple (e_{i-1}, e_i, e_{i+1}) of three consecutive events belonging to a train line, if their respective locations (u, v, w) are placed in Γ such that v has the largest/smallest level of the locations u, v, and w. This definition can be directly translated into an integer linear program. For each train line $\langle e_1, \dots, e_k \rangle$ of train j, we introduce a sequence of triplets T_j such that the i-th triplet $(u, v, w)_i$ in T_j corresponds to the locations of the triplet of events (e_{i-1}, e_i, e_{i+1}) in the train line of k for each $1 < i < k$. Further, let the set of these sequences T_j be denoted by $\mathcal{T}$. In the following, we only consider the case, where we want to count that location v has the largest level assigned to it. The other case, where we want to count the case where v has the smallest level is analogous.

For each triplet $(u, v, w)_i \in T_j$ we introduce an integer variable $\hat{m}_{ij}$ that is equal to $\max\{y_u, y_v, y_w\}$. Note that $\arg\max\{y_u, y_v, y_w\}$ is unique since we expect a reduced event graph that only contains normal turns. Assuming we have a proper formulation that models $\hat{m}_{ij}$ with linear constraints, we use the big-M method to force a binary variable $\hat{b}_{ij}$ to 1 if and only if $y_v = \arg\max\{y_u, y_v, y_w\}$, i.e. $\hat{b}_{ij} = 1$ if and only if a turn at event e_i of a train line j occurs. The following constraint ensures this:

$$\hat{m}_{ij} - y_v + \delta \cdot \hat{b}_{ij} \geq \delta, \tag{7.1}$$

where δ is the minimum distance between two levels (which is 1 for the TSTM problem and an arbitrary values for the slack-level model). If there is a turn at e_i, then

$\hat{m}_{ij} - y_v = 0$, which forces $\hat{b}_{ij} = 1$, as otherwise constraint 7.1 cannot be satisfied. On the other hand, if there is no turn at e_i, then either u or w is assigned the largest level of the three locations (u, v, w) and $\hat{m}_{ij} - y_v \geq \delta$. In this case $\hat{b}_{ij}$ can be either set to 0 or to 1, but since we want to minimize the number of turns, $\hat{b}_{ij}$ will always be set to 0 when possible. Since $\hat{m}_{ij} \geq y_v$ the constant δ is sufficiently large and also tight.

Now it remains to formulate constraints such that $\hat{m}_{ij} = \max\{y_u, y_v, y_w\}$. We only show how to model the expression $z = \max\{x_1, x_2\}$, as we can transform $\max\{y_u, y_v, y_w\}$ into a nested expression, where each max-operator takes only two arguments:

$$\hat{m}_{ij} = \max\{y_u, y_v, y_w\} = \max\{y_u, \max\{y_v, y_w\}\}.$$

In order to model $z = \max\{x_1, x_2\}$, we first need to make sure that z is at least x_1 and at least x_2, which can be done with the following constraints:

$$z \geq x_1 \tag{7.2}$$
$$z \geq x_2 \tag{7.3}$$

These constraints do not suffice, since z can now be larger than $\max\{x_1, x_2\}$. Therefore, we upper-bound z to *either* x_1 or x_2 via the big-M method:

$$z \leq x_1 + Y(1 - q) \tag{7.4}$$
$$z \leq x_2 + Yq, \tag{7.5}$$

where q is an auxiliary binary variable according to the big-M method. In this way, z is lower and upper bounded by $\max\{x_1, x_2\}$ and therefore models $z = \max\{x_1, x_2\}$.

These considerations yield the following integer linear program, where we use the level formulation described in Chapter 4 with

$$y_v = \sum_{l=1}^{Y} l \cdot y_v^l,$$

and where we use additional binary variables $\check{z}_{ij}$, $\hat{z}_{ij}$, $\check{p}_{ij}$, $\hat{p}_{ij}$, $\check{q}_{ij}$, $\hat{q}_{ij}$ for modelling the max-operator as described above.

$$\text{minimize} \quad \sum_{T_j \in \mathcal{T}} \sum_{t_i \in T_j} \hat{b}_{ij} + \check{b}_{ij} \tag{7.6}$$

$$\text{subject to} \quad \sum_{v \in \ell(V)} y_v^l \leq 1 \qquad \forall\, l \in [Y] \tag{7.7}$$

$$\sum_{l=1}^{Y} y_v^l = 1 \qquad \forall v \in \ell(V) \tag{7.8}$$

$$\hat{m}_{ij} - y_v + \delta \cdot \hat{b}_{ij} \geq \delta \qquad \forall T_j \in \mathcal{T}, (u,v,w)_i \in T_j \tag{7.9}$$

$$y_v - \check{m}_{ij} + \delta \cdot \check{b}_{ij} \geq \delta \qquad \forall T_j \in \mathcal{T}, (u,v,w)_i \in T_j \tag{7.10}$$

$$y_u + Y(1 - \hat{q}_{ij}) \geq \hat{m}_{ij} \qquad \forall T_j \in \mathcal{T}, (u,v,w)_i \in T_j \tag{7.11}$$

$$y_u + Y(1 - \check{q}_{ij}) \leq \check{m}_{ij} \qquad \forall T_j \in \mathcal{T}, (u,v,w)_i \in T_j \tag{7.12}$$

$$y_v + Y(1 - \hat{p}_{ij}) \geq \hat{z}_i \qquad \forall T_j \in \mathcal{T}, (u,v,w)_i \in T_j \tag{7.13}$$

$$y_v + Y(1 - \check{p}_{ij}) \leq \check{z}_i \qquad \forall T_j \in \mathcal{T}, (u,v,w)_i \in T_j \tag{7.14}$$

$$\hat{z}_{ij} + Y\hat{q}_{ij} \geq \hat{m}_{ij} \qquad \forall T_j \in \mathcal{T}, (u,v,w)_i \in T_j \tag{7.15}$$

$$\check{z}_{ij} + Y\check{q}_{ij} \leq \check{m}_{ij} \qquad \forall T_j \in \mathcal{T}, (u,v,w)_i \in T_j \tag{7.16}$$

$$y_w + Y\hat{p}_{ij} \geq \hat{z}_i \qquad \forall T_j \in \mathcal{T}, (u,v,w)_i \in T_j \tag{7.17}$$

$$y_w + Y\check{p}_{ij} \leq \check{z}_i \qquad \forall T_j \in \mathcal{T}, (u,v,w)_i \in T_j \tag{7.18}$$

$$\hat{z}_{ij} \leq \hat{m}_{ij} \qquad \forall T_j \in \mathcal{T}, (u,v,w)_i \in T_j \tag{7.19}$$

$$y_u \leq \hat{m}_{ij} \qquad \forall T_j \in \mathcal{T}, (u,v,w)_i \in T_j \tag{7.20}$$

$$\check{z}_{ij} \geq \check{m}_{ij} \qquad \forall T_j \in \mathcal{T}, (u,v,w)_i \in T_j \tag{7.21}$$

$$y_u \geq \check{m}_{ij} \qquad \forall T_j \in \mathcal{T}, (u,v,w)_i \in T_j \tag{7.22}$$

$$y_v \leq \hat{z}_{ij} \qquad \forall T_j \in \mathcal{T}, (u,v,w)_i \in T_j \tag{7.23}$$

$$y_w \leq \hat{z}_{ij} \qquad \forall T_j \in \mathcal{T}, (u,v,w)_i \in T_j \tag{7.24}$$

$$y_v \geq \check{z}_{ij} \qquad \forall T_j \in \mathcal{T}, (u,v,w)_i \in T_j \tag{7.25}$$

$$y_w \geq \check{z}_{ij} \qquad \forall T_j \in \mathcal{T}, (u,v,w)_i \in T_j \tag{7.26}$$

$$\hat{m}_{ij}, \check{m}_{ij} \in [Y] \qquad \forall T_j \in \mathcal{T}, (u,v,w)_i \in T_j \tag{7.27}$$

$$\check{z}_{ij}, \hat{z}_{ij}, \check{p}_{ij}, \hat{p}_{ij}, \check{q}_{ij}, \hat{q}_{ij} \in \{0,1\} \, \forall T_j \in \mathcal{T}, (u,v,w)_i \in T_j \tag{7.28}$$

Thus, for each triplet of consecutive events in a train line, we can verify if there is a turn with constantly many constraints and variables. We summarize this formulation with the following theorem:

Theorem 16. *Let $\mathcal{E}$ be a reduced event graph according to Reduction Rule 6.1. The integer linear program described in Equations* (7.6) *to* (7.28) *models the TSTM. This formulation requires $\mathcal{O}(n)$ integer variables, $\mathcal{O}(n + Y \cdot |\ell(V)|)$ many binary variables and $\mathcal{O}(n + Y + |\ell(V)|)$ constraints.*

7.2 A 0–1 Integer Linear Program via Orderings

Recall that only the relative ordering of locations determines the number of turns in a drawing. Therefore, it is not necessary to model the specific levels of each location, allowing us to formulate a 0–1 integer linear program for the minimization of the number of turns which simplifies the integer linear program described in Section 7.1. In particular, a modelling of the total order on the locations allows us to describe a turn with only two constraints.

For each pair of locations $u \neq v$ of the given reduced event graph $\mathcal{E}$, we model a total ordering of the locations using a binary variable x_{uv}. The variable assignment $x_{uv} = 1$ means that location u is before location v in this total ordering. In order to model a consistent ordering, the variable assignment of x_{uv} and x_{vu} need to agree, i.e. if $x_{uv} = 1$, the variable x_{vu} must be 0 for locations $u \neq v$. Thus, we need the following constraint modelling this asymmetry:

$$x_{vu} = 1 - x_{uv}. \tag{7.29}$$

Further, a total ordering requires a transitive relation of the variables. If $x_{uv} = 1$ and $x_{vw} = 1$ for three distinct locations u, v, w, the variables dictate that u is before v and v is before w. This implies that u must also be before w, therefore x_{uw} must be forced to 1. We model this with the following constraint:

$$x_{uw} \geq x_{uv} + x_{vw} - 1.$$

Since x_{wu} is determined by x_{uw} due to constraint 7.29, it suffices to add this constraint for every $\{u, w\} \in \binom{\ell(V)}{2}$ and every $v \neq u$, $v \neq w$. We denote the set of these triplets (u, v, w) with $\mathcal{Q}$.

Again, for each train line of train j, we let T_j be a sequence such that the i-th triplet $(u, v, w)_i$ in T_j corresponds to the locations of the triplet of events (e_{i-1}, e_i, e_{i+1}) in the train line of j. Further, let the set of these sequences T_j be denoted by $\mathcal{T}$. Each triplet $(u, v, w)_i$ in a sequence T_j corresponds to one possible turn. We count each turn with a binary variable b_{ij}, where $b_{ij} = 1$ means that the events corresponding to the triplet $(u, v, w)_i$ forms a turn. In an ordering, a turn corresponding to $(u, v, w)_i$ occurs, if v is not between u and w. This can be counted with a constraint similar to the transitivity constraint. If v is not between u and w, then v is the smallest or the largest element of the three locations with respect to the total ordering. Thus, the following two constraints, will require b_{ij} to be 1, if there is a turn:

$$b_{ij} \geq x_{vu} + x_{vw} - 1 \tag{7.30}$$
$$b_{ij} \geq x_{uv} + x_{wv} - 1, \tag{7.31}$$

where constraint 7.30 corresponds to the case, where v is the smallest element and constraint 7.31 models the case, where v is the largest element of the three locations u, v, and w.

Thus, we can write the complete binary integer linear program in the following way:

$$\begin{aligned}
&\text{minimize} && \sum_{T_j \in \mathcal{T}} \sum_{t_i \in T_j} b_{ij} && && (7.32)\\
&\text{subject to} && x_{vu} = 1 - x_{uv} && \forall u, v \in \ell(V), u \neq v && (7.33)\\
& && x_{uw} \geq x_{uv} + x_{vw} - 1 && \forall (u, w, v) \in \mathcal{Q} && (7.34)\\
& && b_{ij} \geq x_{vu} + x_{vw} - 1 && \forall T_j \in \mathcal{T}, (u, v, w)_i \in T_j && (7.35)\\
& && b_{ij} \geq x_{uv} + x_{wv} - 1 && \forall T_j \in \mathcal{T}, (u, v, w)_i \in T_j && (7.36)\\
& && x_{uv} \in \{0, 1\} && \forall u, v \in \ell(V), u \neq v && (7.37)\\
& && b_{ij} \in \{0, 1\} && \forall T_j \in \mathcal{T}, t_i \in T_j && (7.38)
\end{aligned}$$

Note that we can easily obtain a concrete level assignment with the bijective mapping $f \colon \ell(V) \to [Y]$ defined as

$$f(u) = 1 + \sum_{\substack{v \in \ell(V)\\ u \neq v}} x_{uv}.$$

Again, we summarize this formulation in the following lemma:

Theorem 17. *Let $\mathcal{E}$ be an event graph reduced by Reduction Rule 6.1. The 0–1 integer linear program described in Equations* (7.32) *to* (7.38) *models the TSTM using $\mathcal{O}(|\ell(V)|^2)$ binary variables and $\mathcal{O}(|\ell(V)|^3)$ constraints.*

The advantage of this formulation is that it only relies on binary variables. However, the number of variables and constraints increases significantly compared to the formulation described in Section 7.1. In an effort to handle the increase in the number of constraints, we propose an algorithm that tries to reduce the number of transitivity constraints 7.34 that need to be considered in order to find an optimal solution.

The algorithm is a variant of the cutting-plane method. The cutting plane method, first described by Gomory [Gom58, Gom10], is a general algorithm to solve (mixed-) integer linear programs. This algorithm iteratively solves a relaxation of the original problem and adds cuts to the relaxation that remove non-integer solutions. The idea of using cuts to separate a solution in a relaxed version of the original problem can also be used when handling problems with a large number of constraints.

In this variant, the algorithm starts with a small set of the original constraints. During the iterative solving process, the current problem formulation containing only a subset of constraints is solved. After a solution is found, the algorithm determines a set of violated constraints and adds them to the current formulation until an optimal solution is found which does not violate any of the original constraints.

This strategy proved to be very successful for finding optimal TSP-tours with formulations containing exponentially many constraints, and is usually combined with branch-and-bound methods [GP79a, GP79b, GH91].

We employ a similar approach for our integer linear program, where we disregard the transitivity constraints 7.34 at the beginning of the solution process and add a small set of violated transitivity constraints with the help of a subroutine which is called, when a solution is found.

A Subroutine for Finding Violated Transitivity Constraints. Given a current solution (x, b) the subroutine constructs a tournament graph from the variables x which model the total ordering of the locations in the following way: Let G' be a directed graph, whose vertices are the locations of $\mathcal{E}$, and where (u, v) is an edge if $x_{uv} = 1$. Due to the constraint that $x_{uv} = 1 - x_{vu}$, either the edge (u, v) or the edge (v, u) is in G', and therefore G' is a tournament graph.

The tournament graph G' helps us find violated constraints since any cycle in G' implies that the ordering is contradictory:

Observation 18. *A violated transitivity constraint corresponds to a cycle in G', implying that if G' is acyclic, the solution (x, b) is valid.*

Thus, the subroutine can find violated constraints by detecting cycles in G' and can also prove that (x, b) is an optimal solution by determining that G' is acyclic. Note that since G' is a tournament graph, it suffices to detect cycles of length 3 because of the following well-known property of tournament graphs, which we prove for the sake of completeness:

Proposition 19. *Let G' be a tournament graph. If there is a cycle in G', then there is a cycle of length 3 in G'.*

Proof. Let C be the shortest cycle in G'. If C has length 3, then we are done. Otherwise, C has length at least 4. Let $v, w \in C$ such that w is neither the direct predecessor nor the direct successor of v on C. Since G' is a tournament graph, there is either $(v, w) \in E'$ or $(w, v) \in E'$. In any case, C contains a chord which implies a shorter cycle than C which is a contradiction to the assumption that C is the shortest cycle. $\square$

Therefore, it suffices to detect cycles of length 3 to determine a subset of violated constraints. In order to find cycles of length 3, we can enumerate cycles of bounded length with algorithms such proposed in [SL76, LT82]. Since there can be many cycles, we restrict the number of detected cycles by a parameter k. For each detected cycle $\langle u, v, w \rangle$, we add the constraint $x_{uw} \geq x_{uv} + x_{vw} - 1$ to the model.

8 Decomposition-based Methods

In this chapter we consider algorithms that work on decompositions of auxiliary graphs constructed from a given event graph. Again, we assume that the event graph only contains normal turns. We start with proposing a fixed parameterized tractable algorithm for the TSTM problem parameterized by the treewidth of the *augmented location graph* (defined below) $\mathcal{L}'$ of a given event graph $\mathcal{E}$. Afterwards, we use decompositions and separations of the location graph to devise a framework that can be used for exact and heuristic algorithms.

8.1 A Parameterized Algorithm

The algorithm is a dynamic program on a *nice tree decomposition* of the augmented location graph $\mathcal{L}'$. The augmented location graph is defined as follows:

Definition 20 (Augmented Location Graph)**.** *Let $\mathcal{E}$ be an event graph. The* augmented location graph *$\mathcal{L}'$ of $\mathcal{E}$ is a supergraph of the location graph $\mathcal{L}$ of $\mathcal{E}$ containing additional edges $\{\ell(e_{i-1}), \ell(e_{i+1})\}$ for each consecutive events (e_{i-1}, e_i, e_{i+1}) in a train line in $\mathcal{E}$.*

The augmented graph $\mathcal{L}'$ has the crucial property that three consecutive events e_{i-1}, e_i,e_{i+1} of a train line whose locations can cause a potential turn form a triangle in $\mathcal{L}'$. Before we describe the dynamic program, we define a (nice) tree decomposition of a graph G. For the sake of clarity, we refer to elements in $V(G)$ of a graph G that is decomposed into a structure T as vertices, and refer to elements in $V(T)$ of the decomposed structure T as nodes.

Tree decompositions. Intuitively, a tree decomposition is a decomposition of a graph G into a tree T which gives structural information about the separability of G. The treewidth of a graph G is a measure that captures how similar a graph G is to a tree. For instance, the treewidth of a tree is 1, the treewidth of a square grid graph is $\lceil\sqrt{n}\rceil$, and the treewidth of a complete graph is $n-1$. More formally, a *tree decomposition* $\mathcal{T} = (T, \{X_t\}_{t\in V(T)})$ of G consists of a tree T, such that

(T1) every node $t \in V(T)$ has an associated bag $X_t \subseteq V$ such that the union of all bags is equal to $V(G)$,

(T2) for each edge $\{u, v\} \in E(G)$, there has to exist at least one bag X_t with $u, v \in X_t$,

(T3) and for each vertex $v \in V(G)$, the nodes whose bags contain v induce a connected subgraph of T.

The *width* of a tree decomposition is defined as $\max\{|X_t| \mid t \in V(T)\}-1$ and the *treewidth* $\mathrm{tw}(G)$ of a graph G is the smallest value, such that there exists a tree decomposition of G with this width. See Figure 8.1 for an example of a tree decomposition.

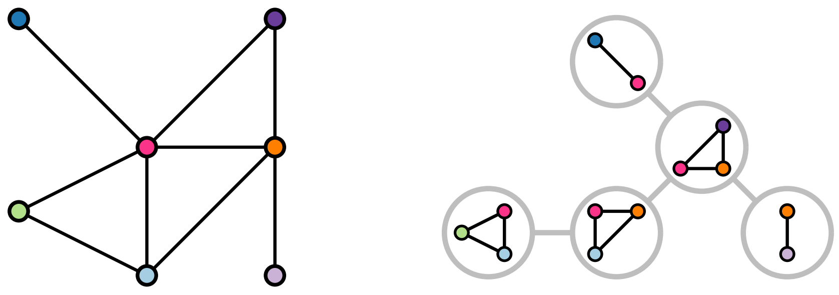

Fig. 8.1: A tree decomposition (right) of a graph (left). The width of this tree decomposition is 2 since the maximum amount of vertices in a bag is three.

We call a tree decomposition $\mathcal{T} = (T, \{X_t\}_{t\in V(T)})$ *nice*, if T is rooted at a leaf node r, the leaf nodes in T have empty bags, and all other nodes are one of the three following different types. A node t is of type *introduce* if t has exactly one child c, and $X_t = X_c \cup \{v\}$ for some $v \notin X_c$. Similarly, a node t is of type *forget*, if t has exactly one child c, and $X_c = X_t \cup \{v\}$ for some $v \notin X_t$. The third type is a *join* node, which is a node t with two children $i, j \in V(T)$ whose bags contain the same vertices of V, i.e., $X_t = X_i = X_j$. Further, we require that the root node r is of type forget.

Any tree decomposition has two well known properties that are important for our purposes:

(P1) Consider an arbitrary edge $\{a, b\}$ in T. The deletion of $\{a, b\}$ creates two connected components T_a, T_b of T, where T_a contains a and T_b contains b. This corresponds to a separation $(A, B) = (\bigcup_{t\in V(T_a)} X_t, \bigcup_{t\in V(T_b)} X_t)$ of G with separator $X_a \cap X_b$. In particular, $A \cap B = X_a \cap X_b$ and there is no edge between a vertex in $A \setminus X_a$ and a vertex in $B \setminus X_b$.

(P2) For every clique K of G, there is a bag X_t in a tree decomposition $\mathcal{T}$ such that $V(K) \subseteq X_t$.

Several algorithms are known for computing a tree decomposition of a graph G. An exact FPT algorithm with respect to the natural parameter $\mathrm{tw}(G)$ was proposed by Bodlaender [Bod96]. While this algorithm has a runtime that is linear in the input size, the runtime also depends on the factor $\mathrm{tw}(G)^{\mathcal{O}(\mathrm{tw}(G)^3)}$; a factor that dominates in most applications. Therefore, approximation algorithms for finding a tree decomposition are of interest. One such approximation algorithm with FPT runtime with smaller runtime factor depending on $\mathrm{tw}(G)$, but a factor of $\mathcal{O}(n^2)$ on the input size is the algorithm by Robertson and Seymour [RS86]. A survey about exact algorithms and heuristics for computing an (optimal) tree decomposition was written by Bodlaender [Bod05]. Notably, there are some simple and efficient heuristics such as the Min-Fill-In or the Min-Degree heuristic that perform well in practice according to this survey.

Given an arbitrary tree decomposition, a nice tree decomposition of the same graph can be computed in polynomial time preserving the width of the given decomposition such that this nice tree decomposition contains $\mathcal{O}(\text{tw}(G) \cdot n)$ many nodes [Bod98]. Therefore, whenever we design an FPT algorithm with respect to the parameter $\text{tw}(G)$, we can assume that we have a nice tree decomposition of width $\text{tw}(G)$.

Theorem 21. *Let $\mathcal{L}'$ be the augmented location graph of a given event graph $\mathcal{E}$. There is a parameterized algorithm with respect to the treewidth $\text{tw}(\mathcal{L}')$ for finding a turn-optimal time-space diagram of $\mathcal{E}$.*

Proof. We begin with introducing notation. In the following we refer to the time-space diagram simply as drawing and for the sake of brevity we say "a drawing of $\mathcal{L}$" and mean the drawing of $\mathcal{E}$ restricted to the locations contained in $\mathcal{L}$. For some $t \in V(T)$, we define the subgraph $\mathcal{L}'_t$ of $\mathcal{L}'$ to be the graph induced by the union of bags contained in the subtree of T rooted at t. For instance, the induced graph $\mathcal{L}'_r$ with respect to the subtree rooted at the root node r is precisely $\mathcal{L}'$.

Further, let π^t be an order of the vertices in the bag X_t. We say that a drawing *respects* π^t if the vertices in X_t are drawn such that for all $u, v \in X_t$ with $u \prec_{\pi_t} v$ vertex u is drawn above vertex v. With $\pi^t_{v \to i}$ we denote the order π^t which is extended by a vertex v such that v has $\text{rank}(v) = i$ within the extended order $\pi^t_{v \to i}$. Lastly, we define $b(v, \pi^t)$ to be the number of turns, where v is one of the three locations (v_{i-1}, v_i, v_{i+1}) of a turn in a drawing of $\mathcal{L}'[X_t]$ respecting π^t. Similarly, we write $b(X_t, \pi^t)$ for the total number of turns occurring in a drawing of $\mathcal{L}'[X_t]$ respecting the order π^t, where all three locations (v_{i-1}, v_i, v_{i+1}) of a turn are contained in X_t. Note that each drawing of $\mathcal{L}'[X_t]$ respecting π^t has the same number of turns since π^t dictates an ordering on every vertex in X_t.

Now, let $\mathcal{L}'$ be a given augmented location graph and let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a nice tree decomposition of $\mathcal{L}'$ rooted at $r \in V(T)$. We define $D[t, \pi^t]$ to be the number of turns in a turn-optimal drawing of $\mathcal{L}'_t$ respecting the order π^t. Therefore, $D[r, \pi^r]$ with the unique empty order π^r corresponds to the number of turns of a turn-optimal drawing of $\mathcal{E}$, since r is the root of T and r is associated with a leaf bag $X_r = \varnothing$.

We show how $D[t, \pi^t]$ can be calculated by the following recursive formulas depending on the node type of t. Based on this recursive formulation, the actual optimal ordering of the locations in $\mathcal{E}$ can be extracted via a straightforward backtracking algorithm.

Leave node (except root): Since the associated bag X_t of a leaf node t is empty, $\mathcal{L}'_t$ is an empty graph and therefore the minimum number of turns of a turn-optimal drawing of $\mathcal{L}'_t$ is $D[t, \pi^t] = 0$ with an empty ordering π^t.

Introduce node: Let v be the vertex that has been introduced in node t, and let c be the only child of t, then

$$D[t, \pi^t] = D[c, \pi^t|_c] + b(v, \pi^t).$$

Inductively, $D[c, \pi^t|_c]$ corresponds to number of turns of a turn-optimal drawing of $\mathcal{L}'_c$ respecting the order π^t restricted to vertices in X_c. The node t extends the graph $\mathcal{L}'_c$ by the vertex v, introducing edges between v and vertices $N(v) \cap X_t$ that can cause turns including v. These turns are counted by $b(v, \pi^t)$. Since π^t dictates the relative position of every vertex in $N[v]$, a turn-optimal drawing of $\mathcal{L}'_t$ respecting π^t must contain every newly introduced turn.

Note that we count a turn at most once in this setting: First, the vertex v is introduced exactly once in $\mathcal{L}'_t$ by the definition of a nice tree decomposition. Further, by the definition of b, we only count a turn if one location of its consecutive events e_{i-1}, e_i, e_{i+1} is v. Therefore, during the computation of $D[c, \pi^t]$, the turns involving v have not been counted previously.

Forget node: Let $X_t = X_c \setminus \{v\}$ be the bag of t, where v is the vertex that has been forgotten in node t and where c is the only child of t, then

$$D[t, \pi^t] = \min\{D[c, \pi^t_{v \to i}] \mid i = 1, \dots, |X_c|\}.$$

At node t, we remove vertex v from the bag X_c, therefore $\mathcal{L}'_t = \mathcal{L}'_c$. The ordering π^t dictates the drawing for $\mathcal{L}'[X_c]$ in a turn-optimal drawing in $\mathcal{L}'_t$ except for v. Thus, the number of turns of a turn-optimal drawing of $\mathcal{L}'_t$ respecting π^t must be a turn-optimal drawing in $\mathcal{L}'_c$ respecting the order π^t, where v is inserted into the order π^t for some $\text{rank}(v) = i$.

Note that every turn involved in a turn-optimal drawing respecting $\pi^t_{v \to i}$ for an optimal i is accounted for precisely once since v only be can forgotten once. Since v was forgotten, X_t is a separating set that separates v from every other vertex in $\mathcal{L}' \setminus \mathcal{L}'_t$, implying that the neighbourhood of v was already processed in $\mathcal{L}'_c$. Further, due to property (P2) of a tree decomposition and the construction of the augmented location graph $\mathcal{L}'$, we know that there is an already processed bag that contains all three locations of consecutive events e_{i-1}, e_i, e_{i-1} that can cause a turn.

Join node: Let i and j be the two children of node t, then we can calculate the number of turns in a drawing of G_t respecting π^t by

$$D[t, \pi^t] = D[i, \pi^t] + D[j, \pi^t] - b(X_t, \pi^t).$$

At a join node, two independent connected components of T are joined, where $\mathcal{L}'_i$ and $\mathcal{L}'_j$ only have vertices X_t in common. By induction, $D[i, \pi^t]$ and $D[j, \pi^t]$ contain the number of turns in a turn-optimal drawing in $\mathcal{L}'_i$ and a turn-optimal drawing in $\mathcal{L}'_j$, where both drawings respect π^t. Consequently, by summing the number of turns in both drawings $\mathcal{L}'_i$ and $\mathcal{L}'_j$, we count turns occurring in X_t twice. Therefore, we need to subtract turns whose three corresponding vertices are contained in X_t. Further, note that no new vertex is introduced in a join node, thus no new turn can occur.

With the description of the recursive formulation of $D[t, \pi^t]$, we have shown that a turn (v_{i-1}, v_i, v_{i+1}) in a turn-optimal drawing is counted at least once in an introduce node of the last introduced vertex v of (v_{i-1}, v_i, v_{i+1}). We have also argued in the description of the forget node that a turn at v is counted at most once. Therefore, we count every turn exactly once in a drawing calculated by $D[r, \varnothing]$, concluding the correctness of the algorithm.

As for the runtime, note that $b(v, \pi^t)$ can be computed in $\mathcal{O}(|X_t|^2)$ time by annotating every clique $\{u, v, w\}$ in $\mathcal{L}'$ corresponding to a potential turn by the number of distinct consecutive event triplets mapping to locations u, v, and w. Since v is involved in each clique, we can enumerate every clique $\{u, v, w\}$ in $\mathcal{O}(|X_t|^2)$ time and due to the annotation, a clique can be processed in constant time. In order to count the total number of turns $b(X_t, \pi^t)$ in a bag X_t, we need $\mathcal{O}(|X_t|^3)$ time. Assuming the algorithm operates on a nice tree decomposition $\mathcal{T}$ of width $\mathrm{tw}(\mathcal{L}')$, there are $\mathcal{O}(\mathrm{tw}(\mathcal{L}') \cdot n)$ many bags in $\mathcal{T}$. For a node t in the tree decomposition, we have to guess $\mathcal{O}((\mathrm{tw}(\mathcal{L}') + 1)!)$ many orders. If t is a leaf node, it can be processed in constant time. Given an order π^t, we can compute any introduce, forget, or join node in $\mathcal{O}(\mathrm{tw}(\mathcal{L}')^3)$ time, yielding an overall runtime of $\mathcal{O}((\mathrm{tw}(\mathcal{L}') + 1)! \cdot \mathrm{tw}(\mathcal{L}')^4 \cdot n)$. □

The treewidth of the augmented location graph $\mathcal{L}'$ can be significantly larger than the treewidth of the corresponding location graph $\mathcal{L}$. If trains only transition from one neighbour $u \in N(v)$ through v to a small subset of neighbours $S(u) \subseteq N(v)$, the treewidth increases only slightly. However, if $S(u)$ is large, the treewidth increases substantially. In the worst-case scenario, where $S(u) = N(v)$, cliques of size $\deg(v)$ are formed, as illustrated in Figure 8.2, implying a treewidth $\mathrm{tw}(\mathcal{L}') \geq \Delta(\mathcal{L})$ due to property (P2).

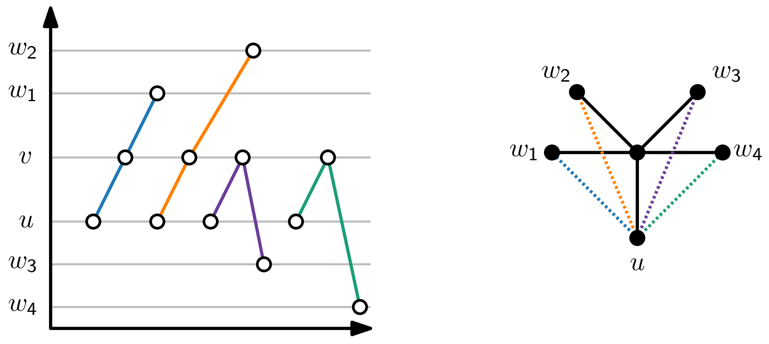

Fig. 8.2: Example of the neighbourhood of a vertex $v \in V(\mathcal{L})$. In this case, there is a train for each neighbour $N(v) \setminus \{u\}$ moving from u to $w_i \in N(v) \setminus \{u\}$, as shown in the excerpt of a time-space diagram (left). This results in edges between u and every neighbour $N(v) \setminus \{u\}$ (right). If this is the case for every neighbour $w \in N(v)$, then $N(v)$ forms a clique in $\mathcal{L}'$.

We can use the general idea of using separators and decompositions to devise a framework for algorithms which we investigate in the remainder of this chapter.

8.2 Divide-and-Conquer through Separators

Consider the location graph $\mathcal{L}$ of an event graph $\mathcal{E}$. Suppose $\mathcal{L}$ contains a bridge, i.e. there is an edge $\{u, v\} \in E'$ whose removal results into disconnecting $\mathcal{L}$ into components C_u and C_v. Every interaction between components C_u and C_v must therefore go through this edge. Thus, edge $\{u, v\}$ also separates a drawing of $\mathcal{E}$ into drawings of C_u and C_v, where the only edges in the drawing between C_u and C_v are drawn between location u and v; see Figure 8.3 for an illustration. As a consequence, a turn-optimal drawing of $\mathcal{E}$ can be computed by computing turn-optimal drawings of C_u and C_v, where the only dependence between the computation of C_u and C_v is the relative placement of the locations u and v. Therefore, if the optimal relative placement of u and v is known, then a drawing of C_u and C_v can be computed independently and later be combined into a drawing of $\mathcal{E}$.

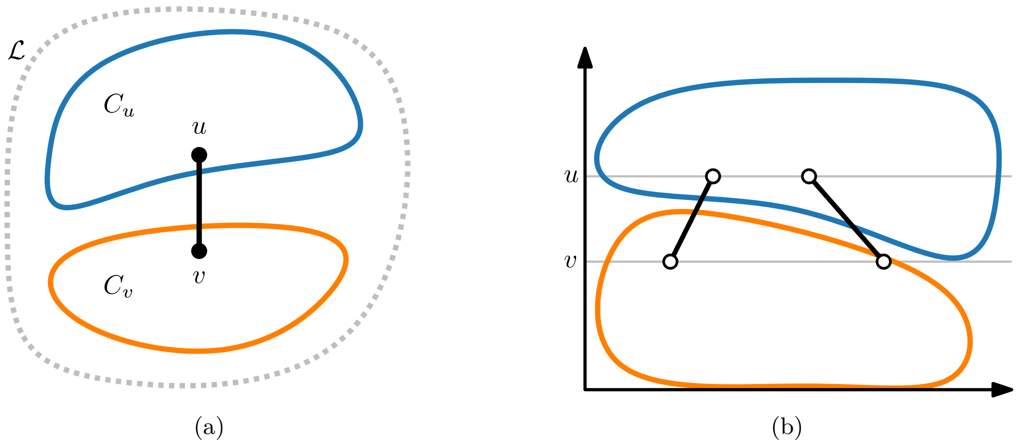

Fig. 8.3: Figure 8.3a shows a location graph $\mathcal{L}$ that can be separated by removing the bridge $\{u, v\}$. A bridge also naturally separates a time-space diagram of $\mathcal{E}$, where the only edges in the time-space diagram between locations in C_u and locations in C_v are between the locations u and v. Note that the intervals formed by the levels of locations in C_u and C_v may overlap.

We can generalize this idea to cuts of arbitrary size. Let (S, T) be a cut of $\mathcal{L}$, and let $\mathcal{C}$ be the set of vertices in S (or T) with at least one neighbour in T (or S). Suppose we know an optimal order of every vertex in $\mathcal{C}$ in a turn-optimal drawing of $\mathcal{E}$, then the interaction between S and T can be fixed to this ordering, and we can compute a turn-optimal drawing of S and T with respect to this order independently. Since the cut edges between S and T can be involved in turns, where the corresponding other edge is either in S or in T, we need to make sure that we also consider these edges in the computation of optimal drawings of both S and T. Therefore, given an optimal order of the vertices in $\mathcal{C}$, the problem of finding a turn-optimal drawing of $\mathcal{E}$ reduces to finding an optimal drawing of $S \cup \mathcal{C}$ and $T \cup \mathcal{C}$ respecting the given order of $\mathcal{C}$. We can repeat this procedure on the reduced instances until the reduced graphs have constant size, where we can easily find a turn-optimal drawing using an exact algorithm. Since we do not know which order of the vertices in $\mathcal{C}$ are optimal, we try every possible ordering. In order to correctly combine two turn-optimal drawings recusrively, we have to contract the vertices of $\mathcal{C}$ in the reduced instances $S \cup \mathcal{C}$ and $T \cup \mathcal{C}$, for reasons that become apparent later. We capture this idea in Algorithm 2.

Before we prove the correctness of this algorithm, we show how the `merge` subroutine can be implemented correctly. The routine has to merge two orderings o_1, o_2 with respect to a fixed order $\pi = \langle v_1, \ldots, v_p \rangle$ such that the merged order o contains all elements in o_1 and o_2 and both orderings are preserved.

For an ordering o, let $o(v) = \operatorname{rank}(v)$ in o and let $v = o[i]$ be the element v with $\operatorname{rank}(v) = i$ in the ordering o. For convenience, we write $o[i, j]$ for the subordering in o of elements $u \in o[i, j]$ satisfying $i \leq o(u) < j$ and $o[i, \cdot]$ for all elements with rank at least i in the ordering o. Symmetrically, we write $o[\cdot, i]$ for all elements $u \in o[\cdot, i]$ with rank $1 \leq \operatorname{rank}(u) < i$.

The general idea behind the `merge` routine is to concatenate the suborderings $o_1[i_1, j_1]$ and $o_2[i_2, j_2]$, where i_1, j_1, i_2, j_2 are the ranks of two consecutive fixed elements v_i, $v_{i+1} \in \pi$ in the orderings o_1 and o_2, respectively. We denote the concatenation of two orderings o_1, o_2 as $o_1 + o_2$, which means that the orderings of o_1 and o_2 are preserved and that every element in o_2 is bigger than every element in o_1. Algorithm 3 shows the `merge` procedure.

Lemma 22. *Given two orderings o_1 and o_2 respecting an ordering $\pi = \langle v_1, \ldots, v_p \rangle$, such that o_1 and o_2 only share elements of π, Algorithm 3 merges both orderings o_1 and o_2 preserving their respective order.*

Proof. First, note that the concatenation of two disjoint suborderings of o_1, o_2 cannot violate the ordering of o_1 or o_2. Further, note that it suffices to show that o_1 and o_2 are preserved as their preservation implies that π is also preserved because o_1 and o_2 already respect π. For the sake of simplicity, we assume that there is an element v_0 with $\operatorname{rank}(v_0) = 0$ in both o_1 and o_2. We show the correctness of Algorithm 3 with two invariants:

(I1) At the beginning of the i-th iteration of the for-loop at Line 4, o contains all elements of $o_1[\cdot, i_1]$ and $o_2[\cdot, i_2]$. In particular, it holds that $i_1 = o_1(v_{i-1}) + 1$ and $i_2 = o_2(v_{i-1}) + 1$.

(I2) At the beginning of the i-th iteration of the for-loop at Line 4, the orderings of $o_1[\cdot, i_1]$ and $o_2[\cdot, i_2]$ are preserved in o.

At the beginning of the first iteration, i_1 and i_2 are initialized with 1. By assumption there is an element v_0 with $\operatorname{rank}(v_0) = 0$ in o_1 and o_2. Since $o_1[\cdot, i_1]$ and $o_2[\cdot, i_2]$ contain only elements with rank at least one, both suborderings do not contain any element. Since o is also empty, (I1) and (I2) hold.

Now, consider the beginning of the i-th iteration of the for-loop in Lines 4 to 7. By the loop invariant (I1) all elements of $o_1[\cdot, i_1]$ and $o_2[\cdot, i_2]$ are contained in o, and $i_1 = o_1(v_{i-1}) + 1$, $i_2 = o_2(v_{i-1}) + 1$. Thus, $o_1[i_1, o_1(v_i)]$ and $o_2[i_2, o_2(v_i)]$ are disjoint and therefore the concatenation in Line 5 safely adds $o_1[i_1, o_1(v_i)]$, $o_2[i_2, o_2(v_i)]$ and v_i

Algorithm 2: Divide-and-Conquer algorithm on separators for a given event graph $\mathcal{E}$

```
/* Let $\Lambda(o)$ be a helper function that counts the number of turns of
   an ordering $o$.                                                     */
1  fn DivideAndConquer(location graph $\mathcal{L} = (V, E)$, fixed relative orders $\Pi = \varnothing$)
2      if $|V|$ has constant size of at most $k$ then
           /* We count the locations of contracted vertices as separate
              locations.  If the location graph has constant size, we
              can use an exact algorithm for finding a turn-optimal
              ordering $\prec$ on the event graph induced by $\mathcal{L}$.          */
3          Undo contractions and construct event graph $\mathcal{E}'$ induced by $\mathcal{L}$ from $\mathcal{E}$
4          return solveExact($\mathcal{E}'$, $\Pi$)
       /* Separate the location graph into partitions $S$, $T$ with the
          separation set $\mathcal{C}$.                                       */
5      $S, T, \mathcal{C} \leftarrow$ separate($\mathcal{L}$)
       /* Construct subgraphs, where the vertices of the cut are
          contracted into a single vertex in each subgraph.          */
6      $G_1 \leftarrow \mathcal{L}[S \cup \mathcal{C}]/\mathcal{C}$
7      $G_2 \leftarrow \mathcal{L}[T \cup \mathcal{C}]/\mathcal{C}$
8      $o^\star \leftarrow \varnothing$               // Current optimal ordering, where $\Lambda(\varnothing) = \infty$
       /* The function permute returns an iterator of all necessary
          orderings of $\mathcal{C}$ that might be in a turn-optimal solution.   */
9      foreach $\pi \in$ permute($\mathcal{C}$) do
10         $o_1 \leftarrow$ DivideAndConquer($G_1$, $\Pi \cup \{\pi\}$)
11         $o_2 \leftarrow$ DivideAndConquer($G_2$ $\Pi \cup \{\pi\}$)
           /* Merge the orderings $o_1$ and $o_2$ into a consistent ordering $o$
              with respect to the fixed relative orders.                 */
12         $o \leftarrow$ merge($o_1$, $o_2$, $\pi$)
13         if $\Lambda(o) < \Lambda(o^\star)$ then
14             $o^\star \leftarrow o$
15     return $o^\star$
```

Algorithm 3: Algorithm for merging two turn-optimal orderings with respect to a fixed order π.

```
Input: A fixed order $\pi = \langle v_1, \dots, v_p \rangle$, two orderings $o_1$, $o_2$ respecting the fixed
       order $\pi$.
Output: A merged order $o$ containing every element in $o_1$ and $o_2$ such that
        both orderings $o_1$ and $o_2$ are preserved.
1 $i_1 \leftarrow 1$
2 $i_2 \leftarrow 1$
3 $o \leftarrow \varnothing$                                  // Declare an empty ordering.
4 foreach $v_i \in \pi$ do
      /* Concatenate the suborderings between the previous fixed
         element and the current fixed element $v$ and add $v$ to $o$.     */
5     $o \leftarrow o + o_1[i_1, o_1(v_i)] + o_2[i_2, o_2(v_i)] + \langle v_i \rangle$
6     $i_1 \leftarrow o_1(v_i) + 1$
7     $i_2 \leftarrow o_2(v_i) + 1$
  /* Add all remaining elements in $o_1$ and $o_2$.                        */
8 $o \leftarrow o + o_1[i_1, \cdot] + o_2[i_2, \cdot]$
9 return $o$
```

to o. Afterwards, Lines 6 and 7 increase i_1 to $o_1(v_i)+1$ and i_2 to $o_2(v_i)+1$. This means that all elements with rank $i_1, \dots, o_1(v_i)-1$ in o_1 are added to o via $o_1[i_1, o_1(v_i)]$ and all elements with rank $i_2, \dots, o_2(v_i)-1$ in o_2 are added to o via $o_2[i_2, o_2(v_i)]$. Since v_i is also added to o, the ordering o indeed contains all elements of $o_1[\cdot, i_1]$ and all elements of $o_2[\cdot, i_2]$. Therefore, (I1) is maintained. Similarly, since concatenation preserves the orders and $i_1 = o_1(v_{i-1})+1$, $i_2 = o_2(v_{i-1})+1$ in Line 5, (I2) is also maintained.

Thus, when the for-loop terminates at the $(p+1)$-th iteration, both invariants hold and precisely all elements in o_1 with rank at most $o_1(v_p)$ and all elements in o_2 with rank at most $o_2(v_p)$ are contained in o and o preserves the orderings of o_1, o_2. Before the algorithm terminates, Line 8 is executed which safely concatenates the remaining elements in o_1 and o_2. Therefore, Algorithm 3 correctly merges the orderings o_1 and o_2. □

Theorem 23. *Given an event graph $\mathcal{E}$ and the location graph $\mathcal{L}$ of $\mathcal{E}$, Algorithm 2 returns a turn-optimal ordering with respect to fixed orders* Π. *In particular, if* $\Pi = \varnothing$, *then Algorithm 2 returns a turn-optimal ordering of $\mathcal{E}$.*

Proof. For location graphs $\mathcal{L}$ with at most k vertices, we return a turn-optimal ordering with respect to Π by the definition of `solveExact`. Therefore, the algorithm is correct for event graphs with at most k locations.

Let $\mathcal{L}$ be a location graph with more than k vertices. Then, $\mathcal{L}$ is decomposed into two subgraphs $G_1 = \mathcal{L}[S \cup \mathcal{C}]/\mathcal{C}$ and $G_2 = \mathcal{L}[T \cup \mathcal{C}]/\mathcal{C}$, with a cut (S,T), and corresponding $\mathcal{C}$ provided in Line 5. The for-loop in Lines 9 to 14 iterates over every necessary permutation of vertices $\mathcal{C}$ to determine an optimal ordering for $\mathcal{C}$. Thus, there is a permutation $\pi^\star$ that corresponds to a turn-minimal ordering of $\mathcal{L}$ that is considered first in the for-loop. By induction Lines 10 and 11 compute turn-optimal orderings for $\mathcal{L}[S \cup \mathcal{C}]$ and $\mathcal{L}[T \cup \mathcal{C}]$ respecting Π and $\pi^\star$. Due to Lemma 22 both optimal orderings of $\mathcal{L}[S \cup \mathcal{C}]$ and $\mathcal{L}[T \cup \mathcal{C}]$ are merged correctly together with respect to the order $\pi^\star$. Further, since $\mathcal{C}$ is contracted in G_1 and G_2, vertices in $\mathcal{C}$ cannot be split into different subgraphs in later recursion steps such that the merged ordering obtained by `merge` also respects Π. Since both orderings of G_1 and G_2 are preserved, no new turn is introduced in the merged ordering. As $\pi^\star$ is the first permutation that corresponds to an optimal ordering, Line 13 is executed and the merged ordering is saved in $o^\star$ and never overwritten.

Importantly, during the computation of the optimal orderings o_1 and o_2, no three locations corresponding to three consecutive events of a train line that might cause a turn are left unconsidered. To see this, consider three locations u, v, w that correspond to three consecutive events of a train line in $\mathcal{E}$, and let $\{u, v\}, \{v, w\} \in E(\mathcal{L})$. For u, v, and w to be considered as potential turn, all three locations must either be completely contained in G_1 or in G_2. If none of the edges lies on the cut $\mathcal{C}$, then all vertices are either completely contained in G_1 or completely contained in G_2. If at least one edge $\{u, v\}$ or $\{v, w\}$ lies on the cut, then $\{u, v, w\} \cap S \neq \varnothing$ and $\{u, v, w\} \cap T \neq \varnothing$. Since $\mathcal{C}$ is contained in G_1 and G_2 as a contracted vertex and later expanded to $V(\mathcal{C})$, this potential turn is considered in at least one subgraph. Note that in the case, where both edges lie on the cut, and therefore u, v, and w are contained in both G_1 and G_2, the turn is predetermined by the fixed position π. $\square$

It is worth pointing out that we can use one of the exact algorithms proposed in Sections 7.1 and 7.2 as subroutine in `solveExact`, which enables us to use a moderately large constant size k. Further, note that the contraction of $\mathcal{C}$ for the subgraphs G_1, G_2 is indeed necessary, as otherwise a merging of two optimal orderings might be impossible due to contradictions. Consider the following example: Let $\pi_1 = \langle v_1^1, v_2^1, v_3^1 \rangle$ be the current ordering of vertices of a cut $\mathcal{C}_1$ of an ancestor node in the recursion tree during an execution of `DivideAndConquer` and let $\pi_2 = \langle v_1^2, v_2^2, v_3^2 \rangle$ be the current ordering of vertices of a cut $\mathcal{C}_2$ in the current recursion step. During the recursion, the vertices of $\mathcal{C}_1$ are split up in different subgraphs, resulting in two optimal orderings o_1 and o_2 in the current recursion step. The ordering o_1 ordered the vertices in $\mathcal{C}_1$ and $\mathcal{C}_2$ such that $v_1^1 \prec_1 v_3^1 \prec_1 v_1^2 \prec_1 v_2^2 \prec_1 v_3^2$ and o_2 ordered the vertices such that $v_1^2 \prec_2 v_2^2 \prec_2 v_3^2 \prec_2 v_2^1$, as Figure 8.4 illustrates. The orderings π_1 and o_1 imply that v_2^1 must be inserted before v_1^2, but π_2 and o_2 require that v_2^1 must be after v_3^2 and therefore after v_1^2. Consequently, o_1 and o_2 cannot be merged without contradicting π_1 and π_2. With the contraction of $\mathcal{C}$, the vertices in $\mathcal{C}$ remain in the same subgraph that is solved in the base case, thereby preventing this conflict.

Improving the Runtime. Testing all permutations of the vertices of a cut $\mathcal{C}$ is not always required. Consider the case, where there are only two vertices v_1, v_2 in $\mathcal{C}$. The algorithm tests the ordering $v_1 \prec v_2$ and $v_2 \prec v_1$ separately. However, this is not necessary since a turn-optimal ordering respecting $v_2 \prec v_1$ can be easily obtained by

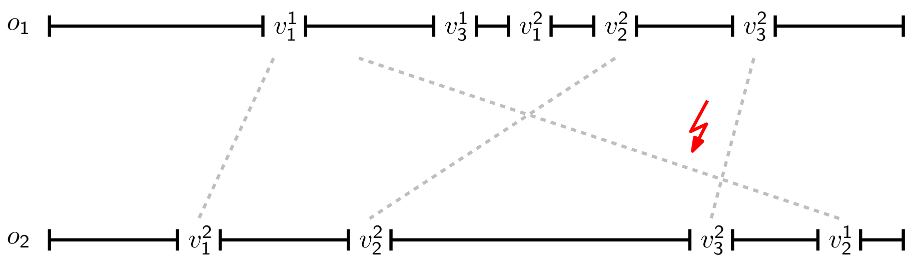

Fig. 8.4: A contradicting ordering o_1 and o_2 after a cut $\mathcal{C}_2$ split the graph into subgraphs G_1 and G_2, where the vertices v_1^2, v_2^2, and v_2^3 of an ancestral cut $\mathcal{C}_1$ are split up such that $v_1^2, v_2^2 \in V(G_1)$ and $v_2^3 \in V(G_2)$.

reversing the turn-optimal ordering computed for the case, where $v_1 \prec v_2$. Since turns are preserved by reversing the orderings, this ordering is also optimal. Thus, if a bridge is cut, only one ordering must be tested.

Further, consider a decomposition tree, that corresponds to the decomposition of $\mathcal{L}$ during the execution of `DivideAndConquer`, where each vertex in the decomposition tree corresponds to a subgraph of $\mathcal{L}$ created during the recursion of `DivideAndConquer`; see Figure 8.5 for an example. Moreover, assume that a contracted vertex is not part of any subsequent cut.

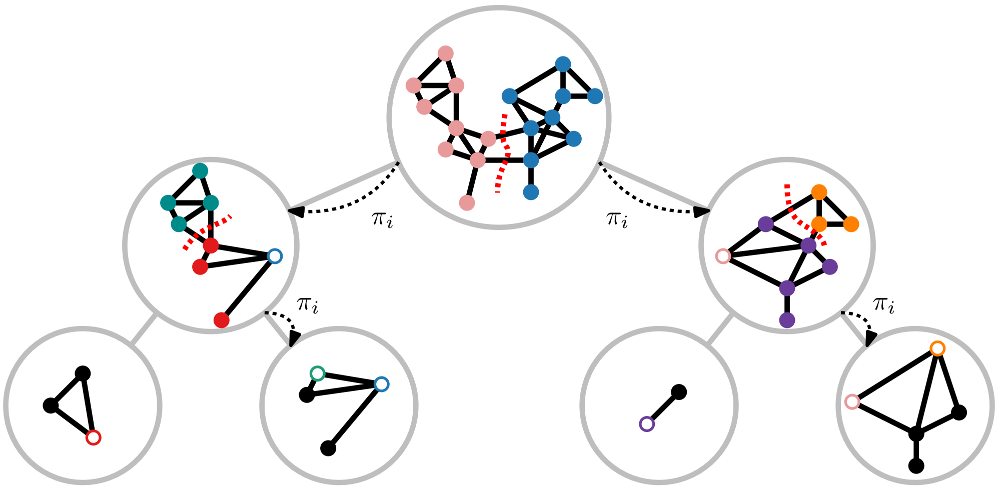

Fig. 8.5: A decomposition tree of an execution of the divide-and-conquer algorithm. Each vertex corresponds to a subgraph of $\mathcal{L}$. The children of a vertex correspond to the subgraphs G_1, and G_2. In this tree, we can see that an order π_i follows precisely two root–leaf paths.

Then, for each possible order π_i which is tested for a subgraph of $\mathcal{L}$, the order π_i follows two paths in the decomposition tree, due to the fact that at each recursion step, the cut $\mathcal{C}$ is contracted into a single vertex in G_1 and G_2. Thus, when π_i is considered the entire tree does not have to be recomputed but only the base case at the end of both paths. We can avoid the unnecessary computation of entire subtrees by using memoization, where we store solutions to the graph corresponding to $\mathcal{L}[S \cup \mathcal{C}]/\mathcal{C}$ in a table $M[S, \pi]$.

Let h be the height of the recursion tree of an execution of `DivideAndConquer` with memoization. In each recursion step, the set of vertices of a subgraph is split into two subsets, S and T. As a result, there are $\mathcal{O}(2^h)$ sets to store. Each set is associated with a cut $\mathcal{C}$ that generated it, meaning that for each set S, we need to consider at most $\mathcal{O}(|\mathcal{C}|!)$ possible orderings. Thus, the total number of entries we need to compute is $\mathcal{O}(2^h \cdot |\mathcal{C}_{\max}|!)$, where $\mathcal{C}_{\max}$ represents the cut with the largest number of vertices among all cuts. The computation of each entry needs linear time for merging the two orderings. Resulting in an overall runtime of $\mathcal{O}(2^h \cdot |\mathcal{C}_{\max}|! \cdot n)$ for `DivideAndConquer` with memoization, which constitutes an exponential improvement, provided no contracted vertex is part of a subsequent cut.

9 Heuristics

In Chapters 3 and 5 we showed that finding optimal solutions to the proposed aesthetics is computationally difficult, even if we relax the problem and allow solutions that are within a constant factor of the optimal solution. For this reason, we develop efficient heuristic approaches that do not make any quality guarantees but are faster than exact algorithms.

We investigate three different approaches for devising heuristic algorithms. First, we consider a local search approach in Section 9.1. Then, we use the divide-and-conquer framework developed in Chapter 8 and use it in two different ways in Section 9.2. Finally, we aim to minimize the number of turns by iteratively adding trains to a solution in Section 9.3

9.1 Local Search Heuristics

The first two heuristics are based on a local search approach, which is can be used for the optimization of both aesthetics. For the sake of simplicity, we only focus on the minimization of the number of turns and treat a mapping of locations to levels as a permutation of the locations.

Both local search algorithms find a new solution by flipping the levels of two locations u, v based on a current solution. Thus, a flip generates a new permutation by swapping two elements in a given current permutation. We can formulate the problem of finding an optimal permutation as a graph search problem, where the vertices of the search graph correspond to the different permutations and two vertices u, v are connected in the search graph if and only if the permutation corresponding to v can be generated by a single flip from the permutation corresponding to u. Note that the diameter of this graph is linear in the number of locations, as any permutation can be reached from another permutation by flipping each element directly to its corresponding position. Thus, we can reach an optimal solution in at most linear number of flips from any initial solution.

Exhaustively searching this graph is infeasible due to its size, in fact, even constructing the search graph of small diameter is impractical since the neighbourhood of each vertex is quadratic in the number of locations. Thus, our first local search algorithm works as follows: First we generate a random initial permutation. Then, we greedily choose the neighbouring permutation that improves the objective the most as the next current permutation and stop if no improvement is possible. With this algorithm we generally find a local optimum such as shown in Figure 9.1.

Note that one iteration of this algorithm therefore needs to consider $\mathcal{O}(|\ell(V)|^2)$ many pairs of locations to flip, which makes this algorithm slow. For this reason, we propose

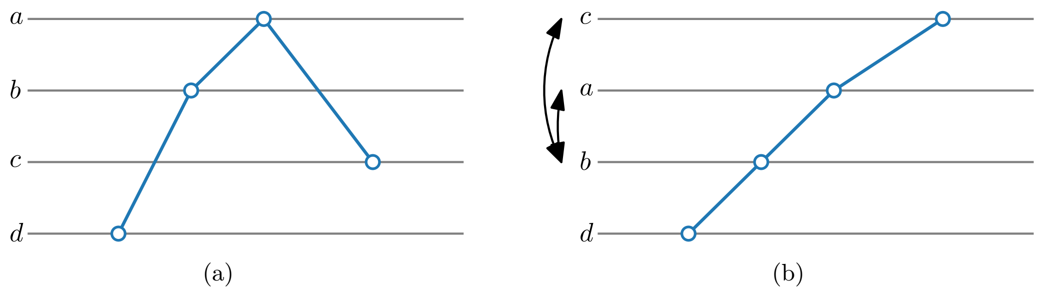

Fig. 9.1: Figure 9.1a shows a local optimum with one turn. No single flip improves the number of turns. However, two flips are sufficient to obtain an optimal solution, as Figure 9.1b shows. Thus, this permutation is a local optimum and our algorithm would terminate at this permutation without finding the optimal solution.

a simplified version, where instead of determining the flip that improves the current permutation the most, we pick a pair of locations at random and flip, if their swap results in an improvement.

9.2 Decomposition-based Approaches

In Chapter 8 we introduced exact algorithms for finding turn-optimal drawings based on decompositions and separators. Based on these, we develop two heuristic algorithms. The first heuristic follows straightforward from the divide and conquer algorithm described in Section 8.2. We keep the algorithm except for one difference. Instead of iterating over all possible orderings at a recursion step, we choose one ordering at random, and therefore do not branch on the different orderings.

The other heuristic is based on the intuition behind decompositions and separators. This heuristic groups locations that have a strong interaction with each other together. This is achieved by recursively separating locations with a minimum cut (S, T) in the location graph $\mathcal{L}$ and placing all locations in S in the upper side and locations in T on the lower side of the time-space diagram based on a given range of levels $\{a, a+1, \dots, b\}$. When applying this separation strategy recursively it makes sense to maintain the interactions between the cut (S, T) such that locations in S interacting with locations in T are placed closer together, and vice versa. So instead of computing minimum cuts recursively on the induced subgraphs $\mathcal{L}[S]$ and $\mathcal{L}[T]$, we contract the corresponding subsets into graphs $\mathcal{L}/T$, and $\mathcal{L}/S$, where T and S become a single vertex in the respective subgraphs. However, this could result in simply cutting off the contracted vertex again in the next recursive step, as such a cut is equivalent to the previous minimum cut, resulting in an algorithm that never terminates. Therefore, we additionally contract the new vertices with an adjacent vertex v that is connected with the heaviest edge before moving onwards recursively. See Algorithm 4 for a complete description of the algorithm.

Algorithm 4: Heuristic for the time-space Diagram via minimum cuts.

```
1  fn separate(location graph $\mathcal{L} = (V', E')$, integer $a = 1$, integer $b = |V'|$)
2      if $|V'| == 1$ then
3          assign location $v \in V'$ the only remaining level $a = b$
4          return
5      $(S, T) \leftarrow$ minCut($\mathcal{L}$)
6      $\mathcal{L}_U \leftarrow$ contractAndBind($\mathcal{L}$, $S$)
7      $\mathcal{L}_L \leftarrow$ contractAndBind($\mathcal{L}$, $T$)
8      separate($\mathcal{L}_U$, $b - |T|$, $b$)
9      separate($\mathcal{L}_L$, $a$, $a + |S|$)

   /* Contract the given vertex set $S$ in $\mathcal{L}'$ such that a single vertex
      $S$ remains.  Bind the vertex $v$ that is most connected to $S$ by
      also contracting the contracted vertex $S$ and $v$.                */
10 fn contractAndBind(location graph $\mathcal{L} = (V', E')$, set of vertices $S \subseteq V'$)
11     $\mathcal{L}' \leftarrow \mathcal{L}/S$        // Contract $S$, collapse multi-edges by summing the
        weights
       /* Select vertex that is connected the most to the contracted
          set $S$                                                        */
12     $v \leftarrow \arg\max_{v \in V' \setminus S} \left( \sum_{w \in N(v) \cap S} w(\{v, w\}) \right)$
13     $\mathcal{L}' \leftarrow \mathcal{L}'/\{v, S\}$               // Contract $v$ with contracted vertex $S$
14     return $\mathcal{L}'$
```

9.3 Greedy Train Orientation

The heuristic tries to minimize the number of turns by iteratively inserting trains into an ordering of locations such that the inserted train does not violate the existing ordering and such that its train line is as monotone as possible. The idea is as follows.

Given an event graph $\mathcal{E}$ and its associated location graph $\mathcal{L}$ with train lines $Z = \{z_1, \dots, z_k\}$, we sort Z with respect to the total weight of the edges of the train lines in $\mathcal{L}$, in descending order.

For simplicity, we assume that each train line is a simple path in $\mathcal{L}$. If this is not the case, we decompose each non-simple train line into multiple simple train lines. We construct a directed auxiliary graph G that is initially empty and to which we iteratively insert the edges of each train line until we obtain a directed version of $\mathcal{L}$. For a directed graph (or simply a set of arcs) H, let $E(H)$ be the set of undirected edges that correspond to the arcs of H.

Let $i \in \{1, \dots, k\}$ the index of the current iteration. For each connected component P of $E(z_i) \setminus E(G)$, we do the following. Note that P is adjacent to at most two vertices of G. If P is adjacent to *exactly* two vertices p and q in G, we test whether there is a q–p path in G. If this is the case, we select among the edges of P one that has the smallest weight in $\mathcal{L}$. We reverse the selected edge. Then we insert the vertices and the directed edges of P into G. (The reversal ensures that G remains acyclic.)

After the last iteration, we compute a topological order π of G and draw the time-space diagram of $\mathcal{E}$ induced by π. Algorithm 5 summarizes the algorithm.

```
Algorithm 5: Greedy Train Orientation Algorithm
  Input: Location graph $\mathcal{L}$ of an event graph $\mathcal{E}$ with simple train lines
         $Z = \{Z_1, \dots, Z_k\}$.
  Output: An ordering of locations aiming to minimize the number of turns in a
          time-space diagram.
1 Sort $Z$ in desc. order by the total weight of each train line $\sum_{(u,v) \in Z_i} w(\{u, v\})$
2 $G \leftarrow (\varnothing, \varnothing)$                // Empty directed auxiliary graph
3 foreach $Z_i \in Z$ do
      /* Split $Z_i$ into $u$-$v$ paths $P_j$ such that $V(P_j) \cap V(G) \subseteq \{u, v\}$     */
4     foreach $P_j \in$ split($Z_i$, $G$) do
5         $G \leftarrow G \cup P_j$
6         if $\exists$ $v$-$u$ path in $G$ then
              /* Insertion of $P_j$ into $G$ resulted in a cycle.        */
7             $(w_1, w_2) \leftarrow \arg\min\{w(e) \mid e \in P_j\}$
8             $G \leftarrow (G \setminus \{w_1, w_2\}) \cup \{(w_2, w_1)\}$
9 return topological order of $G$
```

Since it is crucial that G is an acyclic graph after the execution of Lines 3 to 8, we shortly argue why this is indeed the case. Before the first execution of the for loop G is empty and therefore acyclic. So let G by an acyclic graph, in which train lines $Z_1, \ldots, Z_{i-1}$ have been inserted and consider the insertion of a simple u-v path P_j. By the definition of `split`, P_j and G share at most two vertices, say u and v. If P_j and G do not share any vertex with each other it is clear that no cycle can occur after the insertion of G. Hence, only three distinct cases depicted in Figure 9.2 remain, when inserting P_j into G:

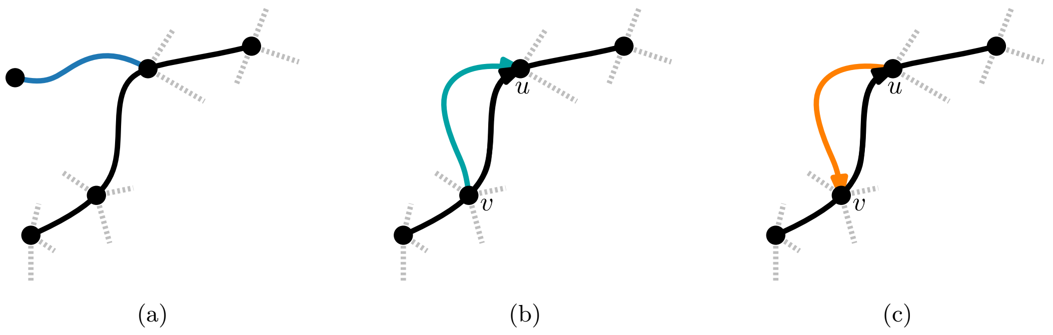

Fig. 9.2: Three different cases that can happen when P_j is inserted into G, where only case (c) results in a cycle in G.

In the first case, shown in Figure 9.2a, P_j and G share only one vertex with each other. Thus, regardless of the direction of P_j no cycle can occur after its insertion. Therefore, any cycle occurring in G after the insertion of P_j must contain u and v. If P_j and G share two vertices u, v, there is only one case in which a cycle in G can emerge, as shown in Figures 9.2b and 9.2c. In this case there is a v-u path in G that forms a cycle together with P_j which is tested by our algorithm. Then, we pick an edge contained in P_j and reverse it such that v cannot be reached by u anymore. Thus, the algorithm ensures that G remains acyclic which means that G admits a topological ordering.

10 Experimental Analysis

In this chapter we analyze a selection of the algorithms introduced in this work with respect to the quality of the solution (in the case of heuristic algorithms) and runtime on a realistic dataset provided by DB InfraGO. The dataset is based on real-world data and was anonymized.

We analyze the effectiveness of the transit component contraction reduction rule introduced in Chapter 6, when restricted to the special case of reducing chains. Further, we test the integer linear programs and all heuristics proposed in Chapter 9. Since a thorough testing of all parameters and optimization criteria is beyond the scope of this thesis due to the long time each experiments requires, we again focus only on analyzing the number of turns and a small set of parameters. While these restrictions unfortunately will not enable us to make general conclusions, we aim to present insights into the behaviour of the algorithms depending on some parameters that can be used to give directions for further experiments in the future. We start with an overview of the dataset.

An Overview of the Dataset. The dataset consists of 19 instances of varying sizes in the number of events, the number of locations, and number of trains, as shown in Figure 10.1. This dataset was generated on the same infrastructure but Figure 10.1 shows that the number of events increase with the number of locations, suggesting that the instances in the datasets cover different situations on different parts of the underlying infrastructure and thus the dataset provides a heterogeneous collection of data. We also observe that the dataset consists of instances with a different number of trains ranging from 3 trains to 20 trains, where every train line constitutes a simple path in the location graph.

We also observe the varying complexity in the dataset by examining the location graphs. Figure 10.2 shows two representative location graphs of the dataset that indicate the variety of complexity.

Figure 10.2 also gives evidence that the maximum degree of a location graph is expected to be low. In fact, the degree of a vertex in a location graph ranges from 1 to 5 in the provided dataset. This is most likely due to the inherently low connectivity of the underlying train infrastructure and can therefore be expected to generalize to other datasets.

Although we have not implemented the FPT algorithm proposed in Section 8.1 and can therefore not test its practical performance, we can investigate the treewidth of the instances in the dataset. Since computing the exact treewidth is difficult and not supported by the graph library we used in our implementation, we compute an interval in which the treewidth of a given (augmented) location graph of the dataset lies. We

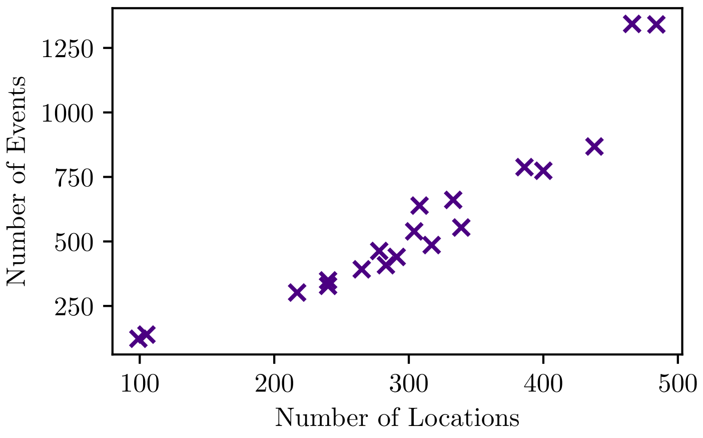

Fig. 10.1: On the left we see the instances with respect to the number of events and number of locations. The histogram on the right shows the distribution of the number of trains in the dataset.

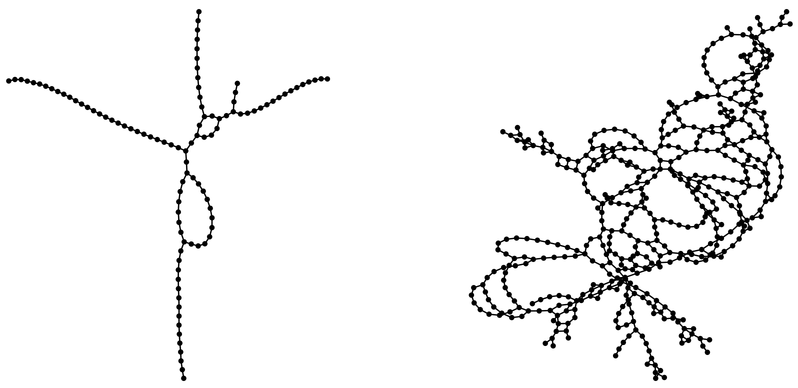

Fig. 10.2: The location graph on the left (instance 50_4) is smaller and sparsely connected compared to the location graph on the right (instance 20_3). For the sake of readability, the self-loops in the location graph are not drawn.

obtain the lower bound of the interval by computing the maximum clique size which is feasible due to the sparsity of the location graphs in the dataset. The upper bound is computed by taking the minimum over the widths provided by the Min-Fill-In and Min-Degree heuristic. Figure 10.3 shows the intervals for the location graphs (left) and the augmented location graphs (right). First, we can see that the treewidth of a location graph is generally low in the dataset with a minimum treewidth of 1 and a maximum treewidth estimate of 8. When considering the treewidths of the augmented location graphs, we observe that the lengths of the intervals, and hence the uncertainty of the true treewidths, increase significantly. However, the lower bound only increases slightly. In fact, the maximum clique size of the augmented location graphs increases at most by 2 compared to the maximum clique size in the corresponding location graphs.

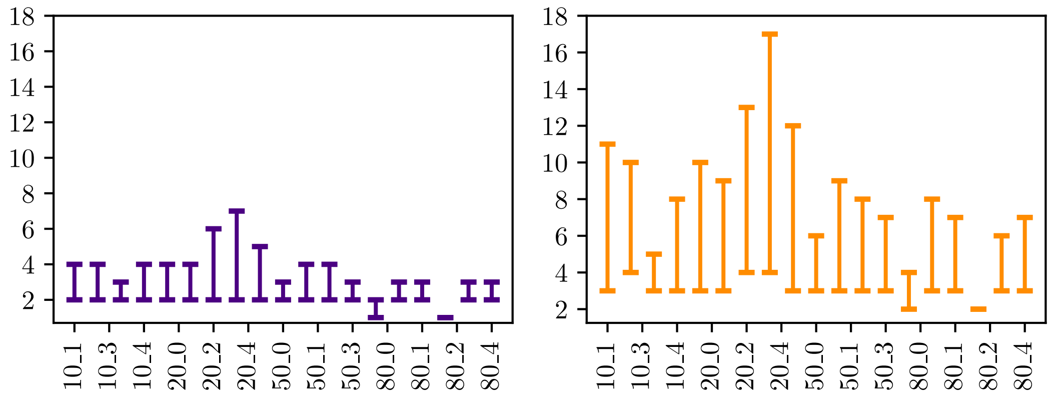

Fig. 10.3: Upper and lower bounds of the treewidths of the location graphs (left) and augmented location graphs (right).

Implementation Details. We implemented the algorithms in the programming language `Python` and used Networkx [HSSC08] for handling most of the graph operations and Gurobi [Gur24] for solving the (mixed-) integer linear programs. All experiments were conducted on a workstation running Fedora 40 with Kernel 6.10.6 using an Intel-7-8850U CPU with 16GB RAM.

In order to implement the callback function described in Section 7.2, we used a method for enumerating cycles of bounded length which is implemented in the Networkx library. The divide-and-conquer heuristic was implemented using a community detection algorithm in order to find balanced cuts of small size. The algorithm used is an edge betweenness partition which is also implemented in the Networkx library.

We computed minimum cuts with the Stoer-Wagner algorithm [SW97] which is implemented in the Networkx library with a runtime of $\mathcal{O}(VE + V^2 \log V)$.

10.1 The Effectiveness of Transit Component Contractions

We test the effectiveness of transit component contraction when restricted to chain contractions on the dataset by testing how much the number of locations is reduced. Figure 10.4 shows the resulting reduced sizes in a bar plot, where the initial size corresponds to the number of locations before the reduction and the reduced size corresponds to the number of locations after the chain reduction was applied. For instance, we notice that instance 10_1 originally contained 386 locations and was reduced down to 137 locations after the reduction was applied. Naturally, we observe that simpler location graphs are reduced stronger compared to more complex location graphs. For example, compare the reduction by 88.6% of the location graph of instance 50_4 to the reduction by 55.2% of the location graph of instance 20_3 (both depicted in Figure 10.2). In this dataset we obtain an average reduction by 75.0%. The rug plot shown in Figure 10.4 (right) breaks down the individual percentages, where a histogram (grey) with bin size of 5% indicates the percentage by which the location graphs in the dataset where reduced and the marks display the precise distribution of the reduction in the dataset. Based on this data, we can extrapolate an expected probability density function via a Gaussian kernel density estimation (black), which predicts the distribution of the reduction size on this specific infrastructure.

Based on the distribution shown in Figure 10.4, we can predict that the reduction rule is expected to significantly reduce many instances, but with diminishing returns for denser location graphs. Note that this implementation only contracted chains, and that a more general implementation is expected to produce better results on denser location graphs.

10.2 Performance of the Integer Linear Programs

Due to the computationally high effort that each test of an integer linear program requires, we only present a small selection of tests, as an exhaustive analysis of all parameters that influence the performance of each formulation is beyond the scope of this thesis. As a consequence, we can only give insights on the specific behaviour of the programs on a small set of test cases, and are therefore not able to make observations that are generalizable on other instances even within the test data.

We imposed a time limit of one hour per execution of an integer linear program. Unfortunately, none of the programs proposed in Chapters 4 and 5 where able to solve the instances in the provided time limit, except when employing the subroutine of the integer linear program described in Section 7.2. Even when running the programs on the reduced instances resulting from applying the reduction rule for removing transit components, most of the instances were not solved in the given time, except for two instances. Also, in this time limit, the optimality gap remained $> 90\%$ on the unsolved instances. Using the solutions of the proposed heuristic approaches as warm starts were not able to help.

However, the 0–1 integer linear program for minimizing the number of turns, using

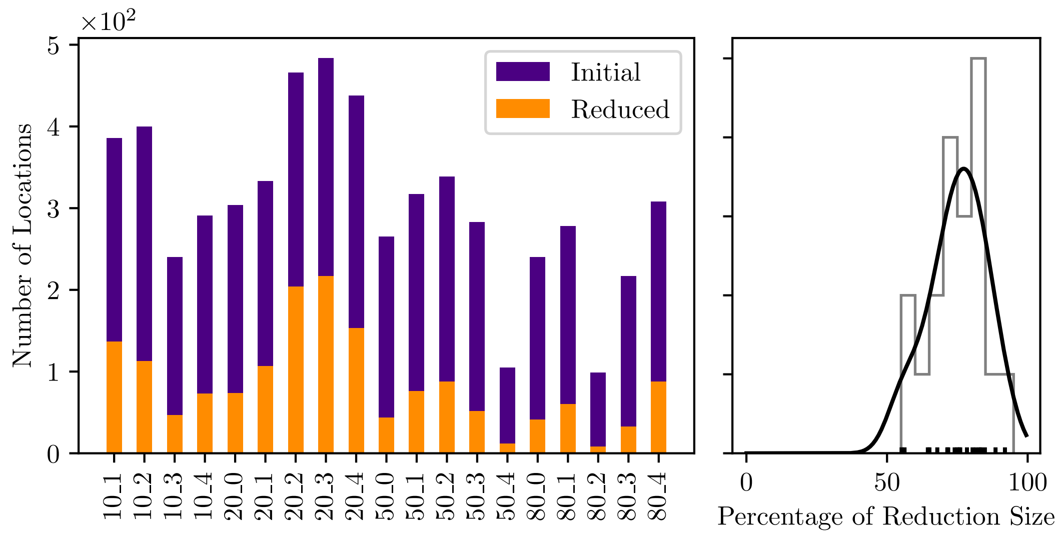

Fig. 10.4: Results of the effectiveness of applying the transit component contraction restricted to chains. On the left side, we see a bar diagram indicating by how much the locations of an instance in the dataset was reduced. On the right side, we see a rug plot showing the distribution of the reduction in percent as markers with a histogram as overlay and the result of a Gaussian kernel density estimation.

the subroutine was able to terminate on all instances within the given time limit. Note that the performance of the algorithm using the subroutine depends on the number of constraints k that are added to the model with each invocation of the subroutine. If the number of constraints that are added is small, then the subroutine is fast, but the solver tends to need more iterations, as a small number of constraints tends to cut off only a small number of solutions. On the other hand, if the number of constraints that are added is large, the subroutine takes longer to enumerate the necessary cycles and subsequent iterations of the solver take longer, as the model size increases rapidly. However, considering the fact that initial tests without the subroutine produced no results within the time limit on most of the instances, the number of constraints that should be added per invocation is most likely small compared to the possible maximum number of constraints.

For this reason, we test the performance behaviour for different values of k in detail on the instance 10_1 which was reduced via Reduction Rule 6.2 constricted to chains to 137 locations. Note that since the range of k is large, more tests on different instances are beyond the scope of this thesis but are required in order to make generalizable statements. Figure 10.5 shows the execution time depending on $k \in [20, 8000]$ in steps of 20. We notice a large variance across different values of k, suggesting that the performance of this algorithm is unstable with respect to k. A possible explanation for this instability is that a single constraint can change the solution of a solver significantly, such that in subsequent iterations these changes are propagated, resulting in an entirely different search structure within the solver.

We observe that in this instance both small values and large values for k indeed result in a performance drop-off, and that there is a range of values for k in which the algorithm is faster than for values that are outside the range. Considering values $k \in [20, 2400]$ shown in Figure 10.6, we can see that increasing k has sharp performance benefits until we reach $k \approx 200$, where the runtime plateaus, until it starts to slightly increase again at $k \approx 2000$. The average runtime in the range $k \in [200, 2000]$ is 21.13s with a standard deviation of 5.62s, where the minimum time of 11.49s occurred with $k = 1290$. Note that subsequent repetitions of the test where not able to confirm the minimum runtime at $k = 1290$, suggesting that this specific value was an outlier. Because of the instability and the lack of data on other instances, no generalizations are possible within the scope of this thesis.

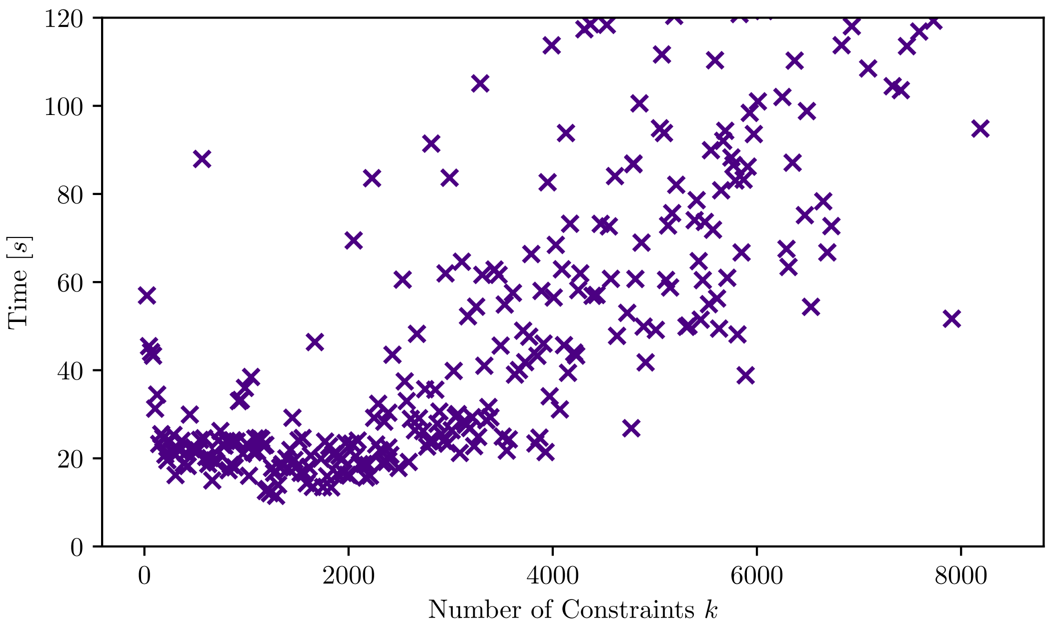

Fig. 10.5: Execution time of the integer linear program depending on the number of constraints added per subroutine invocation k. The data range is $k \in [20, 8000]$ in steps of 20.

In order to give an insight on the overall performance of this algorithm on the entire test data, we set $k = 200$ and repeat the test five times. Note that it is therefore possible that some instances can be solved faster, if $k = 200$ is not in the range that minimizes the runtime of the given instance. Figure 10.7 shows the average runtime of the algorithm depending on the reduced location graph size. We observe that in this setting, the algorithm achieves a runtime of at most 2s for instances with at most 70 locations. Further, we observe that instance 50_2 seems to be more difficult to solve compared to similar sized instances, under the assumption that $k = 200$ is not a bad configuration for this instance. Moreover, Figure 10.7 shows that the minimum number

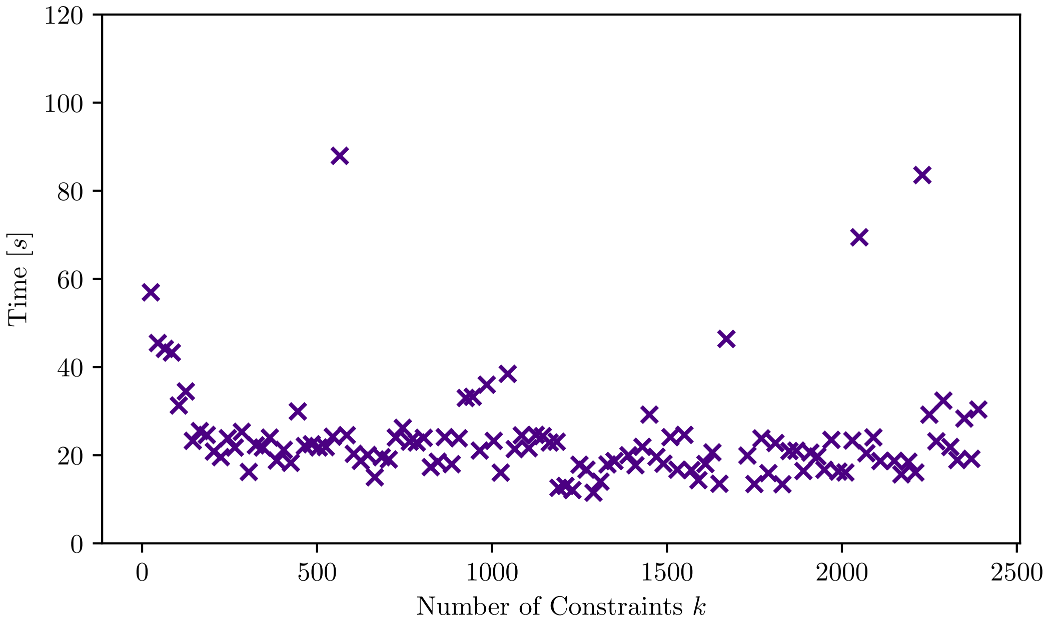

Fig. 10.6: Execution times shown in Figure 10.5 but focused on the range $k \in [20, 2500]$.

of turns in this dataset ranges from 0 to 5.

Considering that the integer linear program without the subroutine was not able to terminate on most instances, this is evidence that the subroutine significantly increases the performance of this program. In particular, small instances (reduced to ≤ 70 locations) can even be solved exactly within a short time.

10.3 Performance of the Heuristics

When employing heuristics we usually trade optimality for speed. Thus, we analyze the number of turns that the heuristics described in Chapter 9 achieve on the test data and the runtime that each heuristic requires. In order to analyze the performance, we ran each heuristic on every instance in the dataset and repeated the experiment 20 times since some heuristics contain a random component and in order to be able to average the runtime. Each instance was reduced by Reduction Rule 6.2 constricted to the reduction of chains beforehand. We analyze the heuristic that is based on the relaxation of the divide-and-conquer algorithm proposed in Section 8.2 separately, since the performance of this algorithm depends on the number of locations at which a subgraph is solved exactly. We start with the analysis of the other heuristics.

Figure 10.8 shows the results from the experiments, where the instances are plotted on the x-axis sorted by their number of locations after the reduction. Since both flip heuristics rely on randomization, as the initial solution is a random permutation of the locations, both heuristics are depicted with a box plot that shows the range of the

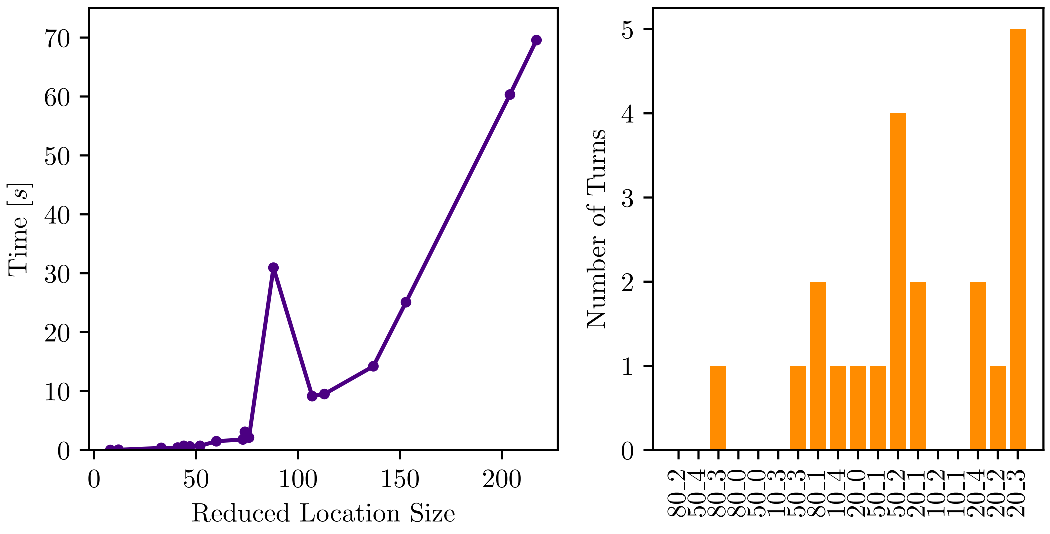

Fig. 10.7: Average runtime (left) of the integer linear program for minimizing turns combined with the subroutine, where the number of constraints per invocation is set to $k = 200$. The bar plot on the right shows the minimum number of turns that are possible in each instance.

number of turns the heuristics returned during the 20 repetitions. Each box represents the first and third quartile of the number of turns.

We observe that the greedy flip and the random flip heuristic return solutions with a similar number of turns, where the greedy flip heuristic returns solutions with slightly fewer turns on average. From the test data we can observe that the minimum cut heuristic is better or on-par with both flip heuristics on most instances, and that the greedy train orientation heuristic returns solutions with the fewest number of turns across the instances compared to these proposed heuristics.

Now, considering the runtime of the heuristics, depicted in Figure 10.9, we see that the greedy flip heuristic has a significantly larger runtime compared to every other heuristic. This is expected since this heuristic takes $\mathcal{O}(n^2)$ time per iteration, where n is the number of locations of the input. The minimum cut heuristic and the random flip heuristic have similar runtime, but the greedy train orientation algorithm is by orders of magnitude faster, with a maximum runtime of 15ms. Thus, we can conclude that from the tested heuristics, the greedy train orientation algorithm has the best optimality to runtime trade-off on this test data.

Testing the Relaxed Divide and Conquer Algorithm. We test the performance of the relaxed divide and conquer algorithm depending on the size of the location graph l at which the location graph is solved exactly with the integer linear program using the subroutine at $k = 200$. Each test is repeated 20 times and the average on the runtime and the number of turns is taken. Note that due to the randomly chosen order of the

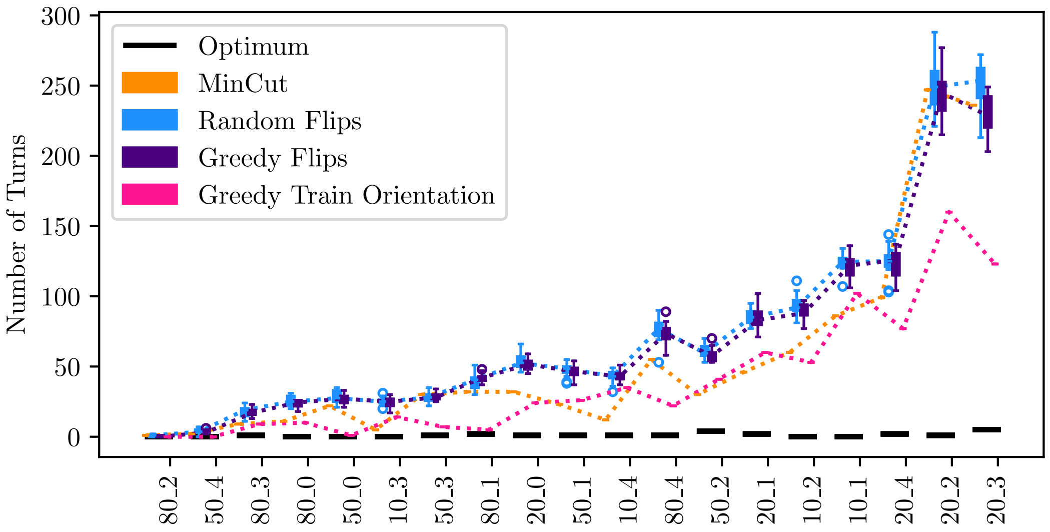

Fig. 10.8: A box plot showing the number of turns achieved by four different heuristics on the test data, where the tests where repeated 20 times. The instances (on the x-axis) are sorted by the number of locations on the reduced instances.

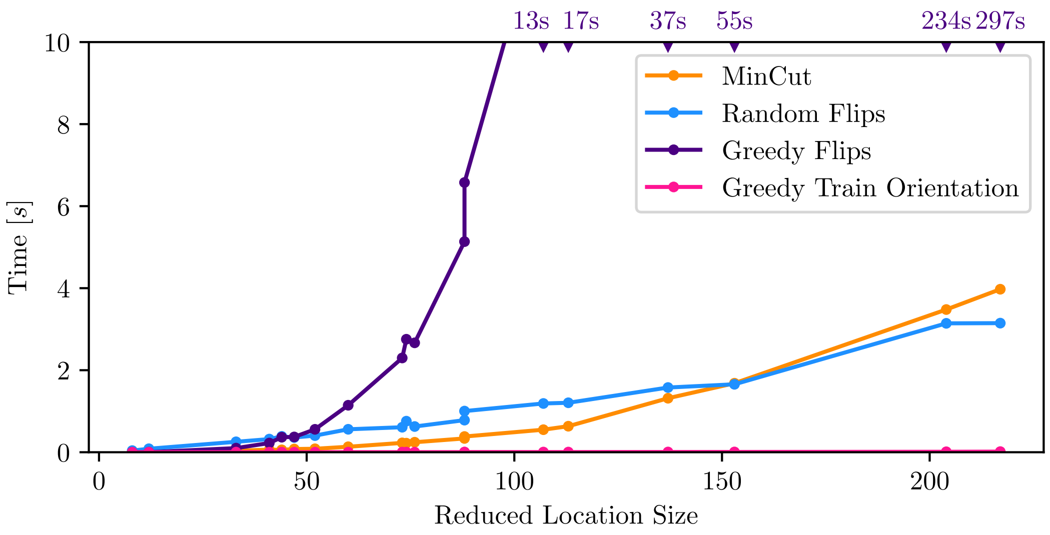

Fig. 10.9: The average runtime of four different heuristics on the test data by the reduced number of locations.

vertices of each cut, the number of turns remains not the same at each repetition.

We analyze the three reduced instances with the largest number of locations in detail, which are 20_3 with 217 locations, 20_2 with 204 locations, and 20_4 with 153 locations. We test the performance with values for l from 15 to 90 in steps of 5. Figures 10.10 to 10.12 show the results of the experiments on the different instances, where the number of turns are plotted with respect to the size l, and the color of the data points correspond to the runtime which is also depicted on the right of each plot. In all experiments, we observe a drop-off in the number of turns, when increasing the size l which is expected, since larger subgraphs are solved optimally with respect to a small fixed ordering of vertices. For the instance 20_3, we observe that the relaxed divide-and-conquer algorithm beats the greedy train orientation algorithm for $l \geq 25$ on average, while the greedy train orientation algorithm is beaten for $l \geq 20$ and $l \geq 15$ on average at instance 20_2 and 20_4, respectively.

However, this improvement in quality comes with a trade-off in runtime, where the runtime ranges between 3.14s to 13.00s for instance 20_3, 2.12s to 5.83s for instance 20_2, and 1.23s to 5.96s for instance 20_4, depending on l. With increasing l, the runtime increases which is expected, as the algorithm then has to solve multiple large subgraphs, and the runtime of the integer linear program increases noticeable at sizes larger than 60 as Figure 10.7 showed.

Thus, we can observe that in these instances a good trade-off between optimality and runtime can be achieved for values $l \in [40, 60]$ which corresponds to a range of 18.4% to 39.2% of the input sizes. Since, we only tested three representatives of a limited dataset, we cannot take any general conclusions and further testing is necessary.

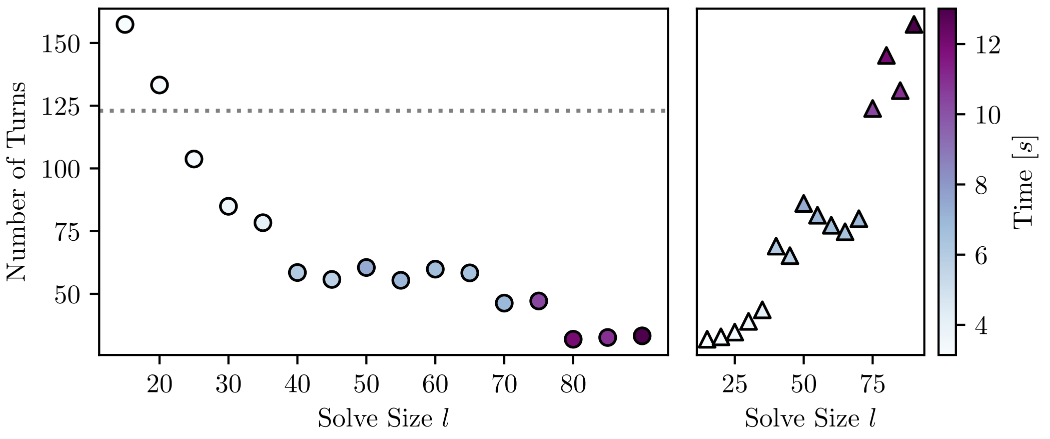

Fig. 10.10: Relaxed divide and conquer algorithm for different values of l on instance 20_3. The horizontal line at 123 turns is the result of the greedy train direction heuristic. The coloring corresponds to the time depicted on the right.

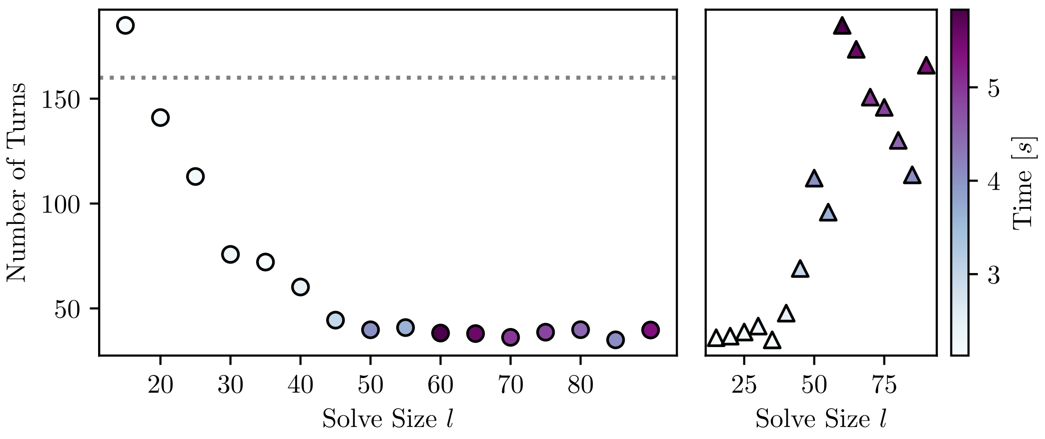

Fig. 10.11: Relaxed divide and conquer algorithm for different values of l on instance 20_2. The horizontal line at 160 turns is the result of the greedy train direction heuristic.The coloring corresponds to the time depicted on the right.

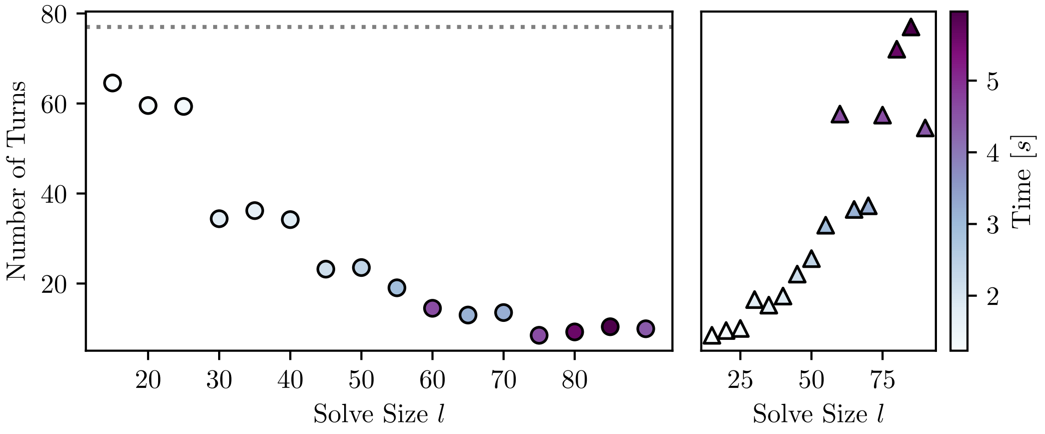

Fig. 10.12: Relaxed divide and conquer algorithm for different values of l on instance 20_4. The horizontal line at 77 turns is the result of the greedy train direction heuristic.The coloring corresponds to the time depicted on the right.

11 Conclusion and Future Work

This thesis aims to be a starting point for the development of a tool that facilitates the (real-time) analysis of event graphs for the scheduling of trains. We have introduced time-space diagrams for the visualization of event graphs that can be used to display delays and violations. We have defined two different criteria for readable time-space diagrams. First, in a good time-space diagram, there should be view crossings between line segments. Second, a polyline in a time-space diagram should be as y-monotone, which means that a polyline should rarely switch between going up or down. All optimization problems corresponding to these criteria were proven to be NP-hard. In fact, it is even NP-hard to find a solution that is within a constant factor of the optimal solution for the problem of minimizing crossings or turns. We also argued that the minimization of the number of turns is most likely more important than the minimization of the number of crossings.

Nonetheless, we proposed exact integer linear program formulations for computing crossing-minimal and turn-minimal time-space diagrams, but focused on algorithms for minimizing the number of turns. In particular, we proposed a 0–1 integer linear program for turn-optimal time-space diagrams and improved its practicability significantly by devising a cutting-plane approach that operates on a possibly incomplete formulation and finds constraints that cut off a current invalid solution. Further, we have introduced two algorithms that work on decompositions of the location graph.

While our modifications to the exact algorithms increased the instance size that are solvable with an exact algorithm in practice, the nature of the complexity of the aesthetics necessitate heuristic approaches for large instance sizes. Therefore, we have developed several heuristics, which can be categorized as follows: Heuristics that work on decompositions and greedy heuristics.

Algorithms in the first category use ideas of the exact algorithms which we proposed before. Algorithms in the second category greedily find good time-space diagrams by either swapping locations in a time-space diagram or directing edges in the location graph in order to obtain an ordering that can be transformed into a time-space diagram.

Based on our experimental analysis of real-world data, we conclude that it is possible to compute turn-optimal drawings of instances that are reducible to approximately 70 locations or fewer with the proposed integer linear program in at most 2 seconds. For instances with more than 70 locations, the runtime of the integer linear program increases rapidly, thus heuristics are needed in a real-world application. From the heuristics proposed in Chapter 9, the greedy train orientation heuristic and the relaxed divide-and-conquer heuristics produced the best results in terms of quality to runtime trade-off, where the former is the fastest and the latter produces solutions with the fewest turns compared to all other proposed heuristics.

However, more (detailed) experiments on different infrastructure are needed and the conducted experiments in this thesis must be done more thoroughly and on more instances. The influence of the number of constraints added per subroutine call would be particularly interesting to study in more detail such that more empirically sound statements of the recommended number are possible, and to determine why or if the integer linear program is unstable with respect to this parameter.

There are several other open questions left that are interesting to study. Considering the divide-and-conquer algorithm proposed in Section 8.2, the following subproblem arises: Given a turn-optimal drawing with respect to an ordering π, what is the complexity of finding a turn-optimal drawing with respect to an ordering π' that resulted from π by swapping two elements? Answers to this problem can yield faster computation times for the divide-and-conquer algorithm since a considerable factor contributing to the runtime of this algorithm is the computation of every possible permutation of a cut. As we can enumerate permutations by subsequently swapping elements, a faster algorithm leveraging a previous turn-optimal solution can be used for every subsequent computation of a permutation, thereby significantly improving its runtime.

The subroutine for finding violated constraints described in Section 7.2 might be improved as well. In its current implementation, the subroutine enumerates cycles of length three. However, it might be beneficial to not aimlessly enumerate cycles but to devise a method of finding cycles that "cover" the locations that are part of violated constraints more broadly. This would help to find optimal solutions faster.

Further, other structures and parameters might yield practical exact algorithms. For example, consider the reduction rule for contracting transit components. One property a transit component must have is that no terminal vertex is within the component. Since time-space diagrams are often snapshots of a larger schedule, the terminal vertices are often located at leaf vertices, which impedes the reduction of entire components. An algorithm parameterized by the number of terminal vertices that are leaves could be employed to solve components containing terminal vertices, rendering the reduction rule more effective.

The greedy train orientation algorithm can be improved by trying multiple orderings for the trains and returning the best solution. Meta heuristics such as simulated annealing can be used to combine the greedy flip and random flip heuristic, which can also provide a way to overcome local optima. Note that it is possible to combine other heuristics. For example, instead of choosing a random ordering of the cut vertices in the relaxed divide-and-conquer heuristic, the ordering can be taken from the greedy train orientation heuristic. As a post-processing step, the random flip heuristic could be used in order to further improve the solution within a given time frame.

A tool for the analysis of event graphs would benefit from allowing the user to dynamically alter the time-space diagram. For instance, it is useful if the user can switch the drawing of trains on and off or set a focus on a specific train. Also, when the given event graph operates on a longer timescale, a smaller time-window can be displayed by the visualization system and can then be shifted along the time axis by the user. Given such dynamic interactions, new possibilities and challenges emerge.

Bibliography

[AG] DB InfraGO AG: Geschäftsbereich Fahrweg. `https://www.dbinfrago.com/web/unternehmen/ueber-uns/profil-12600170#`, visited on 2024-08-06.

[Bod96] Hans L. Bodlaender: A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996, 10.1137/S0097539793251219.

[Bod98] Hans L. Bodlaender: A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209(1-2):1–45, 1998, 10.1016/S0304-3975(97)00228-4.

[Bod05] Hans L. Bodlaender: Discovering treewidth. In Peter Vojtáš, Mária Bieliková, Bernadette Charron-Bost, and Ondrej Sýkora (editors): *Theory and Practice of Computer Science, 31st Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2005)*, volume 3381 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005, 10.1007/978-3-540-30577-4_1.

[BT89] Giuseppe Di Battista and Roberto Tamassia: Incremental planarity testing. In *30th Annual Symposium on Foundations of Computer Science (FOCS'89)*, pages 436–441. IEEE Computer Society, 1989, 10.1109/SFCS.1989.63515.

[CS98] Benny Chor and Madhu Sudan: A geometric approach to betweenness. *SIAM Journal on Discrete Mathematics*, 11(4):511–523, 1998, 10.1137/S0895480195296221.

[Die17] Reinhard Diestel: *Graph Theory.* Springer, 5th edition, 2017.

[DPP07] Andrea D'Ariano, Dario Pacciarelli, and Marco Pranzo: A branch and bound algorithm for scheduling trains in a railway network. *European Journal of Operational Research*, 183(2):643–657, 2007, 10.1016/j.ejor.2006.10.034.

[Emm07] Stephen Emmitt: *Design Management for Architects.* Blackwell, 2007.

[FKM13] Vladimir Filipović, Aleksandar Kartelj, and Dragan Matić: An electromagnetism metaheuristic for solving the maximum betweenness problem. *Applied Soft Computing*, 13(2):1303–1313, 2013, 10.1016/j.asoc.2012.10.015.

[GH91] Martin Grötschel and Olaf Holland: Solution of large-scale symmetric travelling salesman problems. *Mathematical Programming*, 51:141–202, 1991, 10.1007/BF01586932.

[GM00] Carsten Gutwenger and Petra Mutzel: A linear time implementation of spqr-trees. In Joe Marks (editor): *Graph Drawing: 8th International Symposium (GD 2000)*, volume 1984 of *Lecture Notes in Computer Science*, pages 77–90. Springer, 2000, 10.1007/3-540-44541-2_8.

[Gom58] Ralph E. Gomory: Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64(5):275–278, 1958, 10.1090/S0002-9904-1958-10224-4.

[Gom10] Ralph E. Gomory: Outline of an algorithm for integer solutions to linear programs *and* an algorithm for the mixed integer problem. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey (editors): *50 Years of Integer Programming 1958–2008 – From the Early Years to the State-of-the-Art*, pages 77–103. Springer, 2010, 10.1007/978-3-540-68279-0_4.

[GP79a] Martin Grötschel and Manfred Padberg: On the symmetric travelling salesman problem I: Inequalities. *Mathematical Programming*, 16(1):265–280, 1979, 10.1007/BF01582116.

[GP79b] Martin Grötschel and Manfred Padberg: On the symmetric travelling salesman problem II: Lifting theorems and facets. *Mathematical Programming*, 16(1):281–302, 1979, 10.1007/BF01582117.

[Gur24] Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual, 2024. `https://www.gurobi.com`, visited on 2024-09-21.

[HSSC08] Aric Hagberg, Pieter Swart, and Daniel S Chult: Exploring network structure, dynamics, and function using networkx. In *Proceedings of the 7th Python in Science Conference*, pages 11–15. SciPy, 2008, 10.25080/TCWV9851.

[LK09] Shi Qiang Liu and Erhan Kozan: Scheduling trains as a blocking parallel-machine job shop scheduling problem. *Computers & Operations Research*, 36(10):2840–2852, 2009, 10.1016/j.cor.2008.12.012.

[LT82] George Loizou and Peter Thanisch: Enumerating the cycles of a digraph: A new preprocessing strategy. *Information Sciences*, 27(3):163–182, 1982, 10.1016/0020-0255(82)90023-8.

[Mak12] Yury Makarychev: Simple linear time approximation algorithm for betweenness. *Operations Research Letters*, 40(6):450–452, 2012, 10.1016/j.orl.2012.08.008.

[Opa79] Jaroslav Opatrny: Total ordering problem. *SIAM Journal on Computing*, 8(1):111–114, 1979, 10.1137/0208008.

[Pac02] Dario Pacciarelli: Alternative graph formulation for solving complex factory-scheduling problems. *International Journal of Production Research*, 40(15):3641–3653, 2002, 10.1080/00207540210136478.

[PCJ96] Helen C. Purchase, Robert F. Cohen, and Murray James: Validating graph drawing aesthetics. In Franz J. Brandenburg (editor): *Graph Drawing: Symposium on Graph Drawing (GD'95)*, pages 435–446. Springer, 1996, 10.1007/BFb0021827.

[PY91] Christos H. Papadimitriou and Mihalis Yannakakis: Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences*, 43(3):425–440, 1991, 10.1016/0022-0000(91)90023-X.

[QCL15] Wenhua Qu, Francesco Corman, and Gabriel Lodewijks: A review of real time railway traffic management during disturbances. In Francesco Corman, Stefan Voß, and Rudy R. Negenborn (editors): *Computational Logistics: (ICCL 2015)*, pages 658–672. Springer, 2015, 10.1007/978-3-319-24264-4_45.

[RS86] Neil Robertson and Paul D. Seymour: Graph minors. II. algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986, 10.1016/0196-6774(86)90023-4.

[Sav09] Aleksandar Savić: On solving the maximum betweenness problem using genetic algorithms. *Serdica Journal of Computing*, 3(3):299–308, 2009, 10.55630/sjc.2009.3.299-308.

[SKMD10] Aleksandar Savić, Jozef Kratica, Marija Milanović, and Djordje Dugošija: A mixed integer linear programming formulation of the maximum betweenness problem. *European Journal of Operational Research*, 206(3):522–527, 2010, 10.1016/j.ejor.2010.02.028.

[SL76] Jayme L. Szwarcfiter and Peter E. Lauer: A search strategy for the elementary cycles of a directed graph. *BIT Numerical Mathematics*, 16(2):192–204, 1976, 10.1007/BF01931370.

[SW97] Mechthild Stoer and Frank Wagner: A simple min-cut algorithm. *Journal of the ACM*, 44(4):585–591, 1997, 10.1145/263867.263872.

Erklärung

Hiermit versichere ich die vorliegende Abschlussarbeit selbstständig verfasst zu haben, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben, und die Arbeit bisher oder gleichzeitig keiner anderen Prüfungsbehörde unter Erlangung eines akademischen Grades vorgelegt zu haben.

Würzburg, den 2.10.2024

....
Samuel Wolf