TU WIEN Informatics

Elastic Set Visualization

Hypergraph Visualization with Moving Objects and Fat Edges

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Sebastian Adam, BSc
Matrikelnummer 01525543

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Inform. Dr.rer.nat. Martin Nöllenburg
Mitwirkung: Dr. Jules Wulms
Dipl. Ing. Soeren Terziadis

Wien, 10. März 2023

Sebastian Adam | Martin Nöllenburg

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Elastic Set Visualization

Hypergraph Visualization with Moving Objects and Fat Edges

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Sebastian Adam, BSc
Registration Number 01525543

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Inform. Dr.rer.nat. Martin Nöllenburg
Assistance: Dr. Jules Wulms
Dipl. Ing. Soeren Terziadis

Vienna, 10th March, 2023

Sebastian Adam	Martin Nöllenburg

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Sebastian Adam, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 10. März 2023

Sebastian Adam

Danksagung

Zuallererst möchte ich meiner Familie und allen voran meinen Eltern Margit und Christian Adam dafür danken, dass sie mich während meiner gesamten akademischen Laufbahn unterstützt haben. Ein ganz besonderer Dank geht an meine Partnerin Allegra Anwander, die mich in allen Lebenslagen begleitet und unterstüzt.

Ich möchte auch meinen Betreuern Martin Nöllenburg, Jules Wulms und Soeren Terziadis für ihre kontinuierliche Unterstützung und ihr wertvolles Feedback während dieser Arbeit danken.

Schließlich möchte ich mich bei meiner Hündin Ophelia bedanken, die zu Beginn meiner Arbeit an dieser Diplomarbeit als Welpe zu uns kam und seither für emotionale Unterstützung gesorgt hat.

Acknowledgements

First and foremost, I want to thank my family, especially my parents Margit and Christian Adam for supporting me throughout my entire academic career. A very special thanks goes to my partner, Allegra Anwander, who accompanies and supports me in all situations of life.

I also want to thank my supervisors Martin Nöllenburg, Jules Wulms and Soeren Terziadis for their continued support and valuable feedback during this thesis.

Finally, I want to thank my dog Ophelia, that came as a puppy to us at the beginning of my work on this thesis and has provided emotional support ever since.

Kurzfassung

Die Visualisierung von Hypergraphen ist äquivalent zur Visualisierung von Sets, da die Knoten eines Hypergraph als Setelemente und die Hyperedges als Sets interpretiert werden können. Daher ist die Visualisierung dieser Strukturen ein wichtiges Thema sowohl in der Graphendarstellung als auch in der Informationsvisualisierung. Bei einigen Hypergraphen kann die Zuordnung von Knoten im Graphen zu Positionen in der Visualisierung nicht beliebig gewählt werden (beispielsweise bei der Visualisierung von geographischen Daten auf einer Karte). Die Visualisierung solcher Hypergraphen mit einer existierenden Knotenzuordnung ist ein aktives Forschungsgebiet, für das in den letzten Jahren mehrere verschiedene Methoden entwickelt worden sind. Diese bestehenden Visualisierungsmethoden sind im Allgemeinen für eine statische Knotenzuordnung entwickelt worden. In der Praxis ist es jedoch möglich, dass sich diese Knotenzuordnung im Laufer der Zeit ändert oder auf Benutzereingaben reagieren muss. Ein Beispiel dafür wäre die Visualisierung von Flugzeugen auf einer Karte, bei der die Flugzeuge durch Knoten dargestellt werden und die Hyperedges die Zugehörigkeit zu einer Fluggesellschaft darstellen.

In dieser Arbeit schlagen wir eine Technik vor, die die Grundlage für eine solche Hypergraph-Visualisierung bildet. Diese Technik ist inspiriert von State of the Art Hypergraph-Visualisierungstechniken und anderen Konzepten der Graphendarstellung, um Hypergraphen nicht nur in einer statischen Umgebung zu visualisieren, sondern auch mit zeitlich veränderlichen Daten umgehen zu können. Zu diesem Zweck entwickeln und implementieren wir neue Algorithmen, um diese Art der Visualisierung zu konstruieren. Darüber hinaus führen wir Qualitätsmetriken für die Visualisierung ein und verwenden diese Metriken, um die von den implementierten Algorithmen erzeugten Visualisierungen zu bewerten.

Abstract

Visualizing hypergraphs is equivalent to visualizing sets, since the vertices of a hypergraph can be interpreted as set items and the hyperedges as sets. Therefore, visualizing these structures is a prominent topic in both graph drawing and information visualization. For some hypergraphs, the mapping of vertices in the graph to positions in the visualization cannot be chosen arbitrarily (e.g. when visualizing geospatial data on a map). Visualizing such hypergraphs with a fixed vertex mapping is an active area of research for which several different methods have been developed in recent years. These existing visualization methods are generally designed with static vertex mappings in mind. However, in practice, it is possible that vertex mappings change over time or have to react to user input. An example of this would be a visualization of planes on a map, where the planes are represented by vertices and the hyperedges represent the membership to an airline.

In this thesis, we propose a technique that builds the foundation to handle such a hypergraph visualization. It takes inspiration from state-of-the-art hypergraph visualizations and other graph drawing concepts to not only visualize hypergraphs in a static environment but is also adept at dealing with time-varying data. To that end, we develop and implement new algorithms to compute this type of visualization. We furthermore introduce quality metrics for the visualization and use those metrics to evaluate visualizations produced by the implemented algorithms.

Contents

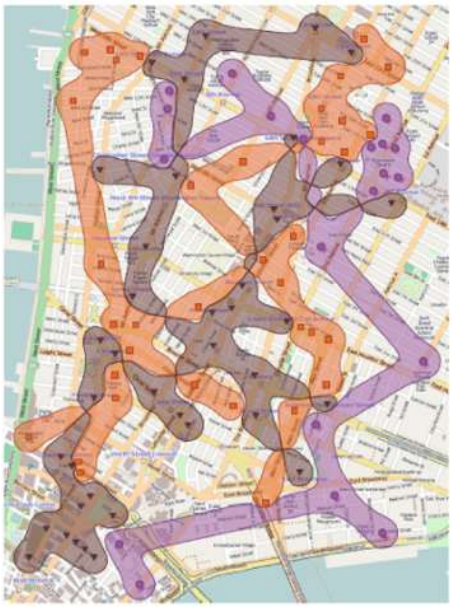
CHAPTER 1

Introduction

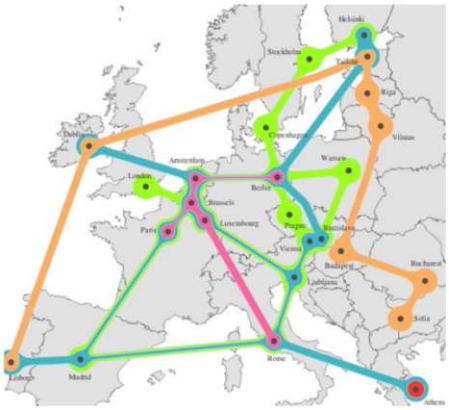
Visualizing graphs for humans is an extensive and well-studied topic. However, there is not always a consensus about the correct or best representation for each graph. Especially visualizations of complex graphs can get quite sophisticated and often come with advantages and disadvantages that have to be evaluated for specific use cases. The simplest form of graphs, aptly called simple graphs, are undirected graphs that do not contain any loops or parallel edges and only contain edges that connect two nodes. Visualizing simple graphs can already be a challenge in itself. Different aesthetic criteria like drawing the edges with as few bends as possible [MW06], only using right-angle crossings [Sch21], or minimizing the number of crossings [MNKF90] are difficult problems to solve and at least NP-hard in their complexity.

In this thesis, we focus on the visualization of a more complex type of graph, the so-called *hypergraph*. The main difference between a hypergraph and a simple graph is that in a hypergraph each edge (also referred to as a *hyperedge*) can join any number of nodes. Hypergraphs are of special interest because visualizing them is equivalent to visualizing a collection of sets with a finite number of elements. In such a visualization the edges of a hypergraph correspond to sets and the nodes of the graph to its elements [AMA+16].

Graphs are often visualized on a two-dimensional surface (also called the *plane*). Such a visualization requires a mapping from the vertices of the graph to positions in the plane as well as a mapping from each edge to a line or curve (depending on the visualization style) in the plane. For some visualizations, the mapping of the vertex positions in the plane is not predetermined and can be altered to better fit aesthetic criteria. In other cases, there are additional constraints regarding this mapping. An example of this would be a graph whose nodes correspond to landmarks in a city that is drawn on a surface that represents the map of the town (see Figure 1.1a). In this case, it certainly makes sense to map the vertices of the graph according to their real-world locations on the map of the town.



(a) Bubble Set method [VPF+14] visualizing hotels (orange), subway stations (brown), and medical clinics (violet) in lower Manhatten



(b) Kelp Diagram method [DvKSW12] visualizing European capital cities in the eurozone (blue), EU founding members (pink), states with good, average, or bad credit ratings (green, orange, and red respectively)

Figure 1.1: Examples of area-based (a) and line-based (b) visualization methods for hypergraphs

Hypergraphs with fixed vertex mapping. When visualizing sets, and therefore by extension hypergraphs, one approach that may seem intuitive, is to place the vertices that represent elements from the same set close to each other, therefore making it easier to draw the hyperedges. This method is for example used in Euler diagrams [SA08], which draw a contour around each set containing all its set items. The spatial proximity of the set items hereby helps to keep the drawn set contours small. In this thesis, however, we focus on the visualization of hypergraphs with already given vertex mappings. This creates an additional challenge since elements from the same set are possibly spread out across the plane, which makes it harder for the yet-to-be-drawn hyperedges to convey set membership. Drawing hypergraphs with an already existing vertex mapping can be useful for visualizing a variety of different applications, including but not limited to real-world places on a map (or geospatial data in general) [WCW+22], data points on a diagram [WCW+22] and metabolic networks [DvKSW12].

Creating the drawing of such hypergraphs with an already existing vertex mapping is an active research area for which several different methods have been developed in recent years. The challenge these techniques have to solve is to create an edge mapping for the drawing that manages to convey set membership correctly, while not visually overloading

the visualization. Approaches that have been developed thus far can be categorized into three main categories [WCW+22], namely *area-based* methods, *line-based* methods, and *hybrid* methods. The different types of methods are further explored in Section 1.1.

Working with moving subjects. Existing hypergraph visualization techniques try to fulfil different structural and aesthetic criteria (see Section 1.1). Generally, these techniques have in common that they are designed with static vertex mappings in mind. In practice, however, it is possible that a vertex mapping is time-varying or has to react to user input. This creates new challenges as edge representations also have to be adapted in accordance to the vertex movement, while still fulfilling the criteria of the visualization.

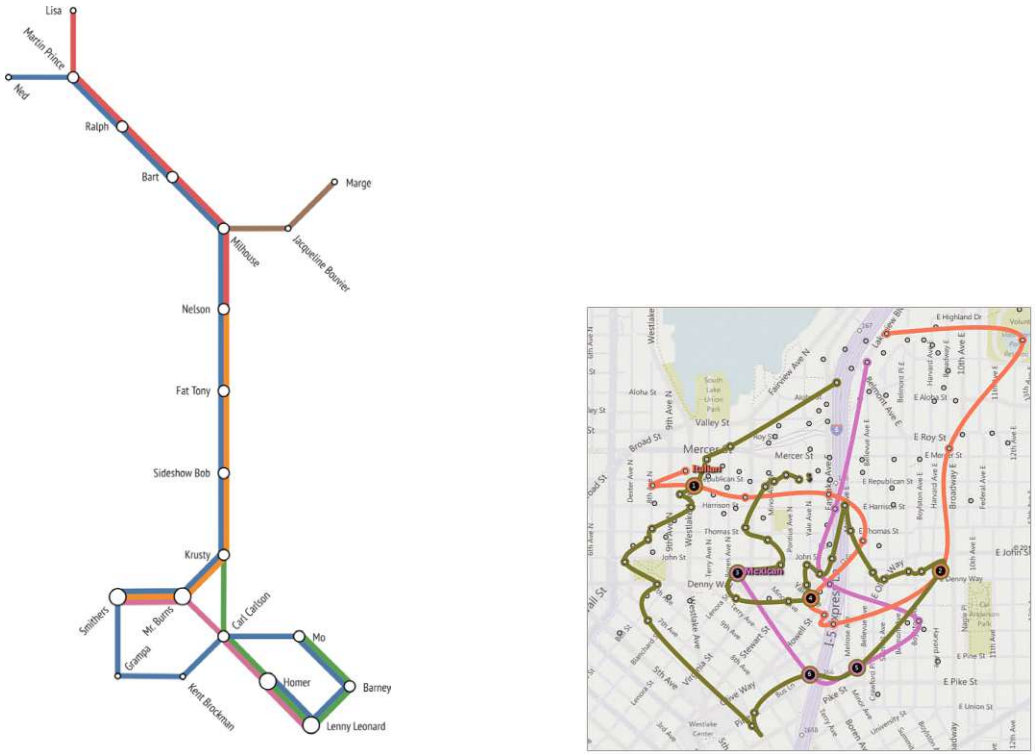
A practical example of such a time-varying mapping would be a hypergraph in which the vertices represent planes and the edges represent the membership of a plane to a specific airline. In this example, the hypergraph is drawn on a real-world map and the vertices are mapped to the locations of the planes they represent. Since the planes certainly change their position over time, this is reflected in a time-varying vertex mapping.

The other important factor that can change the drawing of a hypergraph is user input. In some cases, users might want to change the vertex mapping itself (i.e. move set elements to different positions), in other cases users want to change the outcome of the visualization by adapting the drawn hyperedges to better match the desired aesthetics. Existing techniques like Kelp diagrams [DvKSW12] (see Figure 1.1b) and F2-Bubbles [WCW+22] (see Figure 1.3b) offer such functionality, however, they were designed to create static visualizations of hypergraphs. Therefore their primary goal is to create a new static drawing after a change triggered by a user occurs and not to smoothly transition the existing drawing to match the new state. In a kinetic environment, however, where the goal is to appealingly visualize this movement, the smoothness of the transition between states is important as well.

Developing a new visualization technique. The motivation of this thesis is to extend existing visualization techniques with a new approach that is developed with moving vertices in mind from the start. The main goal of this visualization is to be rather stable (i.e. do not make big changes to the edge mapping that are not necessary) while still allowing to smoothly transition states if changes occur (either predefined time-varying or from user input). Before we elaborate on how our visualization technique tries to achieve this stability, we consider the techniques we take inspiration from, along with other important related work.

1.1 Related Work

The current state-of-the-art techniques to visualize hypergraph embeddings can be differentiated into three different categories: area-based methods, line-based methods, and hybrid methods [WCW+22]. Line-based methods solely use lines to represent hyperedges, while area-based methods use contours that enclose nodes belonging to the



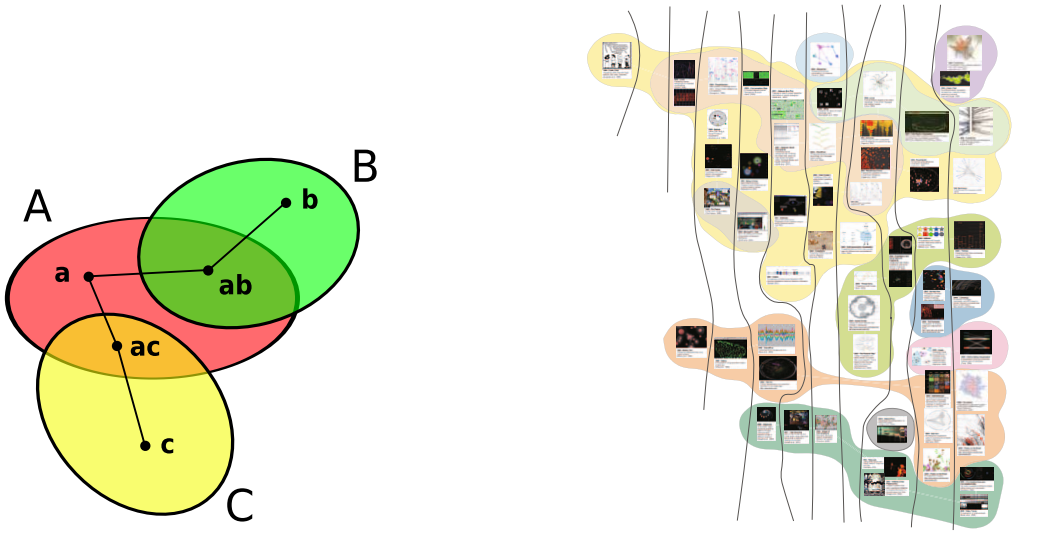
(a) MetroSet method [JWKN21] visualizing a data set that shows connections between characters of the comic series The Simpsons

(b) LineSet method [ARRC11] visualizing different categories of restaurants on a map

Figure 1.2: Examples of line-based visualization methods for hypergraphs

same set. Hybrid methods combine the two approaches. In contrast to the aim of this thesis, current state-of-the-art techniques were mostly designed to draw hypergraphs with a static vertex mapping.

Line-based approaches. MetroSets [JWKN21] (see Figure 1.2a) aim to visualize hypergraphs in the style of metro maps, where the set elements are the stations and the hyperedges are the metro lines connecting these stations. However, the four-step pipeline as well as the design goals defined in [JWKN21] are not designed to visualize hypergraphs with a given vertex mapping. On the contrary, the vertex mapping (i.e. the ordering and spacing along a metro line) is chosen in such a way that it optimizes the quality criteria of the visualization. LineSet [ARRC11] (see Figure 1.2b) and Kelp diagrams [DvKSW12] in contrast are two examples of line-based methods to draw hypergraphs given such a mapping. Both of them use lines to connect the members of a set, however, in the LineSet visualization, each set is represented as a single path traversing all its elements, while Kelp diagrams build an underlying graph structure that is used to compute which edges are drawn.



(a) Euler diagram method [SA08]

(b) F2-Bubbles method [ARRC11] visualizing a grouping of research articles on a timeline

Figure 1.3: Examples of area-based visualization methods for hypergraphs

A general advantage of line-based methods is the reduction of ink that is used to overlay set information [ARRC11]. However, the path-based approach of LineSets can imply a sequential order, as one study shows [MRS+13]. Since sets by definition do not impose any order on their members this property is generally not desired in set visualization.

In the first iteration of our approach, presented in this thesis, our visualization has similarities to LineSets as there exists only a single line connecting the two set elements. However, scaling to general instances, we envision an approach with a topology more similar to Kelp diagrams [DvKSW12], which use an underlying graph structure that is more complex than a simple path traversing all nodes and therefore does not imply any order. Computing such a structure is an active area of research [CvGM+19] and can be a computationally expensive task [WCW+22], which is less of a concern in a static setting. For the non-static approach pursued in this thesis, we, therefore, aim to create a technique that can adapt the paths of an existing visualization, rather than creating an entirely new one, as long as the changes are not too extreme.

Area-based approaches. Euler diagrams (see Figure 1.3a) are one of the more well-known ways to visualize set membership [SA08]. The basic idea of Euler diagrams is that every set is drawn as a contour that splits the plane into two regions: Everything inside the contour is part of the set and everything outside of the contour is not part of the set. If two contours overlap then this indicates an overlap of elements belonging to both sets. In practice, it is not always possible to use such Euler diagrams to visualize hypergraphs

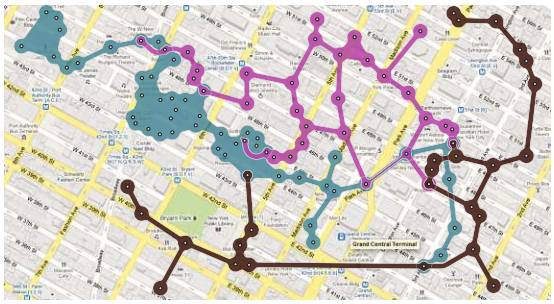


Figure 1.4: KelpFusion method [MRS+13] method visualizing points of interest and their categories on a map

with given vertex mappings, since these could require an overlap of sets even though the sets do not have common elements.

Still, the main idea of Euler diagrams is also applied in other area-based approaches like Bubble Sets. Bubble Sets [CPC09] compute a contour for each set that encloses all of its elements and excludes elements not belonging to the set where possible. F2-Bubbles [WCW+22] are based on the idea of Bubble Sets and try to minimize the area where two or more set enclosures overlap without sharing a common element at that position, by adjusting the contour of the enclosure. A similar idea is used in our visualization by variably adjusting the width of the hyperedges. This is especially important in a non-static setting, where this variably adjustable width allows us to squish and squeeze the paths to react to certain types of movement.

A shortcoming of area-based methods, in general, is that they use a lot of ink, and therefore cannot represent many different sets on a small surface without overloading the visualization. An advantage that area-based methods have in common is that the visual enclosure of nodes instinctively represents set membership for human observers and does not imply a sequential order.

Hybrid approaches. KelpFusion [MRS+13] is a hybrid approach that tries to combine Kelp diagrams and Bubble Sets. It uses lines to connect set elements, but also filled convex hulls of point sets in areas where a set has a high density of elements belonging to it. This way it combines some of the advantages of area-based methods (drawing enclosures, which intuitively represent sets) with the advantages of line-based methods (using less ink and not creating too much overlapping area that does not carry any information).

Fat edges. Apart from state-of-the-art hypergraph visualization techniques, drawing graphs with thick edges is another important area of research for this thesis. Fat edges [DEKW06] as a concept can help visualize the importance of edges or edge weights in simple graphs. However, the concept can also be applied to draw the paths in our hypergraph visualization. The thick edges are expected to better transport the concept

of set membership and the algorithmic aspects and techniques introduced in [DEKW06] can be used to figure out the maximum possible width paths can have in places where the width is restricted.

1.2 Modelling a new set visualization

The idea for our visualization is to combine ideas from different state-of-the-art approaches described in Section 1.1 and develop a visualization that is designed to also work well in a non-static environment. Visually we aim to achieve distinct aesthetic criteria for this visualization. First, we want our visualization of sets to look uniform. We think this is best achieved by a line-based approach, especially for more complex set structures. However, area-based techniques have the advantage that they are very intuitive in conveying set membership. Therefore we want to add some thickness to the lines to give the impression of an area-based approach. Finally, we try to avoid overlaps with other set elements, and overlaps with other sets in general, as much as possible, for clarity.

To achieve these three aesthetic criteria (uniformity, thickness and avoiding elements) we take inspiration from existing techniques and build a model that incorporates the aesthetics in such a way that it easily extends to kinetic settings and further allows users to influence the visualization in some ways (such as control points used to influence edge routings). For uniformity, we aim for a similar style as KelpFusion [MRS+13] (see Figure 1.4) and Kelp diagrams [DvKSW12]. In contrast to Kelp diagrams, however, we want to draw our edges as thick edges to better convey set membership using concepts and ideas of Duncan et al. [DEKW06]. Finally, to avoid overlaps we take inspiration from F2-Bubbles [WCW+22], which use control points to influence their edge routings and can dynamically change edge width. In the following, we define a few core modelling concepts of our visualization.

Similar to the control points used in F2-Bubbles [WCW+22], we introduce a concept called *obstacles*. Obstacles can either represent set elements themselves (where other sets have to be routed around) or user-placed points in the plane which are used to influence the visualization. We want our sets to not overlap with these obstacles and additionally constrain the sets to be routed on a particular side of the obstacles. In a static setting, this already creates a challenging scenario, as it requires a non-trivial algorithm that is able to route the set paths according to the obstacle demands. In a kinetic setting, where the obstacles move, these challenges are amplified as these moving obstacles can push the sets.

To properly route multiple edges along the border of an obstacle we use a concept called *orbits*. Orbits are concentric rings around an obstacle that help us guide the edges. By changing the size of an orbit we get control over the thickness of a set path. Furthermore, the orbits help with uniformness, since the width of a set is uniform when it is routed along an orbit and all sets that route along the same obstacle use the same orbit width. Finally, changing the orbit sizes gives us the possibility to ensure that sets do not overlap with obstacles or with each other. An example of how we imagine our visualization model

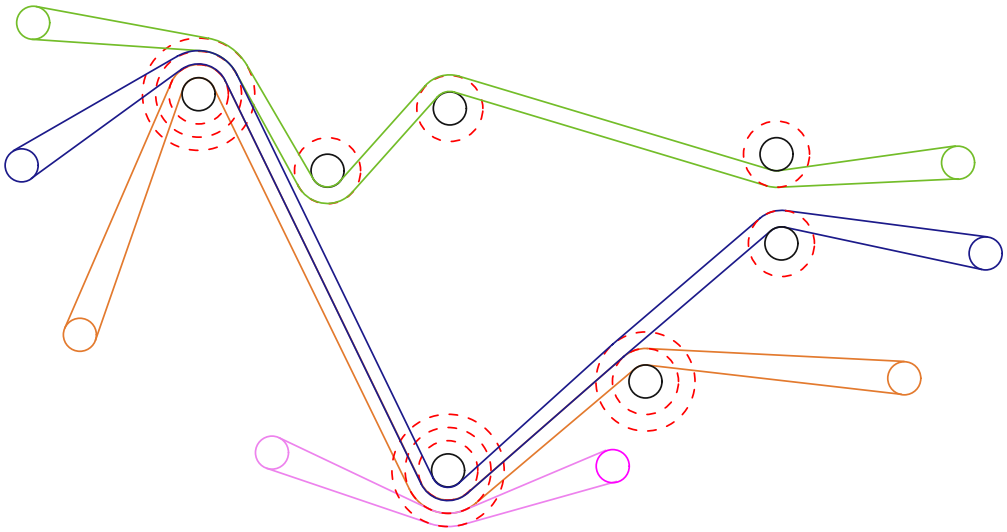


Figure 1.5: Simple hypergraph example using our visualization style containing multiple set items (colored), obstacles (black) and their orbits (red)

to look like, using the introduced concepts of obstacles and orbits as well as thick edges, can be seen in Figure 1.5.

In a kinetic setting, where the vertex mapping of a hypergraph changes (either time-dependent predefined or by user input) and the changes are not too extreme, we imagine that the set paths in our visualization model are still being routed roughly through the same orbits. This is expected to help create a rather stable appearance in which the hyperedges resemble elastic bands. The varying edge width our model achieves with the use of orbits is especially crucial in kinetic instances. For example, if hyperedges are routed between two moving obstacles and the space between them shrinks to be not sufficiently large for the target edge width. Existing techniques that focus on static visualizations can mitigate this issue by simply changing the set routings completely to better fit the end state of the movement. Our model in contrast aims to also provide a smooth transition visualization during the movement, which can be achieved by gradually adapting the orbits of the moving obstacles.

Changing the orbits of obstacles during the movement allows us to use the same set path routings (as long as the changes are not too extreme and we can avoid overlaps), thereby contributing to a mostly stable visualization. Furthermore, by adapting the size of the orbits during the movement we can ensure the thickness and uniformity of the paths similar to the static case. An example of how this orbit adaption can be used to adjust the edge width, such that hyperedges fit through an otherwise too narrow gap created by moving obstacles without creating overlaps, can be seen in Figure 1.6.

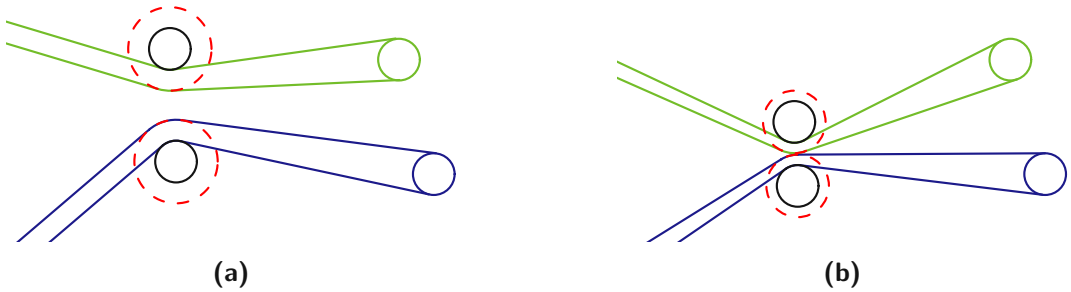


Figure 1.6: Cutout of instance shown in Figure 1.5 before (a) and after (b) movement. The width of the hyperedges is decreased from (a) to (b) by adjusting the orbit radii

1.2.1 Contributions

There are a multitude of things to consider when developing an entirely new visualization technique for hypergraphs. Among other things, one needs to consider in what way the set elements (nodes) of the graph are connected, which involves creating a (tree-like) structure that underlies the actual visualization. Furthermore, in a kinetic environment, one would need to decide how entangled the sets are allowed to become before (discrete) changes to the structure are applied to disentangle them. We, therefore, consider the development of a complete visualization for hypergraphs to be out of scope for this thesis. However, the above mentioned concerns are mostly independent of whether the idea of an elastic set visualization works out conceptually.

Therefore, the aim is to create a solid foundation for the described hypergraph visualization in combination with an objective evaluation of the achieved results. The overarching goal is to create hypergraph drawings that are mostly stable and can transition smoothly when changes to the vertex mapping occur. To that end, we aim to show that obstacles in combination with the orbits around them allow us to create stable and uniform sets with a certain (variable) thickness.

Some of the important challenges associated with these objectives are the same regardless of the size of the hypergraphs. Therefore, and to avoid the above mentioned concerns one has to deal with in a general setting, we focus on the development of an algorithm that automatically creates visualizations on a restricted subset of hypergraph instances. These hypergraphs have the following properties:

- Hyperedges only connect two nodes
- Hyperedges can be represented by a monotone path
- Hyperdges can be drawn without crossings
- Each set item belongs to exactly one hyperedge

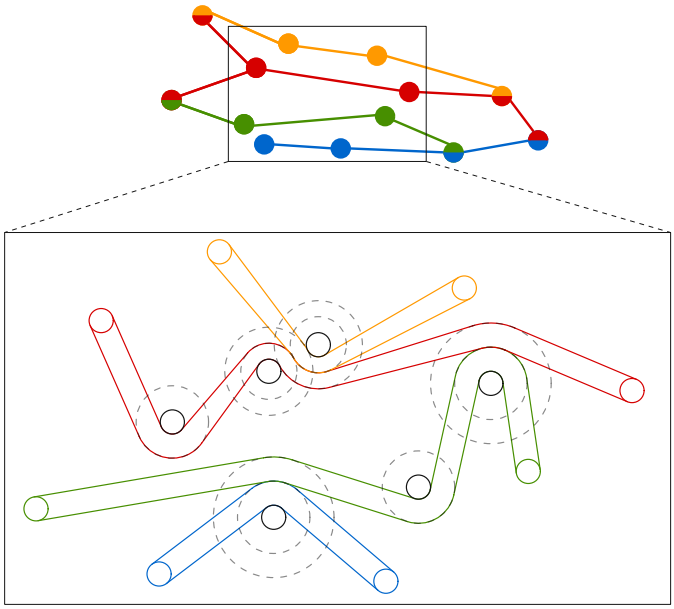


Figure 1.7: Example showcasing our visualization style using obstacles and orbits on a restricted hypergraph instance (bottom) as part of a larger general hypergraph instance (top)

Instances with such restrictions can naturally occur in general hypergraph instances, as can be seen in Figure 1.7. The focus on this restricted subset of hypergraphs allows us to focus on the style of the visualization as well as the visualization of basic movement, without having to take into account the topology of general hypergraph instances.

To get a complete visualization of a general hypergraph from a visualization of such a restricted instance, we would have to find a way to connect the endpoints of the paths in a sensible way. Furthermore, splitting any general hypergraph input instances into the above defined restricted instances would require us to find a way to ensure that the split instances only contain monotone paths and can be drawn without crossings. However, these conversions are not explored in this thesis.

Approach & Outline of the thesis. Using the modelling outlined in the introduction, we develop algorithms for static and kinetic set visualizations. We prove certain properties and provide asymptotic worst-case runtime analysis. However, we are also interested in seeing how the visualization style works in practice. Therefore, we develop a graphical user interface (GUI) and implement the algorithms, such that we are capable of creating and manipulating the defined hypergraph instances. This tool is developed in Java using the GeometryCore library [Meu19] and is also used to create all the figures of example instances throughout this thesis. Finally, we use this implementation for an evaluation of both the static and kinetic quality criteria of the visualizations produced by our algorithms.

In Chapter 2, we provide definitions to all newly introduced as well as existing concepts used in this thesis. Afterwards, in Chapter 3 we describe the visualization algorithm for static hypergraph instances and in Chapter 4 we describe the adaption of our methods for kinetic environments with basic movement. The visualizations produced by these algorithms are then evaluated in Chapter 5 and finally, we summarize our achievements and give a brief outlook on possible future work in Chapter 6.

CHAPTER 2

Preliminaries

In this chapter, the concepts and terminology used throughout this thesis are described.

2.1 Newly introduced Concepts

Obstacles & Orbits. Obstacles are two-dimensional objects in the plane that can either be points or circles with varying sizes of radii. We do not allow the space an obstacle uses in the plane to be intersected by any other obstacle or hyperedge. Furthermore, each obstacle demands from each set that its path is either routed above or below the obstacle. This list of demands is also referred to as obstacle demands.

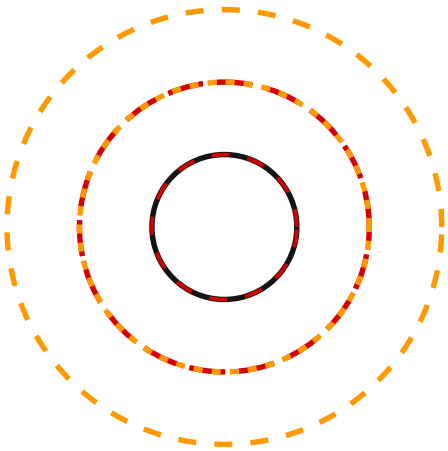


Figure 2.1: Obstacle with two orbits (red and orange). Each orbit consists of two concentric circles.

Orbits are thick concentric circles around obstacles. The inner (r_i) and outer (r_o) radius of such an orbit are defined as $r_i = r_b + nr_d$ and $r_o = r_b + (n+1)r_d$, where r_b is the base radius of the obstacle, r_d is the width of the orbits and n is the order of the orbit (starting with $n = 0$ for the first orbit). These orbits are used to route the set paths around the obstacles. An example of an obstacle with two obits can be seen in Figure 2.1.

(Static) Instance. The instances used throughout this thesis consist of set elements and obstacles. Formally, they can be defined as $\mathcal{I} := \{S, O, f\}$, where S is a set of sets (each containing two set element positions), O is a set of obstacle positions and f is a boolean function that takes a set and an obstacle as an input and returns whether the set has to be routed above or below the obstacle. Depending on the demand an obstacle has regarding a set we refer to it as an *above obstacle* or *below obstacle*.

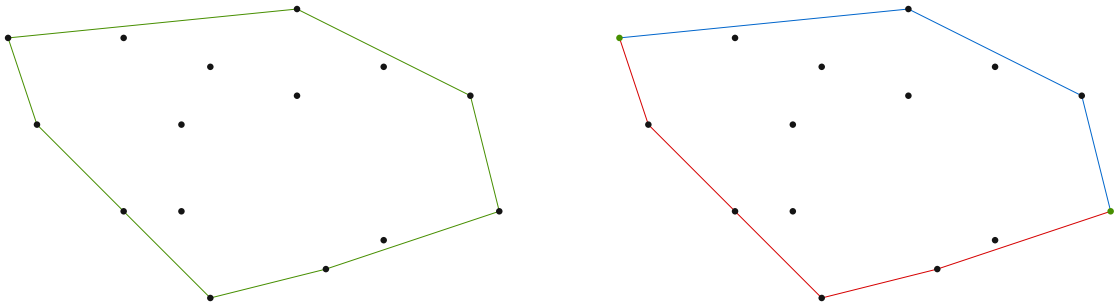
Each visualization creates a set path routing out of an instance input. The created visualization can be defined as $\mathcal{V} := \{S, O, d, e, g, h\}$, where S is again the set of sets (each containing two set element positions) and O is a set of obstacle positions. The function d maps the sets to tuples of elements (set elements and obstacles) describing the set path. The functions g and h map the obstacles to tuples of sets, describing the sets that are routed on orbits above and below the obstacles respectively. Finally, the function e maps the obstacles to an orbit radius that each of the sets routed along an orbit of the obstacle uses.

2.2 Existing Concepts

Stateless & State-aware algorithms and visualizations. Meulemans et al. [MSVW17] make the assumption that an algorithm is a function from input to output. Such an algorithm is regarded as being *stateless* since the output of the algorithm is only dependent on the input. In contrast to that, and especially in a kinetic environment, algorithms exist that produce different outputs for the same input, depending on previous states. These algorithms are referred to as being *state-aware*.

As an extension to the algorithm-specific definitions from Meulemans et al. [MSVW17], in this thesis we call kinetic visualizations stateless/state-aware, depending on if they are computed by a stateless/state-aware algorithm.

Convex Hull & Graham Scan. The convex hull of a point set P in a two-dimensional plane is a closed polygon with minimal perimeter that contains all points of P. An example of such a convex hull can be seen in Figure 2.2a. In this thesis, we further use the terms upper and lower convex hull. The upper convex hull is the upper envelope of a convex hull and the lower convex hull is the lower envelope of a convex hull between two endpoints. The upper and lower envelope of a convex hull can be identified by finding the extremal points in horizontal direction (leftmost and rightmost), which split the convex hull into an upper chain and a lower chain of points. These chains represent the upper



(a) Convex hull (green) of a point set P

(b) Upper convex hull (blue) and lower convex hull (red) of point set P between two extremal points (green)

Figure 2.2: Convex hull examples

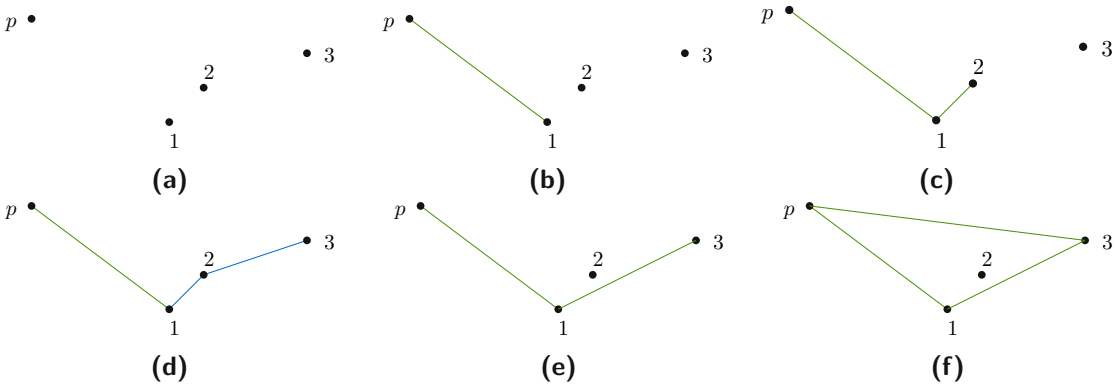


Figure 2.3: Graham Scan example on a small point set. The numbers beside each point represent the order in which the points are considered. In (d) the three most recent points form a right turn (blue), which is why point 2 is removed from the resulting set.

and lower envelopes of the convex hull. An example of such a partition of the convex hull can be seen in Figure 2.2b.

The Graham Scan algorithm [Gra72] is one of the simpler algorithms that exist to find the convex hull of a point set P. The algorithm works incrementally and starts with an extreme point of the point set (e.g. the point furthest to the left), which is known to be part of the convex hull. It then sorts the remaining points of P in a list depending on their angle to the starting point and starts adding them to the resulting set of points that lie on the convex hull. After each step the algorithm checks if the last three added points form a *right turn* (see Figure 2.3d), which would mean that the middle point lies inside the convex hull and consequently has to be removed from the resulting set. The algorithm ends when it has processed all the points in the sorted list. An example run of how the Graham Scan is able to find the convex hull of a small point set is given in Figure 2.3.

The runtime of the Graham Scan algorithm is $O(n \log n)$, where n is the number of all points in the point set. Other, and under the right circumstances, more efficient algorithms exist, however. For example, Chan's algorithm [Cha96] is able to find a convex hull with an output-sensitive runtime of $O(n \log h)$, where h is the number of points that lie on the convex hull.

CHAPTER 3

Static Visualization

This chapter describes the methods used to efficiently generate a visualization of multiple sets in the plane. Depending on the conditions of the input instance and the desired look of the visualization different algorithms are used. The overarching goal of these visualizations is to draw the hyperedges as thick paths and route them as efficiently as possible (i.e. trying to avoid unnecessary detours or waste too much space) while still fulfilling all of the obstacle demands. The instances on which the algorithm is tested are restricted, in that all their sets only contain two set items and the edges can be drawn without any overlaps.

In the following sections, the evolution of the algorithm is incrementally described, starting with a method that is capable of handling single sets in simplified instances.

3.1 Basic Routing

This first iteration of the algorithm operates in simplified instances of the original problem. In these instances, the set items, as well as the obstacles, are represented by points in the plane, and the paths, portraying the edges between the set items, consist of straight line segments with zero width. Furthermore, this initial method is intended to only handle a single set in such an instance.

The goal of this first iteration of the algorithm then is to efficiently find the shortest path an edge between two set items can take on, while still fulfilling all the demands of the obstacles on its way. Figure 3.1 shows what such a path should look like in an exemplary instance. This problem is equivalent to solving the homotopic shortest path problem on monotone instances as shown by Sergei Bespamyatnik [Bes03]. However, unlike the approach presented by Sergei Bespamyatnik [Bes03] that solves the problem in linear time, we intend to further extend our approach later on with multiple sets, obstacles with a non-zero radius and thick paths.

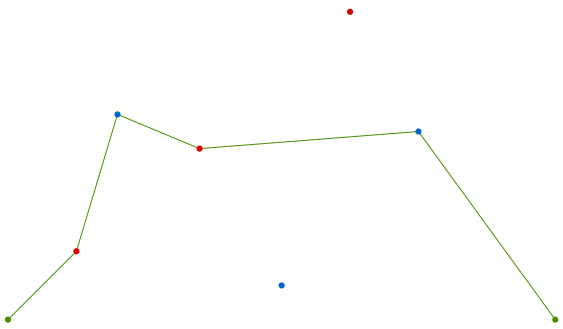


Figure 3.1: Exemplary instance for the basic routing algorithm containing one set (green) and multiple above obstacles (blue) as well as below obstacles (red).

It is visible from Figure 3.1 that some of the obstacles lie directly on the shortest path, while the demands of others can be fulfilled without creating a detour. The resulting visualization of a hyperedge can be described by the obstacles that lie on its path. In the following, we will therefore refer to such obstacles as being *used* by a set path.

The idea for this first basic routing algorithm now is to successively find obstacles that have to be used by the path and mark them as such until all demands are fulfilled. One way to determine such an obstacle is to locate the ones at the extreme positions of the plane (i.e. the obstacles with an above demand and the highest y-coordinate and the obstacles with a below demand and the lowest y-coordinate) between the set items. This is because no matter which other obstacles are ultimately used by the path, the demands of these obstacles at extreme positions can only be fulfilled if they are also used by the path themselves or if the path already fulfils all obstacle demands within its start and end point.

Lemma 3.1.1. *The demand of an obstacle at an extreme position that is not yet fulfilled can only be satisfied if the obstacle is used by the set path.*

Proof. Assume that there is an obstacle o_1 at an extreme position between the two set items that is not used by the final set path. W.l.o.g. we further assume that the extreme obstacle is the above obstacle with the highest y-coordinate, as an analogous argument can be made for the below obstacle with the lowest y-coordinate. In case the extreme obstacle is positioned such that a theoretical straight line between the two set items would lie above the obstacle Lemma 3.1.1 is trivial to prove, as the demand of the obstacle at the extreme position, as well as all other above obstacles in this case, is already satisfied. In the following, we therefore focus on the case in which the extreme obstacle is positioned such that a theoretical straight line between the set items would lie below the obstacle.

By definition of the obstacles and their demands, we know that the final drawing of the edge has to be routed above the obstacle. By assumption, we know that o_1 is not used by the set path, but since the path still has to be routed above the obstacle it can be

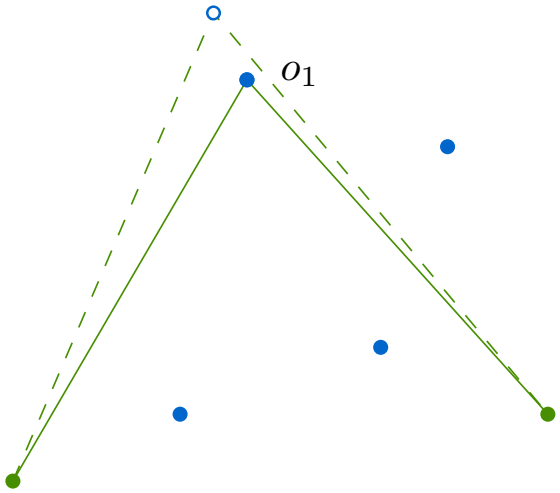


Figure 3.2: Sample instance showing that the only way that the extreme obstacle o_1 is not used by the set path is if there is another above obstacle above it (meaning it is not an extreme obstacle in the first place).

concluded that there has to be some space between o_1 and the set path. Since we are searching for the shortest path, the path can only pass the obstacle without touching it, if it was pushed further upward by another obstacle. For an obstacle to push the path upward, it needs to be an above obstacle and to push the path beyond o_1 the obstacle needs to be above o_1, as visualized in Figure 3.2. This is a contradiction since o_1 is the above obstacle with the highest y-coordinate. Therefore the assumption has to be wrong and o_1 has to be used by the set path. □

With this method, it is possible to find one obstacle at a time that has to be used by the final set path. After such an obstacle is found it splits the problem instance into two subproblems, where the just-found obstacle acts as an end or start point. These subproblems then need to be scanned again for their extreme obstacles within their start and end points that possibly also have to be added to the set path. This creates a recursive method that adds obstacles to the set path until a subproblem is not further split because all obstacle demands between its start and end point are already fulfilled. For this method, it does not matter if it first searches for the extreme above obstacle or the extreme below obstacle as long as it only stops within a subproblem if both of their demands are fulfilled.

Another approach for finding obstacles that have to be used by the set path, which is capable of finding multiple of them at a time, is to utilize the characteristics of the convex hull. Specifically, all obstacles on the upper (or lower) convex hull created using the set items as well as all above (or below) obstacles between them have to be such obstacles. The idea of why every obstacle on the boundary of such a convex hull has to be used by the final path is similar to the previously described method that uses extreme obstacles.

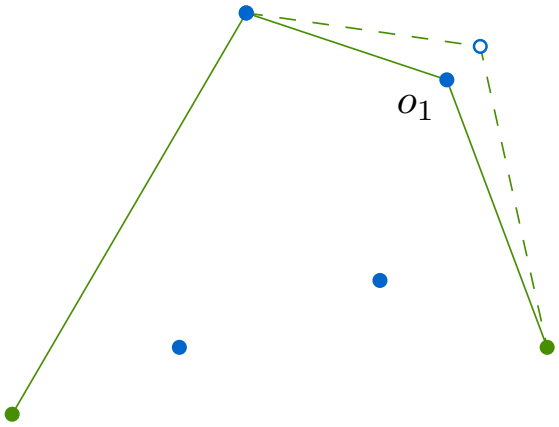


Figure 3.3: Sample instance showing that all obstacles on the upper convex hull created using the above obstacles between two set items, have to be used by the set path. The only way an obstacle like o_1 is not used by the final set path is if there is another above obstacle pushing the set path (and the upper convex hull) upward, meaning that o_1 is not on the upper convex hull in the first place.

Lemma 3.1.2. *Let O be a set consisting of two endpoints (obstacles or set items) already known to be used by the set path and all above (below) obstacles between them. All obstacles on the upper (lower) convex hull of O have to be used by the set path as well.*

Proof. W.l.o.g. assume that there is an obstacle o_1 on the upper convex hull between two items used by the path using the above obstacles between the set items, as an analogous argument can be made using below obstacles and the lower convex hull. Further, assume for contradiction that this obstacle is not used by the final path. Since it is an above obstacle we know that the final set path has to pass the obstacle to the north and since o_1 is not used in the final set path there has to be a space between o_1 and the final set path routed above o_1. Since the set path has to be the shortest path we can further conclude that there has to be another above obstacle above o_1 that pushed the path upward beyond o_1 and therefore beyond the upper convex hull. This means that there exists an above obstacle between the two set items and north of the convex hull. However, this cannot be the case as such an obstacle would have also been part of the upper convex hull. Therefore the assumption has to be wrong and o_1 is used by the final set path. □

Computing such a convex hull —and therefore finding obstacles that are necessarily used by the path between the set items— is straightforward using the Graham Scan algorithm [Gra72] described in Chapter 2 and can be done in $O(n \log n)$, where n is the number of (above or below) obstacles. The runtime of this can further be improved in most cases to $O(n \log h)$ by using the output sensitive Chan's algorithm [Cha96], where h is the number of obstacles on the convex hull.

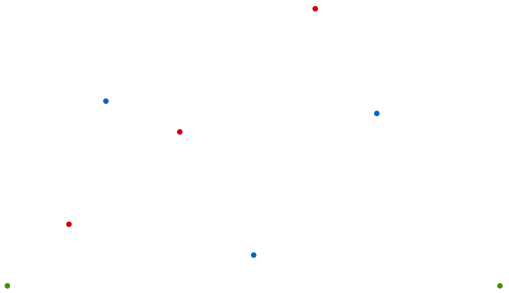
All obstacles on such a convex hull have to be used by the resulting set path, however not all obstacles used by the set path are necessarily found by the first computed (upper or lower) convex hull. To find all such obstacles this technique has to be applied multiple times recursively, similar to the method using extreme obstacles. The idea here is to start by computing the upper convex hull between the set items. Then, each of the obstacles found on the boundary of that convex hull splits the problem into subproblems, for which we then compute the lower convex hull. This procedure then is repeated (always flipping the computation of the upper and lower hull) until for every created subproblem both the upper as well as the lower convex hull do not contain any obstacles other than the start or end point of that subproblem. The detailed algorithm for this computation is given in Algorithm 3.1 and an exemplary run can be seen in Figure 3.4.

Algorithm 3.1: Basic Routing using Convex Hull

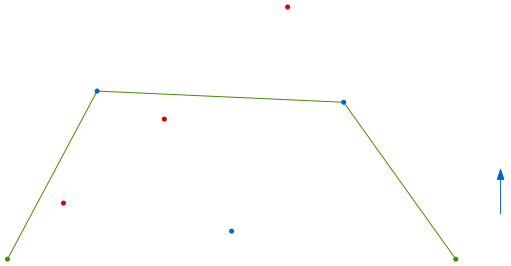
Input: A start set item *start*, an end set item *end* and a list of obstacles O between them

Output: A list of obstacles S used by the set path between *start* and *end*

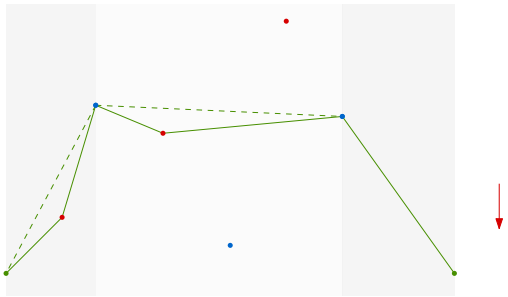
```
1  S ← {}
2  Recursion(S, start, end, true, false)
3  return S
4
5  Function Recursion(S, start, end, upper, twice):
6      C ← ConvexHull(start, end, upper)
7      if |C| = 0 then
8          if twice then
9              return
10         else
11             Recursion(S, start, end, !upper, true)
12         end
13     else
14         s ← start
15         for o ∈ C do
16             S ← S ∪ {o}
17             Recursion(S, s, o, !upper, false)
18             s ← o
19         end
20         Recursion(S, s, end, !upper, false)
21     end
22
23 Function ConvexHull(start, end, upper):
24     if upper then
25         return list of obstacles on the upper convex hull using all above obstacles
             between start and end ordered ascending by their x-coordinate
26     else
27         return list of obstacles on lower convex hull using all below obstacles
             between start and end ordered ascending by their x-coordinate
28     end
```



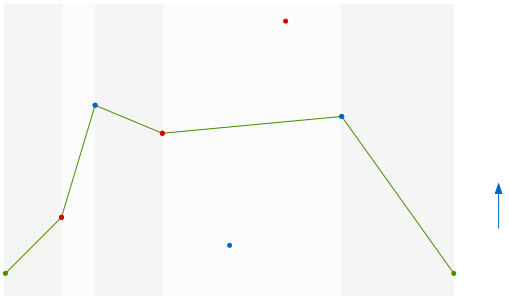
(a) Initial instance



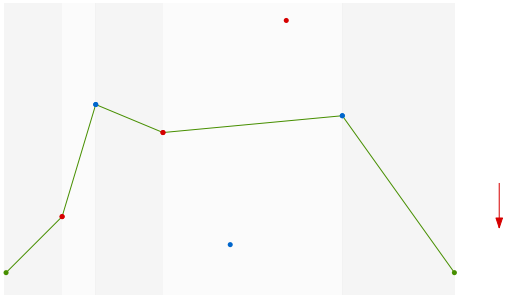
(b) Result of computation of first upper convex hull



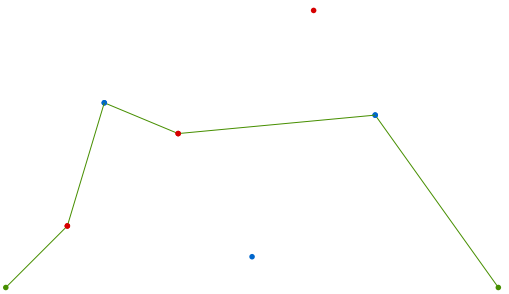
(c) Result after splitting the problem of (b) into subproblems and computing the lower convex hull for each



(d) Result after splitting the problems of (c) into subproblems and computing the upper convex hull for each



(e) Result after computing the lower convex hull for each of the subproblems of (d)



(f) Final result after combining the subproblems again

Figure 3.4: An exemplary run of Algorithm 3.1

It remains to be proven that all obstacles found by Algorithm 3.1 have to be used by the final set path and that only such obstacles are used by the path. Both of these properties can be derived from the previously given proofs.

Theorem 3.1.3. *Algorithm 3.1 correctly finds all obstacles used by a path in $O(n^2)$ time.*

Proof. First, we prove the correctness of Algorithm 3.1 by induction.

Induction hypothesis: Algorithm 3.1 correctly identifies all obstacles that have to be used by a set path for each (sub)problem with j obstacles between its two endpoints.

Base case: A (sub)problem without any obstacles (that have unfulfilled demands) between its two endpoints ($j = 0$). Algorithm 3.1 does not find any new obstacles and hence also does not create any new subproblems. Therefore, the induction hypothesis holds.

Induction step: A (sub)problem with any amount of obstacles (with unfulfilled demands) between its two endpoints ($j \geq 0$). We apply the `Recursion` routine from Algorithm 3.1 using the endpoints of the (sub)problem as *start* and *end*. Thanks to Lemma 3.1.2 we know that all obstacles found by the `ConvexHull` function must be used by the set path. At this point, there are two options.

If the `ConvexHull` function identifies new obstacles between the endpoints we use them to create new subproblems. At this point, we can apply our induction hypothesis to finish this case.

However, if the `ConvexHull` function does not find any new obstacles we apply the `ConvexHull` function for the inverted hull. Again, thanks to Lemma 3.1.2 we know that all obstacles found by this function must be used by the set path. This leaves us again with two options.

Similar to before, if the `ConvexHull` function identifies new obstacles between the endpoints we use them to create new subproblems and we can apply our induction hypothesis to finish this case.

Finally, if again there are no new obstacles found, then the upper and the lower hull w.r.t. the above and below obstacles respectively, consist of only the two endpoints of the instance. Hence, all obstacle demands of the (sub)problem are already met.

Running time. Computing the convex hull of one subproblem can be achieved in $O(n \log h)$ time using Chan's algorithm [Cha96] where n is the number of (above or below) obstacles and h is the number of obstacles on the computed convex hull. The number k of times the convex hull computation has to be executed is bound by the number of obstacles that are used by the final path as each new obstacle creates new subproblems. Since this number cannot exceed the number of obstacles, k is also bound by n.

Let h_i be the number of obstacles on the convex hull that Algorithm 3.1 finds in iteration i and assume that in every iteration $O(n)$ obstacles are considered for the convex hull computation. Then the overall running time of Algorithm 3.1 can be expressed as:

$$\sum_{i=1}^{k} O(n \log h_i)$$

Which is equivalent to:

$$O(n) \sum_{i=1}^{k} O(\log h_i)$$

Crucially we observe that $\sum_{i=1}^{k} h_i \leq n$ as the sum of all used obstacles by the set path cannot exceed the number of obstacles in the instance. Furthermore, since $\log h_i < h_i$, no matter how big h_i is, we conclude that $\sum_{i=1}^{k} \log h_i \leq n$. This leaves us with a worst-case running time of:

$$O(n) \cdot O(n) = O(n^2)$$

□

This worst-case running time for Algorithm 3.1 can happen if each subproblem finds at most one new obstacle, thereby creating two new subproblems. An example of how such an instance could look and how the presented algorithm handles it is given in Figure 3.5. In such a case the convex hull computation has to be executed $O(n)$ number of times and each subproblem consists of $O(n)$ obstacles. However, each computed convex hull has a fixed number of vertices (the two endpoints of the subproblem and the new obstacle) on its border. Therefore the overall runtime of the whole algorithm in such a case is $O(n^2 \log 3) = O(n^2)$.

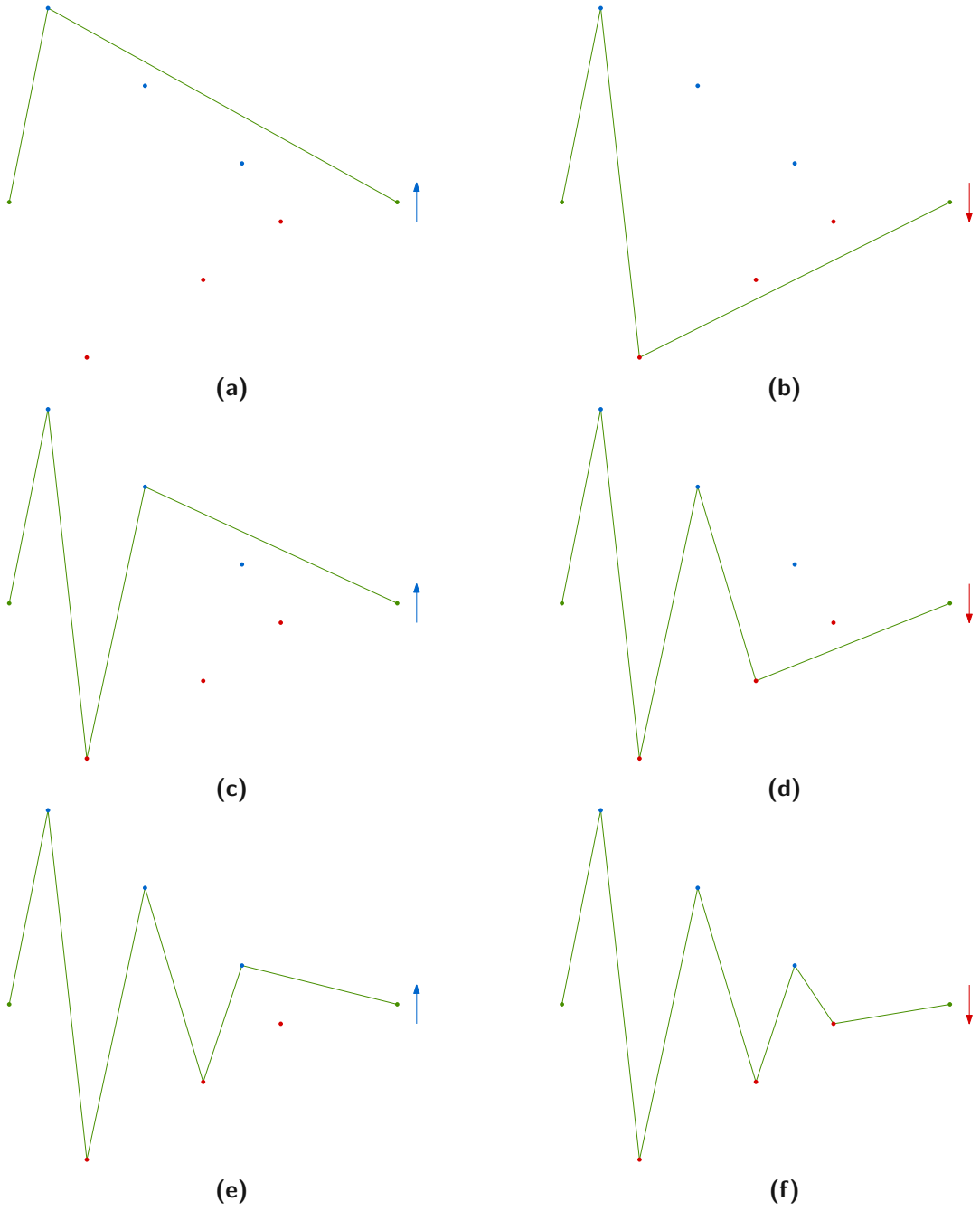
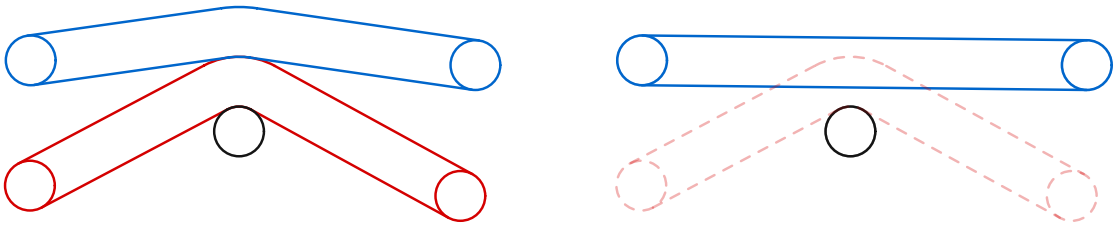


Figure 3.5: Exemplary instance showcasing the worst-case runtime of Algorithm 3.1



(a) The routing of the red edge increases the effective size of the obstacle in the middle. The blue edge has to adapt.

(b) Without the red set, the blue edge can be routed in a straight line between its endpoints.

Figure 3.6: Showcase how set paths with non-zero width can influence each other

3.2 Multiple Sets & Fat Edges

Up until now, we have covered the routing of a single hyperedge with zero width through obstacles that are represented as points in the plane. Provided an instance contains multiple edges not drawn as fat edges and we do not care if the edges of multiple sets overlap at some point we can run the same algorithm in isolation for each of the sets. This would result in a worst-case runtime of $O(sn^2)$ where s is the number of sets.

However, since we intend to introduce a width to the edges we have to keep in mind that routing an edge along an obstacle increases the size of the obstacle for all other sets (i.e. subsequent paths that use the same obstacle need to use different orbits). Since this increase in size can lead to obstacles becoming relevant for sets that they would not be relevant for when considered in isolation this is not a viable option. This problem is highlighted in Figure 3.6 and specifically, Figure 3.6a, where it can be seen that the blue set is only routed along an orbit of the obstacle in the middle because the red set is already using the innermost orbit of that obstacle. If the red set would not exist (like shown in Figure 3.6b) or would take another path (even if it is a small change like using the obstacle as a below obstacle) it would be possible to draw the blue edge as a straight, thick, line between its set items.

One way to circumvent this issue is to not compute the set paths individually but rather by intertwining the process. A method on how this is achieved is presented in Algorithm 3.2 and described in the following.

For this process, the sets are sorted from bottom to top. This is necessary because the sets in our instances are monotone (from left to right). Therefore, by sorting the sets from bottom to top, we create an ordering that enables the algorithm to generate a crossing-free output. With that in mind, the procedure starts by computing the upper convex hull for all sets starting with the bottom-most one. This first computation not only returns obstacles that have to be used by the set but because it is the upper convex hull we further know that these are the most northern obstacles the path will use even in its final form. Additionally, we can be sure that no other sets have to be routed between the set and the obstacles, as we know that a crossing-free output exists and we have

started with the bottom-most set. Therefore, we know that this set will not be influenced by other sets around these obstacles and we can be certain about the potential influence of this set on subsequent path computations. This potential influence is handled by increasing the effective radii of the obstacles when checked by the convex hull method for subsequent paths. When considering an obstacle for a convex hull, its effective radius is increased by the width of the already computed paths that use this obstacle. Therefore, if we now go on and compute the upper convex hull of the next set we can be sure that the obstacles used there either already account for the sets below or the sets below will never use them, as they are above the northernmost obstacles in their path.

After the algorithm has computed the first upper convex hull for each of the sets, it switches to computing the lower convex hull of all the subproblems generated thus far. For this computation, however, the algorithm also reverses the vertical order of the sets (starts with all subproblems generated by the topmost set). This procedure (switching between computing the upper and lower convex hull and reversing the set order) is then repeated until there are no more subproblems left.

An example of this procedure can be seen in Figure 3.7, in which the middle obstacle is a below obstacle for both sets and the other obstacles are above obstacles for both sets. Figure 3.7b shows what the visualization would look like after computing only the first upper convex hull of each set. In Figure 3.7c the algorithm has already computed the lower convex hull for all subproblems of the blue set. Still, it has yet to compute the lower convex hull for the subproblems generated by the red set in the previous iteration. This step highlights how crucial the correct order is for the algorithm, as it is only possible to route the red set correctly around the middle obstacle as shown in Figure 3.7d if the blue set was processed before it.

The running time of this intertwined method of computing the set paths is the same as computing every set path in isolation as the steps being taken are identical. Only the order of the steps taken by the algorithm is interleaved instead of sequential, which is crucial for the procedure to give the desired output.

3.3 Squishing

Unfortunately, the method described in the previous section is not without its flaws. One obvious challenge that arises by using a non-zero width for the paths is that the space between obstacles can be limited. Therefore it is not always possible to draw all edges with the same width uniform along their whole paths. One option to deal with this issue is the introduction of variable edge width. By adapting the edge width variably along the path it is possible to "squeeze" or "squish" the edges where necessary, while still allowing them to extend to their desired width where possible. From a computing perspective, there are two fundamental approaches to how this squeezing can be incorporated into the algorithm. Either all paths are computed upfront with zero width and then expanded to their desired width where possible or the path computation already starts with the

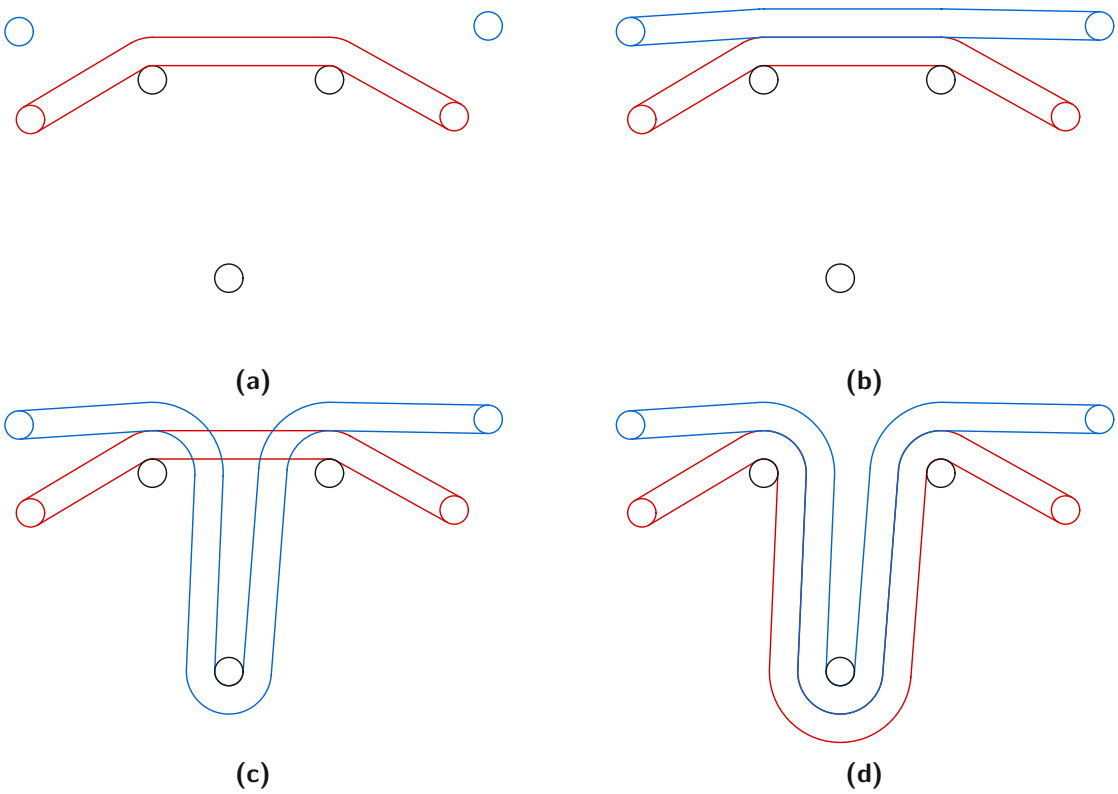


Figure 3.7: Instance highlighting the importance of the order in which subproblems are computed when dealing with multiple sets

target width of each path, like the previously described algorithm, and shrinks the edge width where necessary. In the following, both options are explored.

Algorithm 3.2: Intertwined Algorithm using Convex Hull (+ Squishing)

Input: A list of sets S, sorted from bottom to top
Output: A list of obstacles S_i for each of the sets in S used by their set path

```
1  upper = true
2  P ← {}
3  for s ∈ S do
4  |  P ← P ∪ {Subproblem(s := s_start, e := s_end, t := false)}
5  end
6  while |P| ≠ 0 do
7  |  T ← {}
8  |  for p ∈ P do
9  |  |  C ← ConvexHull(p_s, p_e, upper)
10 |  |  if |C| = 0 then
11 |  |  |  if p_t = false then
12 |  |  |  |  T ← T ∪ {Subproblem(s := p_s, e := p_e, t := true)}
13 |  |  |  end
14 |  |  else
15 |  |  |  t ← p_s
16 |  |  |  for o ∈ C do
17 |  |  |  |  S_i ← S_i ∪ {o}
18 |  |  |  |  IncreaseRadius(o)
19 |  |  |  |  Squishing(o)
20 |  |  |  |  T ← T ∪ {Subproblem(s := t, e := o, t := false)}
21 |  |  |  |  t ← o
22 |  |  |  end
23 |  |  |  T ← T ∪ {Subproblem(s := o, e := p_e, t := false)}
24 |  |  end
25 |  |  P ← P \ p
26 |  end
27 |  for t ∈ T do
28 |  |  P ← P ∪ {t}
29 |  end
30 |  upper =!upper
31 end
32
33 Function IncreaseRadius(obstacle):
34 |  Increase the radius ConvexHull uses when considering obstacle by the size
   |   of one orbit
35
36 Function Squishing(obstacle):
37 |  diskObstacle ← disk of influence of obstacle
38 |  for o ∈ O do
39 |  |  diskO ← disk of influence of o
40 |  |  if diskO intersects diskObstacle then
41 |  |  |  d ← distance between boundary of o and obstacle
42 |  |  |  k ← sum of sets on the orbits of o and obstacle
43 |  |  |  decrease the radius of each orbit of o and obstacle to d/k
44 |  |  end
45 |  end
```

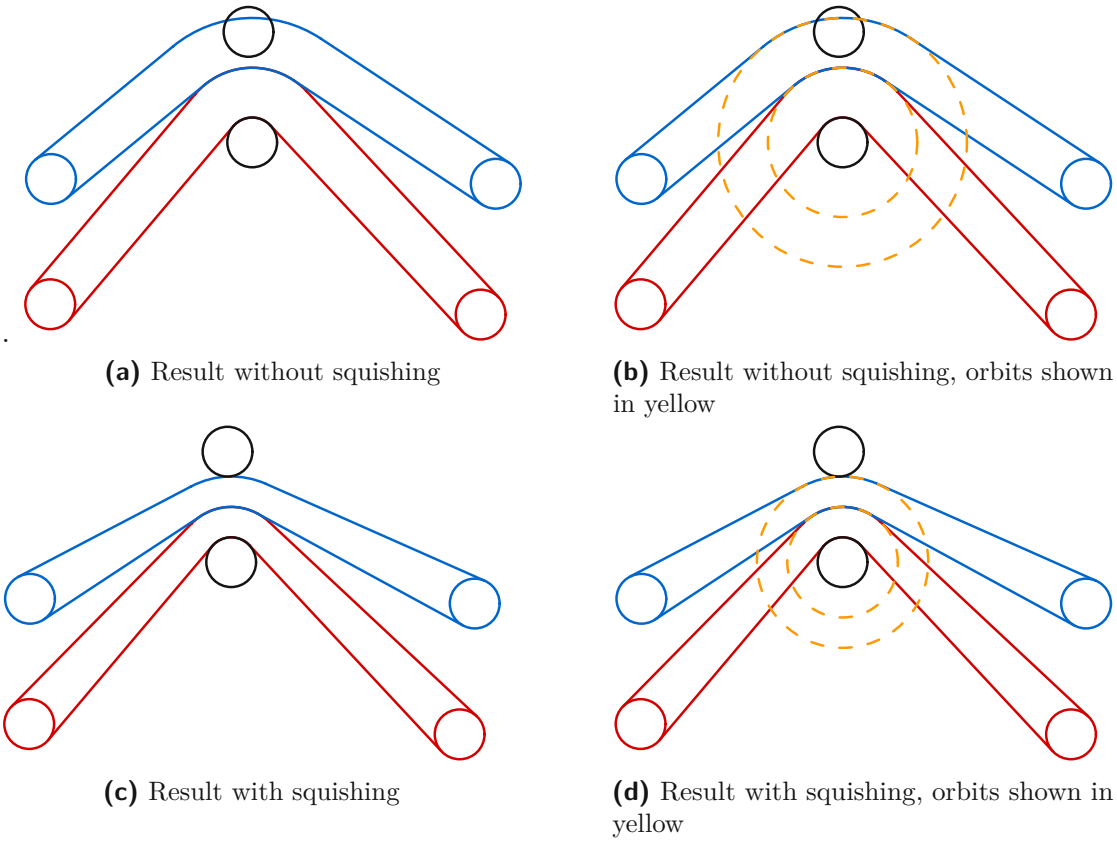
3.3.1 Alternating hull algorithm

Starting the computation of the path routing with edges that already have the desired width is an intuitive approach. Since this way the paths already have some width during the routing, the occupied space around obstacles can already be accounted for by subsequently routed sets. However, starting the routing algorithm explained above with edges that have a non-zero width comes with a set of drawbacks and challenges of its own. Most obviously there are instances where Algorithm 3.2 (without extension) will not be able to find a correct solution for a given path width, without variably adjusting the width. This can for example happen if sets are routed through a gap between obstacles and the size of the gap is smaller than the sum of the path widths.

An example of this can be seen in Figure 3.8, where Figure 3.8a and 3.8b show how the convex hull method would route the sets after computing the upper convex hull for each set once. In this example, the lower obstacle is an above obstacle and the upper obstacle is a below obstacle for both sets. Continuing with Algorithm 3.2 (without extension) at this stage would be pointless since there is not enough space between the two obstacles to properly route the sets. Therefore the width of the paths has to be reduced (i.e. the sets have to be *squished*) to both fit between the obstacles. However, since the goal is to draw the edges as thick as possible instead of uniformly decreasing the edges' width along their whole path we only shrink them between the two obstacles creating the bottleneck. Since the sets follow the orbits surrounding the obstacles used by their path, this squishing can be achieved by reducing the radii of these orbits as seen in Figures 3.8b and 3.8d. When two obstacles create such a bottleneck for the sets routed between them we speak of them influencing each other.

Squishing Extension. Algorithm 3.2 (without extension) has no way of knowing which obstacles influence each other and therefore cannot handle the squishing of the paths. The idea to fix this issue is to introduce safe zones around each obstacle, called *disk of influence*, in which only sets routed along the orbits of that obstacle can exist. In their simplest form, these disks of influence describe a circle with the same center point as the obstacle itself and extend the radius up to the outermost set path routed along the obstacle. Disks of influence allow us to detect whether obstacles influence each other by checking if their disks of influence overlap each other. We use this check to extend Algorithm 3.2 such that each time an obstacle is added to a set path the algorithm is able to see if squishing is necessary. This way Algorithm 3.2 can fix an overlap of disks of influence, that possibly indicates an intersection, by shrinking the radii of the orbits around the involved obstacles proportionally to how many sets each of the obstacles carries on its orbits, thereby shrinking their disks of influence to such an extent that they do not overlap each other anymore.

Any sizing of the two disks of influence that ensures that there is no overlap between the two also ensures that the set paths do not intersect between the two obstacles. Proportional shrinking allows all involved set paths to have uniform width at the point



(a) Result without squishing

(b) Result without squishing, orbits shown in yellow

(c) Result with squishing

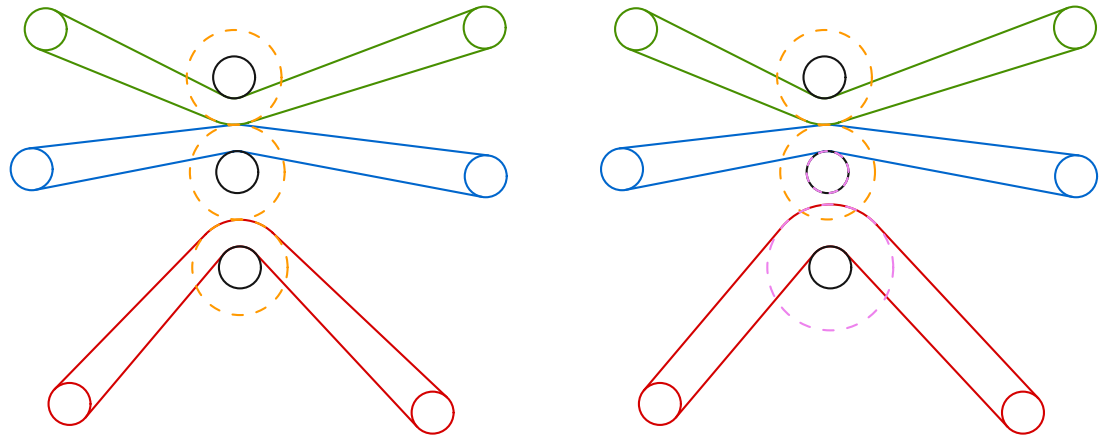
(d) Result with squishing, orbits shown in yellow

Figure 3.8: Instance showing that variable edge width is needed to route sets through bottlenecks created by obstacles

of squishing. However, shrinking in proportion is a design choice we made and not a necessity for the algorithm to work.

Using the maximal radius of any set path around each obstacle as their disk of influence has the advantage that it is easily computable. Furthermore, this means that the disk of influence of an obstacle is the same no matter against which obstacle it is compared to. However, this simplification also has a downside, which can be seen in Figure 3.9. Specifically, Figure 3.9a shows that the red set path is squished along the bottom obstacle even though, looking at the actual routing of the paths, there is no necessity for it. In contrast to that Figure 3.9b shows how the drawing could look if the disks of influence would also include the context of the obstacle they are compared against in their computation. In this scenario, the disk of influence of the middle obstacle is the same size as in Figure 3.9a when compared to the top obstacle, but only the size of the obstacle itself when compared to the bottom obstacle. Therefore the red path can extend to its intended size along the bottom obstacle.

It is, however, not guaranteed that all set paths that use an obstacle still need to use



(a) Choosing the outermost orbit as the disk of influence for all other obstacles

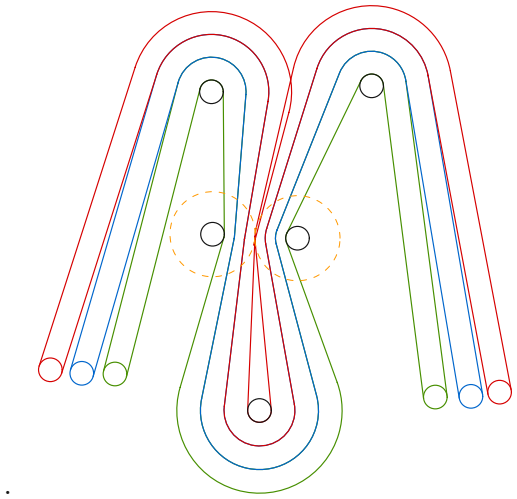
(b) Choosing different disks of influence depending on which obstacles are involved in the computation

Figure 3.9: Impact of the difference between the computation methods for the disks of influence

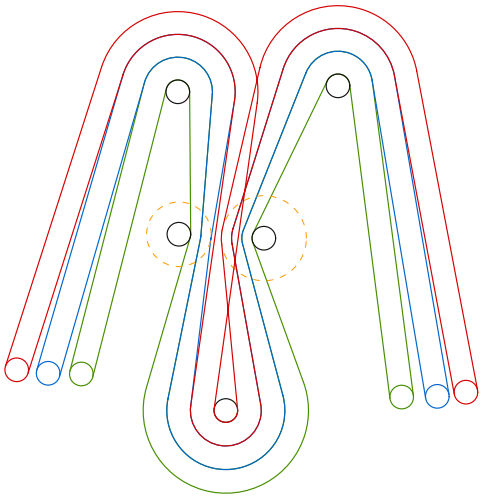
the obstacle after squishing is applied, as squishing decreases the overall footprint of an obstacle. An example of this can be seen in Figure 3.10a where, after the squishing between the two middle obstacles is applied, the red set makes an inward bend (the outer angle is smaller than 180 degrees) on both sides of the path at the orbit of the left obstacle. The convex hull method therefore would have never routed the red path along this obstacle in the first place.

Resolving inward bends. There are multiple approaches to how this issue can be solved, each of them affecting the visualization in different ways, leading to different outcomes. One way is to simply reduce the orbit radii of neighbouring obstacles, such that the outer angle on the outermost set of the obstacle is not smaller than 180 degrees. However, since this method reduces the orbit radii of other, not yet involved obstacles, it could lead to a chain reaction, creating multiple instances of the same issue. Another approach would be to remove the obstacle from the set path at which the inward bend is happening and instead replace it with the obstacle it was squished against. The downside of this approach is that it would require a more complex data structure and adaption of the algorithm, as the data structure would need to be able to use obstacles multiple times for each set path and even change their orientation for a set throughout its path.

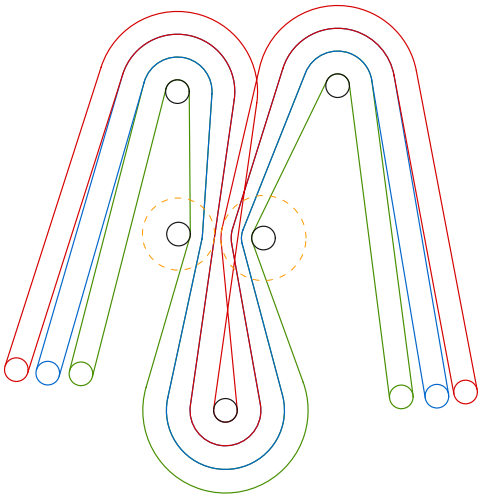
The approach we use in our implementation removes the obstacle at which the inward bend occurs from the set path. This can seem counterintuitive at first, as it allows the path to cross the orbit of the other obstacle involved in the squishing as is visible in Figure 3.10b. However, this crossing can easily be resolved in a post-processing step, by reducing the orbit radii of the right obstacle, as described in Section 3.3.2. After the set containing the inward bends on both sides of its path is removed from the obstacle, the



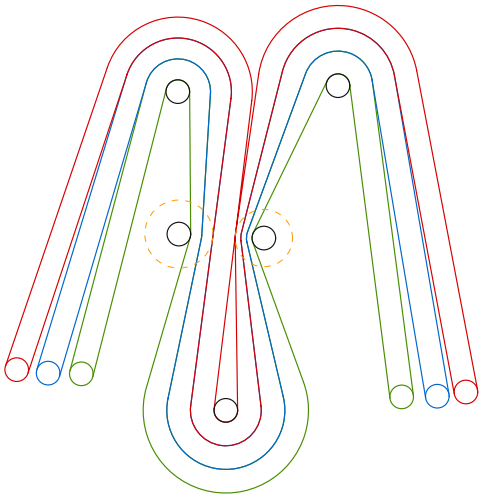
(a) Squishing between middle obstacles creates inward bends on the outermost orbit of left obstacle



(b) Removing left obstacle from red set path resolves the inward bend, but creates crossing with other obstacle involved in squishing



(c) Extending left obstacle's orbit radii such that the outermost set has a 180-degree outer bend



(d) Instance after final post-processing applied.

Figure 3.10: Instance showing how inward bends created by squishing are handled in the program

issue remains that the now outermost set on the obstacle's orbits still has an inward bend on its outer path side, like the blue set in Figure 3.10b. However, since we have removed one set from the obstacle's orbits that was in part responsible for the smaller radii of the orbits, we can now increase the orbit radii again. Overall the obstacle's orbit radii can be safely increased until the outer angle of the outermost set is exactly 180 degrees, making it sit flush against the previously removed set, as can be seen in Figure 3.10c.

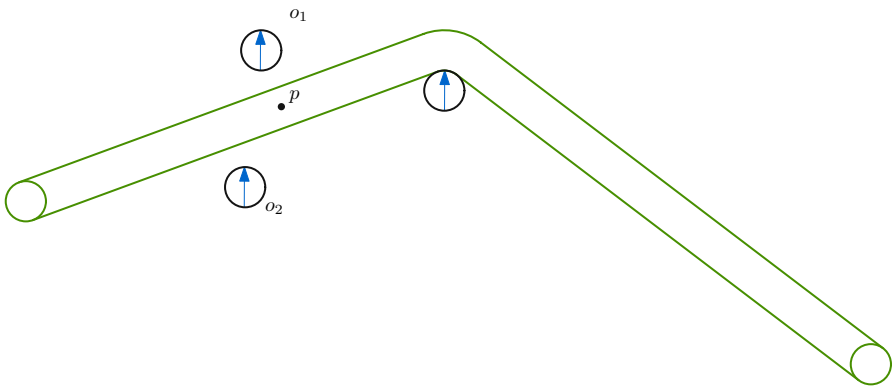


Figure 3.11: Orientation check at position p for the set returns false for obstacle o_1 and true for obstacle o_2

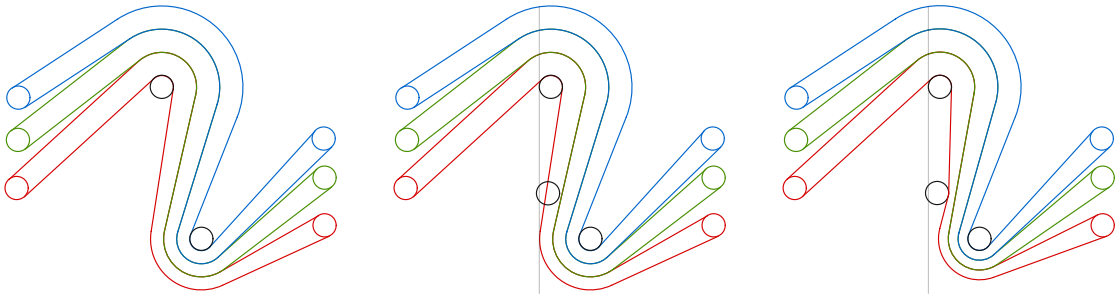
3.3.2 Post Processing

The initially described Algorithm 3.2 with its squishing extension described in the previous section is capable of handling and routing thick edges through multiple obstacles in many instances. However, the non-zero width paths introduce further problems that cannot be handled easily by the algorithm and we, therefore, choose to fix in a post-processing step.

The post-processing algorithm relies on an *orientation check*. This check compares an obstacle against a specific part of a set path. Specifically, it tests if the north or the south side of the set path at the specified position is closer to the obstacle and compares this information against the obstacles demand regarding this set. If and only if the obstacle demand for the set is to be routed above (respectively below) and the south (respectively north) side of the path is closer to the obstacle the orientation check is positive. An example of this check can be seen in Figure 3.11.

One of the remaining issues after Algorithm 3.2 is executed can be observed in Figure 3.12. Looking at Figure 3.12a we see three different sets whose paths are routed by the former described method through one above and one below obstacle (the obstacles have the same demands for all sets). In this case, the algorithm works as expected. However, Figure 3.12b highlights an issue of this method, by introducing a new above obstacle (again with the same demand for all sets), which now intersects with the paths routed by the algorithm. The main issue here is that the newly introduced obstacle is slightly left to the topmost obstacle. Because we assume monotonicity this obstacle is therefore not considered in the relevant subproblem between the other two obstacles. However, since we are using a non-zero width for the set paths it is possible that the set paths extend beyond the borders of the end points of their subproblem, leading to such cases.

Pushing. To solve this issue a new step in the post-processing, called *pushing*, is introduced that is executed after the Algorithm 3.2 is finished. Pushing, which is part of the post-processing Algorithm 3.3, checks all obstacles and their orbiting sets for similar



(a) Instance with a correct solution produced by Algorithm 3.2

(b) Instance with a faulty solution produced by Algorithm 3.2

(c) Instance of Figure 3.12b after post processing

Figure 3.12: Instance showing how pushing can solve intersections not handled by Algorithm 3.2

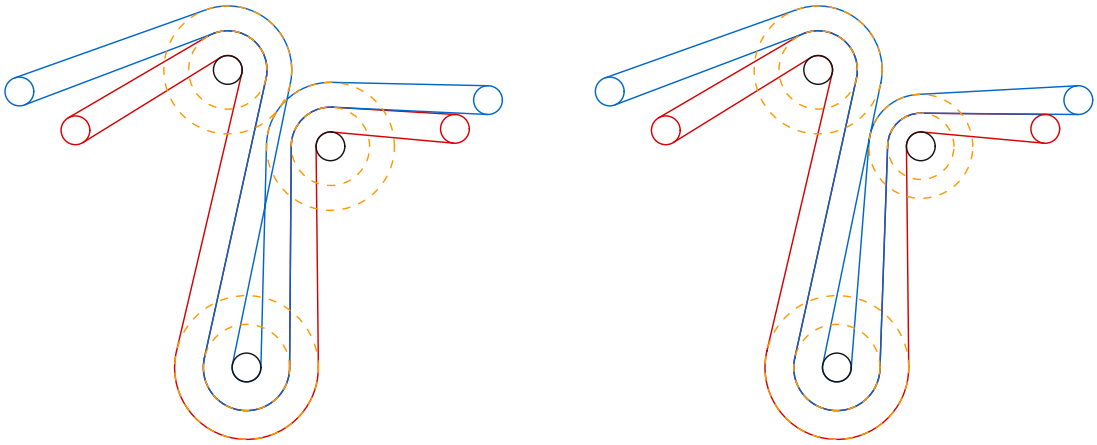
intersections as pictured in Figure 3.12b. Specifically, it searches for set paths that cross an obstacle or the paths on the obstacle's surrounding orbits within its disk of influence and for which the orientation check is positive. This orientation check is important because it prevents this post-processing step from being executed in situations like the one shown in Figure 3.10c, where the red set path crosses the middle right obstacle's orbit, but the orientation check is negative. Instead, these cases are handled by a different post-processing step described below.

If the orientation check is positive (like in Figure 3.12b), the obstacle whose disk of influence is intersected is added to the set path of the intersecting set(s). This is possible because thanks to the orientation check we know that the set path(s) are not using the obstacle in a different direction yet. Adding sets to an obstacle's orbits of course also affects its disk of influence. Therefore the squishing technique previously described has to be executed again at this point for the specific obstacle.

Inverse Pushing. Another issue, not handled by the intertwined convex hull method in combination with squishing, is shown in Figure 3.13. Specifically, in Figure 3.13a it can be observed that two paths can intersect with each other even though the orbits around the nearest obstacles are disjoint. To circumvent this problem another post-processing step called *inverse pushing*, which is also part of Algorithm 3.3, is introduced.

In this step, all obstacles and their surrounding orbits are checked for intersections similar to the one seen in Figure 3.13a. Specifically, this step searches for set paths that cross an obstacle or the paths on the obstacle's surrounding orbits within its disk of influence and for which the orientation check is negative. Thanks to the negative orientation check we know that the possibility exists, that the set already uses the obstacle with a different direction. Therefore, we handle these cases distinct from regular pushing to avoid obstacles being used more than once and in different directions by the same set.

This is done by reducing the radii of the corresponding obstacle's orbits such that they



.

(a) The blue set has an intersection with itself even though the orbits of the associated obstacles are disjoint

(b) Reducing the radii of the associated obstacle to only extend up until the set path that was intersected solves the issue

Figure 3.13: Instance showing that set paths can overlap even though their surrounding orbits are disjoint

do not intersect with the set path anymore, as seen in Figure 3.13b. The new size the obstacle's orbits can occupy is calculated by measuring the distance between the set path that caused this inverse pushing to happen and the boundary of the obstacle itself in a straight line that is normal against the set path. Resolving inward bends, as described in Section 3.3.1, always results in negative orientation checks, and hence these overlaps are also resolved with inverse pushing. An example of this can be seen in Figure 3.10d, where inverse pushing creates the outcome that can be seen in Figure 3.10d.

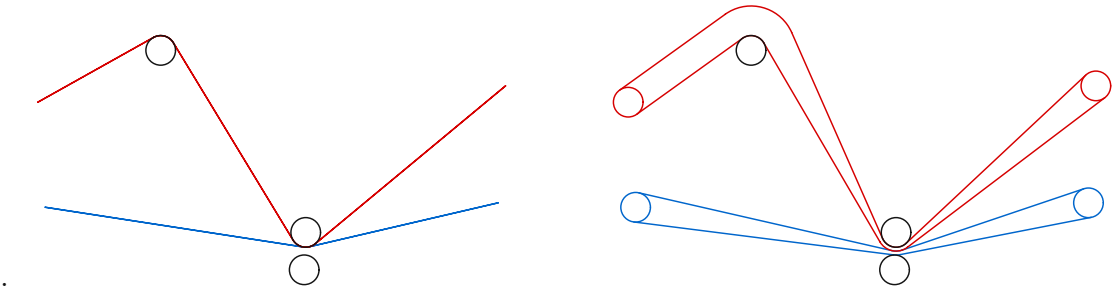
It is important to note, that both of these post-processing steps will create outputs that differ from the optimal set path routing design we have set out to create (thick paths where possible, evenly distributed squishing where necessary). However, they allow us to keep the complexity of the algorithm manageable, while still producing correct set path routings, that strive to adhere to these design principles where possible. Furthermore, these post-processing steps are in no way unique, meaning that different methods of solving the presented issues that may produce other visual outcomes are likely to exist.

The main idea behind Algorithm 3.2 is to alternately use the upper and lower convex hull to achieve a set path routing that also works with thick paths thanks to the squishing extension. Therefore, we will refer to this algorithm, in combination with the post-processing described in Algorithm 3.3, as *alternating hull algorithm*.

Algorithm 3.3: Post Processing

Input: An instance created by Algorithm 3.2
Output: A version of the instance without any paths intersecting each other and/or obstacles

```
1  for o ∈ O do
2  |  if Pushing(o) = true then
3  |  |  Squishing(o)
4  |  end
5  end
6  for o ∈ O do
7  |  InversePushing(o)
8  end
9
10 Function Pushing(obstacle):
11 |  added ← false
12 |  for s ∈ S do
13 |  |  intersect ← Intersect(s, obstacle)
14 |  |  if intersect ≠ null then
15 |  |  |  if Orientation(s, obstacle, intersect) = true then
16 |  |  |  |  Add o to the set path of s at position of intersect
17 |  |  |  |  added ← true
18 |  |  |  end
19 |  |  end
20 |  end
21 |  return added
22
23 Function InversePushing(obstacle):
24 |  for s ∈ S do
25 |  |  intersect ← Intersect(s, obstacle)
26 |  |  if intersect ≠ null then
27 |  |  |  if Orientation(s, obstacle, intersect) = false then
28 |  |  |  |  path_s ← section of path of s that intersects with obstacle
29 |  |  |  |  dist ← distance(path_s, obstacle)
30 |  |  |  |  Set sum of radii of orbits of obstacle to dist
31 |  |  |  end
32 |  |  end
33 |  end
34
35 Function Intersect(set, obstacle):
36 |  path ← the path of set as well as the arcs of the sets routed along obstacle
37 |  return intersection(path, obstacle)
38
39 Function Orientation(set, position, obstacle):
40 |  path_s ← south side of path of set at position
41 |  path_n ← north side of path of set at position
42 |  if demand(set, obstacle) = above then
43 |  |  return distance(path_s, obstacle) < distance(path_n, obstacle)
44 |  end
45 |  if demand(set, obstacle) = below then
46 |  |  return distance(path_s, obstacle) > distance(path_n, obstacle)
47 |  end
```



(a) Initially the paths are computed with zero width

(b) The routed paths are then expanded until every further expansion would lead to an intersection.

Figure 3.14: Instance showing the expansion method

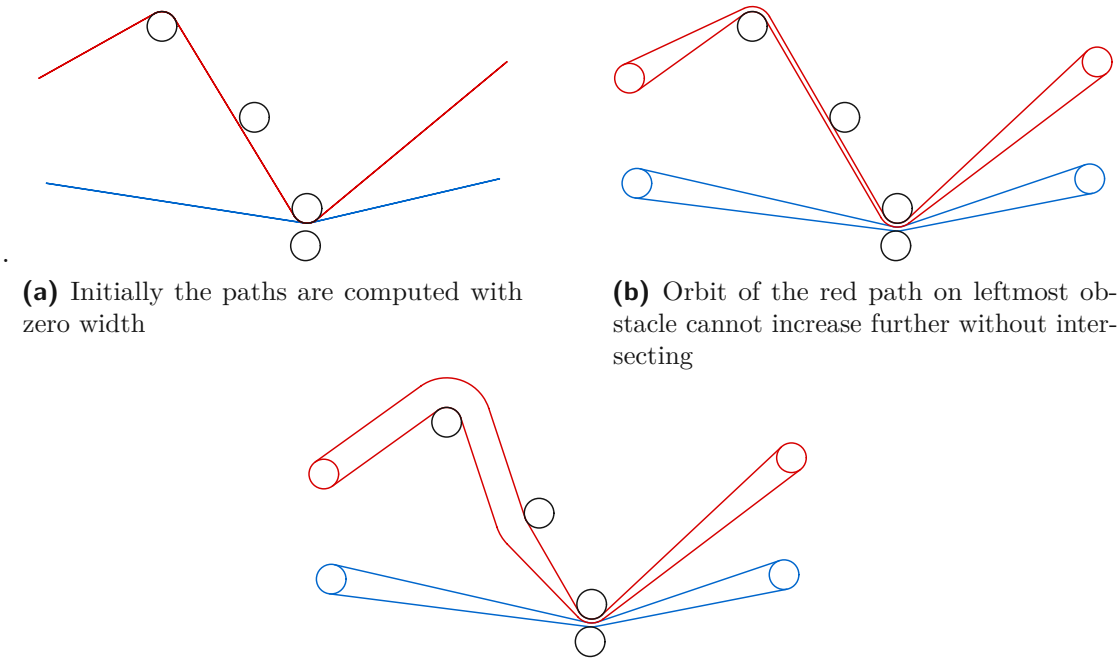
3.3.3 Expansion method

A second approach to prevent any issues with fat sets overlapping obstacles or each other is to start the computation with an initial path routing that uses zero-width paths as shown in Figure 3.14a. This allows for a much simpler routing algorithm, as the obstacles do not change in size when an edge is routed on one of their orbits and the algorithm, therefore, does not need to consider squishing at all. Furthermore, since the algorithm starts with zero-width paths and the paths do not influence each other, every set can be computed in isolation. This is in contrast to Algorithm 3.2 that specifically required a certain order in which the subproblems of each set are computed, such that their influences on each other are taken into account.

Once the initial routing is computed for each set the algorithm then proceeds to expand the orbits around each of the obstacles. This is achieved by increasing the radius of each of the obstacles carrying at least one of the sets on its orbits. For each of these obstacles we increase all its orbit radii by some value x, one obstacle at a time. The order in which the obstacles' orbit radii are increased is arbitrary. In our implementation, we choose to order the obstacles from left to right.

After each increase, the path segments adjacent to the obstacle are checked for intersections with the surrounding obstacles and paths. If such an intersection is found then the increase of that obstacle's orbit size is directly responsible for it and therefore needs to be decreased by x again. This procedure then is repeated until none of the obstacle orbit radii can be increased by x anymore or if the obstacle orbit radii have reached their target size. At this point, we set $x = \frac{x}{2}$ and repeat the whole procedure. This is done until $x <= t$, where t is a threshold value set by the user. The result of this algorithm is a visualization using fat edges, in which increasing any obstacle's orbit radii by t would lead to an intersection of one of the paths on the obstacle's orbit with another edge or an obstacle as seen in Figure 3.14b.

One downside of this approach is that it is possible to generate instances, where the procedure is not able to increase the size of an obstacle's orbit, even though there would



(a) Initially the paths are computed with zero width

(b) Orbit of the red path on leftmost obstacle cannot increase further without intersecting

(c) Visualization computed by Algorithm 3.2

Figure 3.15: Instance showing the limitations of the expansion method

be enough space around it. An example of this can be seen in Figure 3.15b, where the red set path could be much thicker around the first two obstacles. However, in the initial routing (seen in Figure 3.15a) the red set does not use the second left-most obstacle. Therefore the size of the radius of the orbit around the left-most obstacle can only increase so much, without creating a possible intersection with the second left-most obstacle. Figure 3.15c shows what the same instance would look like when computed with the previously introduced alternating hull algorithm, which does not have this issue.

Since this approach starts with zero-width paths and only starts to expand the orbit radii after it computed the set path routing, we will refer to this approach as *expansion method* for the rest of this thesis.

CHAPTER 4

Kinetic Visualization

Until now we focused on the creation of visualizations for static instances, meaning that each set item and obstacle has a fixed position in space. However, one of our previously defined goals is to make this process as efficient as possible, such that it could also be applied in a kinetic environment, where the position of set items and obstacles can change over time. This creates additional challenges since the routing of the set paths as well as their thickness has to adapt to this movement.

This chapter, therefore, describes how the algorithms for static instances and the learnings we took from them can be applied in a kinetic environment, where the main challenge is to properly handle this time-varying data (i.e. make proper adjustments to set path routing and orbit radius sizes to act on the movement of set items and obstacles).

In the following, instances containing such time-varying data will be referred to as *kinetic instances*. As a first step towards the visualization of unrestricted kinetic instances, in this thesis, we limit the movement to the obstacles (i.e. the set endpoints themselves do not move). Furthermore, each obstacle is only allowed to traverse a single straight line during each movement operation and finally, the velocity with which the obstacles in an instance move is defined by an overall time limit, rather than setting a fixed velocity with which the obstacles change their position.

The main challenge of visualizing kinetic instances is that instead of computing and rendering a single state, the visualization now needs to render multiple frames on (continuously) changing input. In each of these frames, the position of at least one obstacle changes. Depending on the obstacle's movement and its surroundings this can have a multitude of different implications on the visualization that need to be accounted for. For example, if a set path is added to the orbits of an obstacle because of its movement, the size of the disk of influence of that obstacle also increases. This, in turn, creates the necessity to reassess if the obstacle (and specifically its orbits) need to be squished.

Furthermore, the aesthetic criteria defined for static visualization are not necessarily the most important criteria when judging the quality of the visualization during the movement in an instance. As an example, the algorithms for static instances presented in Chapter 3 in general strive to maximize the path width of each set throughout the instance. However, in a kinetic instance, we might want to prioritize the stability of the model (i.e. prevention of discrete changes) over maximizing the used space.

There are a variety of different approaches, with varying degrees of implementation simplicity and computational efficiency, to algorithms that deal with the movement of obstacles and the implications thereof. In the following, we describe a simple, but inefficient naive approach as well as an interpolation approach that requires more insight but is more efficient.

4.1 Recomputation Method

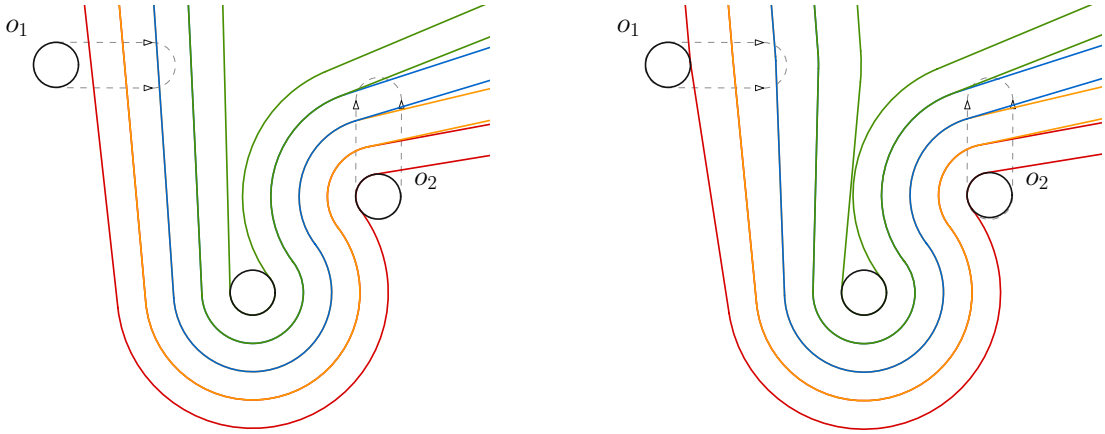
One way to implement the visualization of kinetic instances is to recompute the visualization for each rendered frame. This can be achieved by computing the new positions of the obstacles for each frame and then running the alternating hull algorithm described in Section 3.3.1. Although easy to implement, this approach has some downsides that need to be considered.

For one, re-computing the algorithm for each frame is computationally an expensive endeavour. This is because, among other things, this approach computes each frame as a self-contained instance and does not utilize information from previous frames, i.e. the algorithm is *stateless*.

Another issue is that this stateless approach can lead to discrete changes during the movement. An example of such a discrete change occurring can be seen in Figure 4.1, which shows a part of two consecutive frames of a kinetic instance, visualized using a naive movement algorithm that computes Algorithm 3.2 with the post-processing of Algorithm 3.3 for every frame.

In this example, there are two moving obstacles. The discrete change in this instance happens because in the frame of Figure 4.1b the static algorithm decides to route the set paths along the orbits of obstacle o_1, even though the obstacle o_1 would not intersect with the red set path yet if the obstacle would not have been used by the red set path. The change is especially disturbing because the orbits used by obstacle o_1 in Figure 4.1b are larger than the thickness of the set paths at the same y-coordinate in Figure 4.1a, which amplifies how disruptive the discrete change is perceived.

In a static instance, this is not an issue and actually can be a desired outcome, as it leads to overall thicker paths and less space left unused. However, in a kinetic instance, this behaviour leads to a discrete change of the path widths as visualized in Figure 4.1, which is counterproductive to the goal of creating a smooth movement.



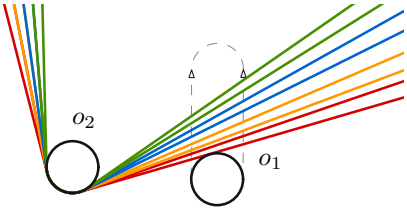
(a) Frame just before the discrete change **(b)** Frame after the discrete change

Figure 4.1: Example of a discrete change during movement using the naive approach

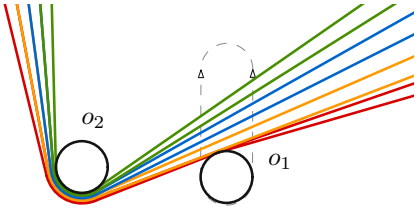
Expansion Method. The same naive approach of re-computing the static algorithm for each frame can also be applied using the expansion method detailed in Section 3.3.3. However, the problems with discrete changes using this naive approach are even more apparent as illustrated in the four consecutive states shown in Figure 4.2. Because of the way the expansion method works (reduce edge width to prevent intersections with obstacles that are not part of the zero-width routing) the orbit radii of obstacle o_2 are narrowed at first until they reach a width of zero as shown in Figure 4.2a. At that point in time, the zero-width set path of the red set is however also routed along the moving obstacle o_1, which allows the obstacle o_2 to expand its orbit radii again. This expansion happens instantly as soon as the red set is routed along the orbit of obstacle o_1, leading to the discrete change in orbit radii of obstacle o_2 going from the frame shown in Figure 4.2a to the frame shown in Figure 4.2b.

The expansion algorithm then tries to prevent an interception of the orange set with the upward moving obstacle o_1 and therefore starts to shrink the orbit radii of o_2 again until the zero-width path of the orange set uses the obstacle o_1. At that point, again, the orbit radii of o_2 are instantly increased, leading to the discrete change going from Figure 4.2c to Figure 4.2d. This behaviour is repeated for each of the sets that obstacle o_1 crosses along its way, leading to a series of discrete changes in the visualization.

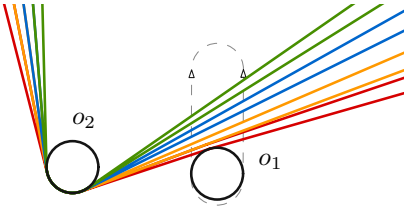
Furthermore, the expansion of the orbit radii of o_2 is larger with each additional set routed along the orbits of obstacle o_1, as is visualized in Figure 4.2d when compared to Figure 4.2b. Therefore the visual impact of these discrete changes is also more substantial each time it happens.



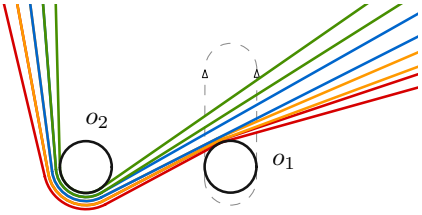
(a) Obstacle o_2 shrinks its orbit radii, such that the red set does not intersect with the obstacle o_1



(b) The zero-width path of the red set uses obstacle o_1, therefore the orbit radii of obstacle o_2 can expand again



(c) Obstacle o_2 shrinks its orbit radii again, such that the orange set does not intersect with the obstacle o_1



(d) The zero-width path of the orange set uses obstacle o_1, therefore the orbit radii of obstacle o_2 can expand again

Figure 4.2: Example of a kinetic instance with a multitude of discrete changes using the expansion method as the static algorithm for the naive movement approach

4.2 Interpolation Algorithm

An approach to visualizing motion, that has the potential to be more efficient, is to interpolate the states during the movement between the start and the end configuration of the kinetic instance. The main advantage of such an approach is that, contrary to the naive algorithm described in Section 4.1, this approach theoretically only needs to run the whole static algorithm twice (for the start and the end state), as the states of the frames between those can be interpolated.

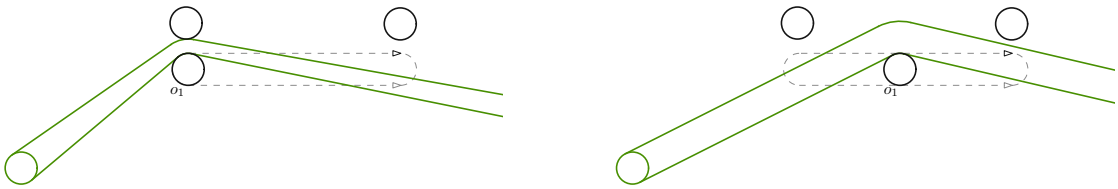
The interpolation itself is done by adjusting the radii of the orbits of affected obstacles. An example of a kinetic instance that works with this approach can be seen in Figure 4.3. For each frame of the movement between the start state in Figure 4.3a and the end state in Figure 4.3b an interpolation can adjust the orbit size of the affected obstacle accordingly using linear interpolation, as the size of the orbit is linearly dependent on the distance between the two obstacles squishing each other.

However, because of multiple reasons explained in the following, this theoretical approach of interpolating between a start and an end state, as pictured in Figure 4.3, in practice requires more effort. One of the issues encountered with this approach can be seen in Figure 4.4. During the movement of obstacle o_1 in this instance the orbit radius of o_1 expands from small to big first, however towards the end of the movement, the radius decreases again to fit between the squishing obstacle and is roughly the same size as in the start frame. A simple linear interpolation as suggested above, contrary to the naive

(a) Start of movement

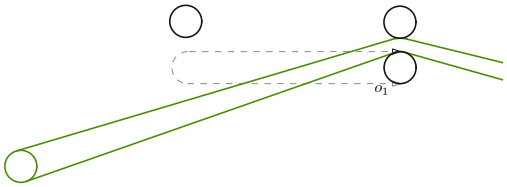
(b) End of movement

Figure 4.3: Example of how the transformation can be interpolated between a start and an end state



(a) Obstacle o_1 is squished at the start state

(b) During the movement the orbit radius of o_1 can expand to its full width



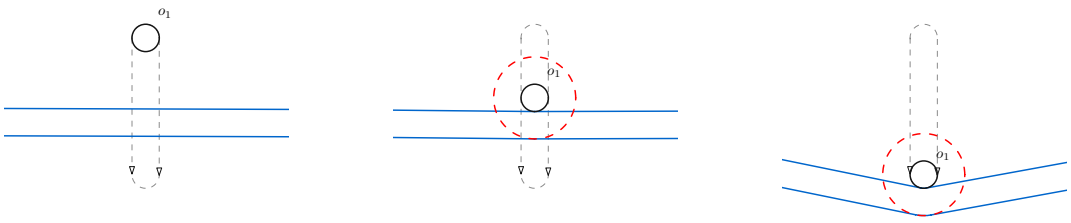
(c) Obstacle o_1 is again squished at the end state

Figure 4.4: Example instance in which the orbit transformation cannot be fully reproduced with just an interpolation between the start and end frame

approach detailed in Section 4.1, would not capture this transformation, as it would start with the size of the orbit radii in the start state and interpolate to the size of the orbit radii in the end state, omitting the intermediate expansion between the two.

Furthermore, solely interpolating the orbit radii between the start and the end state of a kinetic instance does not account for changes in the actual set routing that might need to occur throughout the movement. An example of such a movement can be seen in Figure 4.5. In this instance, the obstacles used by the blue set change midway during the movement (specifically at the time of Figure 4.5b). The proposed technique that only interpolates the orbit radii between the start and the end state of a kinetic instance would therefore also not be able to handle this instance correctly, as the obstacle o_1 would just pass right through the set, without adding it to the obstacle's orbits.

Simple interpolation. As shown in these examples, solely interpolating between two states, is not enough in most cases. Instead, a technique that can interpolate between a sequence of states, that represent significant changes and can therefore not be naively interpolated over, is needed. Ideally, we would find the exact points in time, for when



(a) Obstacle o_1 is not used by any set path

(b) During the movement of o_1 the blue set starts using o_1 and is added to the obstacle's orbits

(c) Obstacle o_1 is still used by the blue set at the end of the movement

Figure 4.5: Example instance in which a set is added to an obstacle's orbit during the movement

these states are reached, to interpolate between them. However, in practice and as explained in the following, we use a discretized way to find the frames, where these states are reached, the so-called *critical frames*. Using such a method, the question is how to find these critical frames and what properties a frame has to possess to qualify as a critical frame. We have identified the following types of critical frames:

- **Start and end.** The start and end frames of a kinetic instance are critical frames since they are used as a starting/end point.
- **Orbit radii tipping point.** The frame in which the growth direction of any obstacle's orbit radii reaches a tipping point (i.e. changes from expanding to shrinking or vice versa) is also a critical frame. This is because during the interpolation between two frames each obstacle's orbit radii can only grow in one direction (either expand or shrink). However, since we strive to capture the full transformation of the instance, as exemplarily shown in Figure 4.4, the frames that contain a tipping point for the growth direction of any obstacle's radii have to be critical frames as well.
- **Set addition/removal.** A frame in which the obstacles used by any set in the instance change also needs to be a critical frame. This is because the simple interpolation only changes obstacles' orbit radii during a movement, but is not capable of adding/removing sets from an obstacle's orbit. However, instances exist like the one shown in Figure 4.5, which necessitate the capability of adding/removing obstacles to a set path throughout the transformation.

- **Squishing obstacle change.** Obstacles reduce their orbit radii when they are squished with another obstacle. The frame at which an obstacle that is responsible for another obstacle's orbit radii reduction changes is a critical frame. This is because, during the actual movement, these obstacle orbit radii are linearly dependent on the distance between the two obstacles squishing each other.

While there may exist a more sophisticated method that is able to find these critical frames throughout the transformation of a kinetic instance, we decided to use a simple technique that initially computes every frame with the full static algorithm, like the naive approach described in Section 4.1. This allows the algorithm to identify the critical frames, which are then used to visualize the actual movement, by continuously interpolating between each pair of consecutive critical frames.

This technique of identifying critical frames and processing through a queue of them is not too dissimilar from other state-of-the-art techniques that deal with movement. Particularly kinetic data structures [Bas99] also compute a series of *events* (similar to our concept of critical frames) ahead of time. However, instead of the naive method we use, kinetic data structures use efficient methods to detect when such events happen.

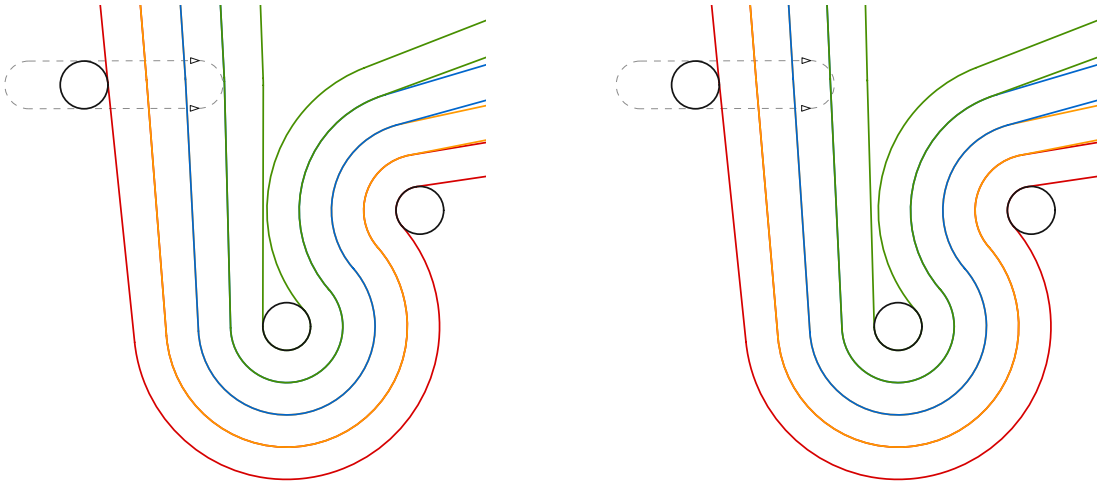
The naive method of recomputing every frame to identify the critical frames enables a straightforward way of testing the capabilities of the interpolation approach. The method of finding critical frames is simple and uses our previous algorithm. However, it can be replaced by a more efficient method (for example a kinetic data structure [Bas99]) to make the interpolation approach as a whole more efficient.

During the actual movement, the critical frame interpolation then either runs the whole static algorithm (at the time of a critical frame) or interpolates the obstacles' orbit radii at any given point in time between two critical frames. This *simple interpolation* technique produces an identical outcome to the stateless naive approach described in Section 4.1 and therefore also has the same problem with discrete jumps as shown in Figure 4.1.

However, since we already compute each frame with the static algorithm before starting the actual visualization, it is possible to alter the computed states before interpolating between them. This critical frame altering can be used to circumvent some discrete changes throughout the movement.

State-aware interpolation. The altering of critical frames can take a variety of different forms. These include changing the orbit size of an obstacle or altering the obstacles used by a set path in an existing critical frame. Furthermore, it is possible that, in order to create a smooth interpolation, entire critical frames are removed or added to the set of critical frames between which the visualization interpolates.

For example, the discrete jump that is shown in Figure 4.1 can be detected preemptively and avoided by altering the set of critical frames. Specifically for this example, this means that the critical frame shown in Figure 4.1b is replaced with a different frame



(a) State without adaption as it is created by the alternating hull algorithm

(b) State with adaption, such that the path widths do not change discretely

Figure 4.6: Example showing the impact of adapting the state in a critical frame to bypass a discrete change. The change is most visible when comparing the differences in space between the green paths in the middle, which stems from the larger orbits the moving obstacle in Figure 4.6a uses.

at a later stage in the movement (where the moving obstacle would hit the red path naturally without expanding the paths and creating a bottle shape). In general, if the pre-computing of the states finds a critical frame that is the cause of such a discrete change, it can be replaced with a frame of a later state.

In the example instance of Figure 4.1 the technique described above replaces the frame of Figure 4.1b with the frame shown in Figure 4.6a. Therefore, during the movement, the sets are added to the moving obstacle at a later stage, right at the time when the obstacle would touch the red set.

Upon close inspection of Figure 4.6a it is visible, that the static algorithm used to produce the visualization in that frame, still produces a discrete change. Although the sets are added to the obstacle's orbits at the right time, the width of the set paths at the obstacle's orbits is increased compared to the width of the sets before the obstacle touches. Since the algorithm is aware of this previous state, it can however adjust the orbit radii of the obstacle in the critical frame of Figure 4.6a such that the set paths stay the same width on first contact with the obstacle. This change, which is shown in Figure 4.6b, eliminates this discrete change of the kinetic instance entirely.

In contrast to the naive recomputation method, this final version of the critical frame interpolation approach consequently is not using a stateless algorithm anymore. The fact that the altering of critical frames depends on the state of other critical frames makes this implementation state-aware.

CHAPTER 5

Evaluation

This chapter describes the evaluation process as well as the results achieved by the various techniques presented in this thesis. To that end, we first and foremost formally define aesthetic criteria inspired by the goals we have for the visualization. In addition to that we define measurable metrics that aim to objectively evaluate the instances based on these criteria.

To exhaustively test the algorithms, we created 60 test instances for the static case and 10 additional instances that include movement. These instances are created semi-randomly (i.e. random placements within a given rule set) and contain various unique features that allow us to classify them into different categories. In addition to the actual evaluation of all these instances and a discussion of the achieved results, this chapter further provides a high-level overview of the generation process used to create the instances.

5.1 Aesthetic Criteria

Before describing the actual evaluation metrics it is essential to define the aesthetic goals we strive for with the visualization, as the specified metrics are constructed such that they quantify how successful the visualization is in trying to adhere to these objectives. The two main aesthetic criteria for this visualization are *thickness* and *uniformity*. Since we set out to create a visualization using fat edges, the achieved thickness of the paths is consequently an important aesthetic criterion on which the visualization can be judged. However, as explained below, the uniformity of the paths also plays a crucial role in the perception of a human observer.

The impact that width uniformity can have on a visualization is illustrated in Figure 5.1, where in both cases the same amount of space between two obstacles is occupied by set paths, but the distribution of orbit radii is changed. This could entice an observer to think that one side is more important than the other, which is not the case. Furthermore,

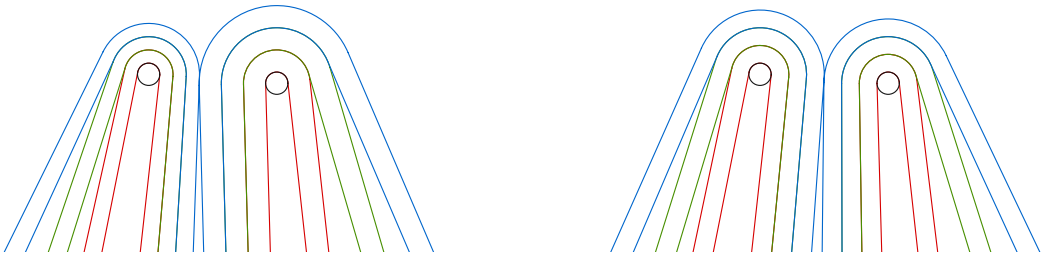


Figure 5.1: The orbit sizes in (a) and (b) add up to the same amount, however in (b) the orbit radii are evenly distributed

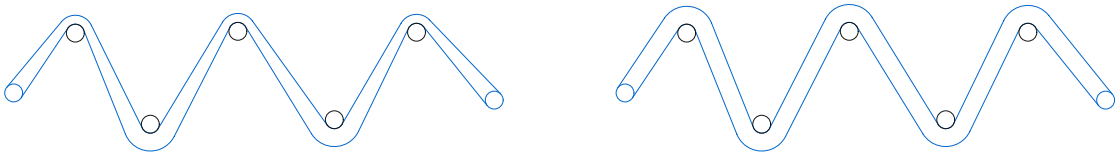


Figure 5.2: The set in (a) changes its width throughout its set path, while the set path in (b) is consistent in its width

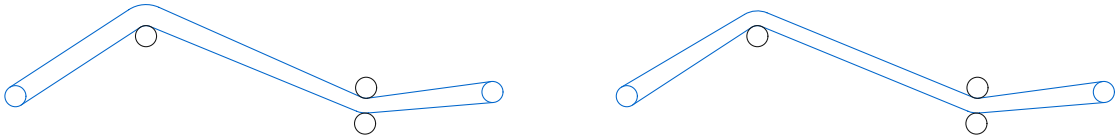
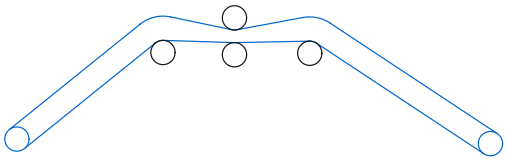


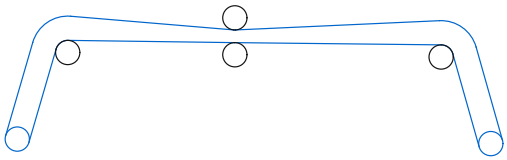
Figure 5.3: In (a) the algorithm tries to maximize the width of the set path at each used orbit, while (b) tries to keep its path width more uniform

if a set changes its width dramatically and frequently throughout its path, the perceived consistency of that edge changes for a human observer as visualized in Figure 5.2. In such a visualization one might infer some meaning from the width changes, where in actual fact there is none.

It is not always possible to satisfy both of these aesthetic criteria to their fullest extent, since sometimes they work against each other. An example where this is the case is shown in Figure 5.3, where Figure 5.3a tries to maximize the thickness of the drawn edge and Figure 5.3b focuses on the uniformity of the set path. In such cases, it is impossible to fully satisfy both goals, because of the restrictions imposed by the obstacles, creating the need for a compromise. However, in general, both of these aesthetic criteria are desirable for a static visualization and therefore important considerations when defining the evaluation metrics.



(a) Instance with a higher value of c and a higher value of w compared to the instance shown in (b)



(b) Instance with a lower value of c and a lower value of w compared to the instance shown in (a)

Figure 5.4: Example showcasing two similar instances, that contain the same narrow opening between the two obstacles in the middle, with different results for the average (weighted) path width w and average (weighted) change in width c

5.2 Static Evaluation Metrics

The static evaluation metrics used to assess the quality of a visualization are described below. These metrics aim to meaningfully and objectively judge the visualizations based on the aesthetic criteria described in Section 5.1.

In addition to formal definitions of an input instance given in Chapter 2, we define the following denotations to further define the computation of the evaluation metrics:

$E_s := (e_1^s, ..., e_n^s)$ ... tuple of elements (set items and obstacles) of the path of set s

l_s ... total length of set s

$l_{x,y}^s$ ... length of the path of set s between the elements at position x and y

w_t ... target width of the set paths

r_i^s ... radius size of element at position i in the set path of set s

$l := \sum_{s \in S} l_s$ ... total length of all set paths in an instance

Average (weighted) change in width c. The goal of this metric is to be an indicator of the uniformity of the set paths in an instance. To achieve this the metric measures the average change in thickness that occurs on all set paths from one orbit to the next. However, since the visual perception of such a transition is different depending on the distance between the orbits, as shown in Figure 5.4, the metric is weighted by these lengths (i.e. the shorter the distance between the orbits, the bigger the impact of the change).

The average (weighted) change in width of each set s is therefore computed as follows:

$$c_s = \sum_{i=1}^{|E_s|-1} (1 - \frac{l_{i,i+1}^s}{l_s})|r_i - r_{i+1}| \tag{5.1}$$

For an average over a whole instance, the values for each set c_s are then weighted again by the length of their sets, as shorter sets overall have less of an impact on the whole instance than longer ones do:

$$c = \sum_{s \in S} \frac{l_s}{l} c_s \tag{5.2}$$

Since one of the aesthetic goals is to keep the set paths as uniform as possible, it is desired that the value c is as small as possible, as this is indicative of smaller changes over bigger distances throughout an instance.

Average (weighted) path width w. This metric aims to be an indicator of how thick the set paths in an instance are. Path thickness is one of the aesthetic goals defined in Section 5.1. To compute the average path width of a set one can sum over all orbit radii of the elements used by the set path and then divide by their amount. However, similar to the average change in width, an orbit radius along a set path has a much bigger impact on the visual perception of the visualization if its neighbouring orbits on the set path are further away, as is visualized in Figure 5.4.

Therefore, for every part of the set path we compute the average width of the orbits between two elements and then weigh this width by the distance between them. This allows us to compute the average (weighted) path width for each set s like so:

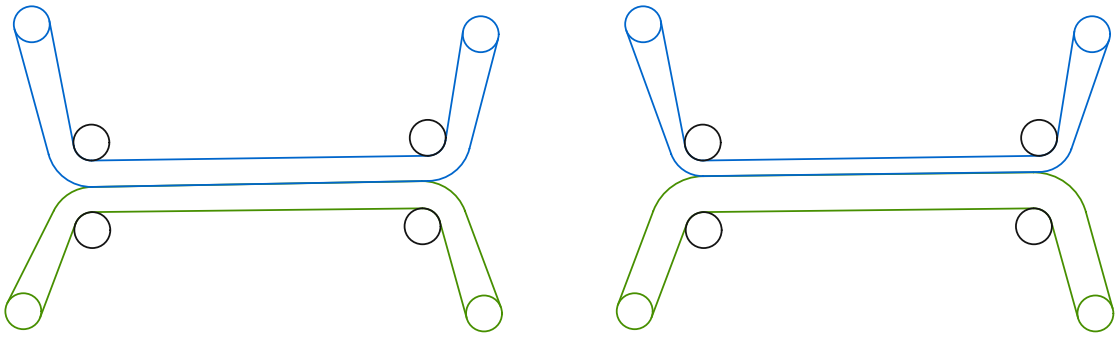
$$w_s = \sum_{i=1}^{|E_s|-1} \frac{l^s_{i,i+1}}{l_s} \frac{r^s_i + r^s_{i+1}}{2} \tag{5.3}$$

Finally, we compute the average (weighted) path width w of the whole instance, similar to the average (weighted) change in path width, by weighting the results by their set length like so:

$$w = \sum_{s \in S} \frac{l_s}{l} w_s \tag{5.4}$$

The bigger the average (weighted) path width w of an instance is, the thicker the set paths of that instance are for longer distances. w can be at best as high as w_t (target width of the set path), if not a single set path is squished at any point throughout the instance.

Standard deviation σ of average (weighted) path width. The average (weighted) path width w of an instance is a good indicator of how thick the set paths are on average and how well the space is used. However, it falls short to characterize how this thickness is distributed. As an example, in an instance with a width of $w = 10$ and two sets of the



(a) Instance with a low σ since all paths are roughly the same width

(b) Instance with a high σ since all paths have different widths

Figure 5.5: Example showcasing two similar instances, that have a similar average (weighted) path width w, but differ in their standard deviation σ of average (weighted) path width

same length, there are multiple ways on how this average (weighted) path width might be achieved. For w it makes no difference, if $w_1 = 5$ and $w_2 = 15$ or $w_1 = 10$ and $w_2 = 10$, however for the visual perception of a human observer this can be a big difference that is not captured by the average (weighted) path width alone, as is shown in Figure 5.5.

Therefore, to better capture the consistency in set path thickness throughout an instance, we compute the standard deviation σ of the average weighted path width:

$$\sigma = \sqrt{\frac{1}{|S|}\sum_{s \in S}(w_s - w)^2} \tag{5.5}$$

The smaller σ the more similar in thickness the set paths of an instance, leading to an overall more consistent visualization.

Number of turns t. A turn in a set path describes a point at which the growth of thickness of the path changes direction (either from getting smaller to becoming bigger or vice versa). The idea behind the computation of the metric t is, that fewer turns lead to more uniformity (see Figure 5.1).

A turn in a set path occurs if there is a chain of (at least three) consecutive elements

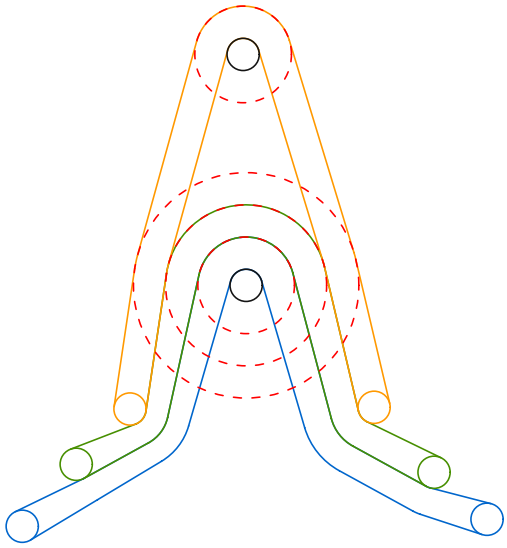


Figure 5.6: Example of an instance in which an orbit is used twice by the same set

$(x, y_1 ... y_c, z)$ on the set path with the following properties:

$$r_{y_1} = r_{y_2} = ... = r_{y_c}$$
$$\text{and}$$
$$r_x > r_{y_1}$$
$$r_z > r_{y_c}$$
$$\text{or}$$
$$r_x < r_{y_1}$$
$$r_z < r_{y_c}$$

The number of turns in a set t_s is a count of how many of these chains occur in a given set path. The number of turns t for the whole instance is then calculated as follows:

$$t = \sum_{s \in S} t_s \tag{5.6}$$

Number of orbits used m. Using more or fewer orbits is not strictly an indicator of a better or worse visualization judged by the aesthetic criteria defined in Section 5.1. Therefore it cannot be generalized if a higher or lower number of orbits used throughout an instance is desirable. Nevertheless, the different algorithms presented in Chapter 3 differ quite significantly in some cases in the number of orbits they use, which is why it can still be informative to observe the differences.

The elements in E_s for each set s can contain multiple instances of the same obstacle because a set path can use an obstacle's orbit more than once. An example of such an

instance can be seen in Figure 5.6. However, the number of orbits used for each set m_s only counts each orbit once and therefore is equal to the number of different elements in E_s minus 2 (not counting the set items, i.e. start and end).

The number of orbits used throughout an instance m is computed as follows:

$$m = \sum_{s \in S} m_s \tag{5.7}$$

Average difference u to orbit radius with optimal space usage. The aesthetic criteria defined in Section 5.1 can be summarized in that their goal is to use the space of an instance as efficiently as possible (i.e. make the set paths as thick as possible), while still trying to keep them somewhat uniform. Uniformity, among other things, means that the space between obstacles should be distributed evenly between set paths. This is precisely what the average difference u to orbit radius with optimal space usage tries to measure.

To achieve this we first aim to find the optimal orbit radii of any two obstacles (in terms of space usage of the set paths running between them). We then compare this value against the actually achieved orbit radii and average this difference over all of the evaluated obstacle pairs. Since not all algorithms use the same obstacles/orbits, this metric tries to find the largest subset of obstacles that all our algorithms have in common by using zero-width paths (similar as the expansion method described in Section 3.3.3). The obstacles used by the sets with this method are guaranteed to be used by every algorithm. For each obstacle that is used in a set path, and found with this method, we then find its associated, most restrictive, obstacle. How restrictive a pair of obstacles is to each other is measured as follows:

$$\frac{d_{o_x,o_y}}{p_{o_x,o_y} w_t}$$

Where d_{o_x,o_y} is the distance between the two obstacles o_x and o_y, w_t is the target width of the set paths and p_{o_x,o_y} is the number of times a set path crosses a direct line between the two obstacles o_x and o_y. The smaller this fraction is, the more restrictive the two obstacles o_x and o_y are to each other.

For each of the most restrictive obstacle pairs it is then possible to compute the theoretical optimal disk radius $b_{o_x} = b_{o_y}$, such that the space between the two obstacles o_x and o_y is used optimally, like so:

$$b_{o_x} = b_{o_y} = \min\left(\frac{d_{o_x,o_y}}{p_{o_x,o_y}}, w_t\right) \tag{5.8}$$

The difference to orbit radius with optimal space usage u_{o_x} for an obstacle o_x can then be calculated as follows:

$$u_{o_x} = |S_{o_x}| \cdot |b_{o_x} - a_{o_x}| \tag{5.9}$$

Where a_{o_x} is the actually achieved radius of the obstacle o_x and S_{o_x} is the set of sets running along the orbits of o_x.

Let O_r be the subset of obstacles of O which are found even with zero-width paths. Finally, the average difference to orbit radius with optimal space usage u, for a whole instance, can be computed as follows:

$$u = \frac{\sum_{o_x \in O_r} u_{o_x}}{|O_r|} \tag{5.10}$$

Since this value is indicative of how close the orbit radii of the instance are to a theoretical optimum, a lower value of u is desirable.

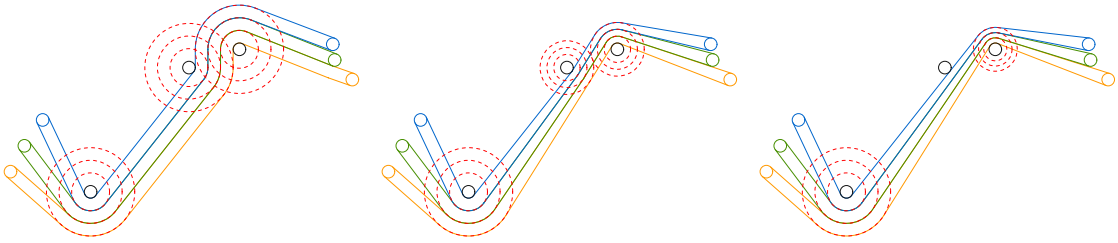
Maximal difference u_{max} to orbit radius with optimal space usage. In some instances, the average difference u to orbit radius with optimal space usage is low, despite there being some obstacle pairs between which the space usage of the orbit radii is far from optimal. This is because u only measures the average difference and a big amount of obstacles in O_r with a lower u_{o_x} lead to an overall smaller u, despite there being some obstacles with a big u_{o_x}. Therefore, in addition to the average, the maximal difference u_{max} to orbit radius with optimal space usage is an interesting metric as well, showcasing how far from the theoretical optimal an instance is at its worst.

5.3 Static Evaluation

To test our algorithms for static instances we created a set of 60 test instances on which we evaluate the different algorithms using the metrics defined in Section 5.2. These instances are created semi-randomly, meaning that the placement of obstacles and set items within each instance, as well as the number of sets and obstacles used, is random, but adheres to a set of rules. Depending on the set of rules used, the created instance falls into one of six different categories. The categorization of our test instances allows us to group the evaluation results, such that possible patterns that may emerge in instances with a specific feature can be better recognized.

An example of such a category of instances is the category drop, in which the instances all look similar to the instance shown in Figure 5.8. Each of the six categories covers ten instances. The rule set to create the instances of each category is described in the respective section of each category. Parameters that are used for all six categories are a target path width of 20, as well as a maximum amount of sets of 7 and a maximum amount of obstacles of 30. In our testing these bounds allowed for fairly complex instances that are still comprehensible for a human observer.

For every instance, we run three different algorithms and evaluate them on the previously defined metrics. The different algorithms are the alternating hull algorithm described in Setion 3.3.1, the same algorithm but only using disjoint orbits (as shown in Figure 5.7b) and the expansion method described in Section 3.3.3. An example comparison between



(a) Alternating hull algorithm **(b)** Alternating hull algorithm (with disjoint orbits) **(c)** Expansion method

Figure 5.7: Instance visualized with three different algorithms used in the static evaluation

the three different algorithms can be seen in Figure 5.7, where the alternating hull algorithm and the variant using disjoint orbits use the same set of orbits (although with different radii to keep them disjoint) and the expansion method uses only the orbits found with zero-width paths.

To visualize the achieved results of the different algorithms on the various instances and metrics we decided to use a visualization style similar to the one used by Nickel et al. [NSM+22]. In our case the instances are represented by columns, the evaluation metrics by rows and the different algorithmic approaches are represented by colour. An example of this representation can be seen in Figure 5.8, which visualizes the achieved results for the different instances of the category *Drop*. The goal of this way to represent the achieved results is to make it easily comprehensible for a human observer to not only understand the results but also allow for a more accessible way of finding patterns that may emerge.

The style used by Nickel et al. [NSM+22], adjusted to our needs, relies on the idea that every algorithm, on every instance, is represented as a bar from 0 (empty) to 1 (full), for every metric. Intuitively a full bar represents a better result for a human observer than an empty bar. Therefore, many of our evaluation metrics, described in Section 5.2 not only need to be normalized but also adapted, as for most of them a lower value is more desirable. Hence the following adaptions are made for the visualization of the achieved results:

- **Average (weighted) change in width c.** The achieved value c is subtracted from w_t and then normalized on a scale from 0 to w_t
- **Average (weighted) path width w.** No adjustment made, achieved value is normalized on a scale from 0 to w_t
- **Standard deviation of average (weighted) path width σ.** The achieved value σ is subtracted from $\frac{w_t}{2}$ and then normalized on a scale from 0 to $\frac{w_t}{2}$

- **Number of turns t.** The achieved value t is subtracted from the maximum number of orbits achieved per instance by any algorithm m_{max} and then normalized on a scale from 0 to m_{max}, as every orbit could potentially be used to make a turn
- **Number of orbits used m.** No adjustment made, achieved value is normalized on a scale from 0 to the maximum number of orbits achieved per instance by any algorithm m_{max}
- **Average difference to orbit radius with optimal space usage u.** The optimal average space usage u_{opt} for each instance is calculated and the achieved value u is subtracted from it, then the value is normalized on a scale from 0 to u_{opt}
- **Maximal difference to orbit radius with optimal space usage u_{max}.** The optimal space usage u_{opt} for the obstacles responsible for u_{max} is calculated and the achieved value u_{max} is subtracted from it, then the value is normalized on a scale from 0 to u_{opt}

In the following, the different test instance categories as well as the achieved results of the different algorithms are described.

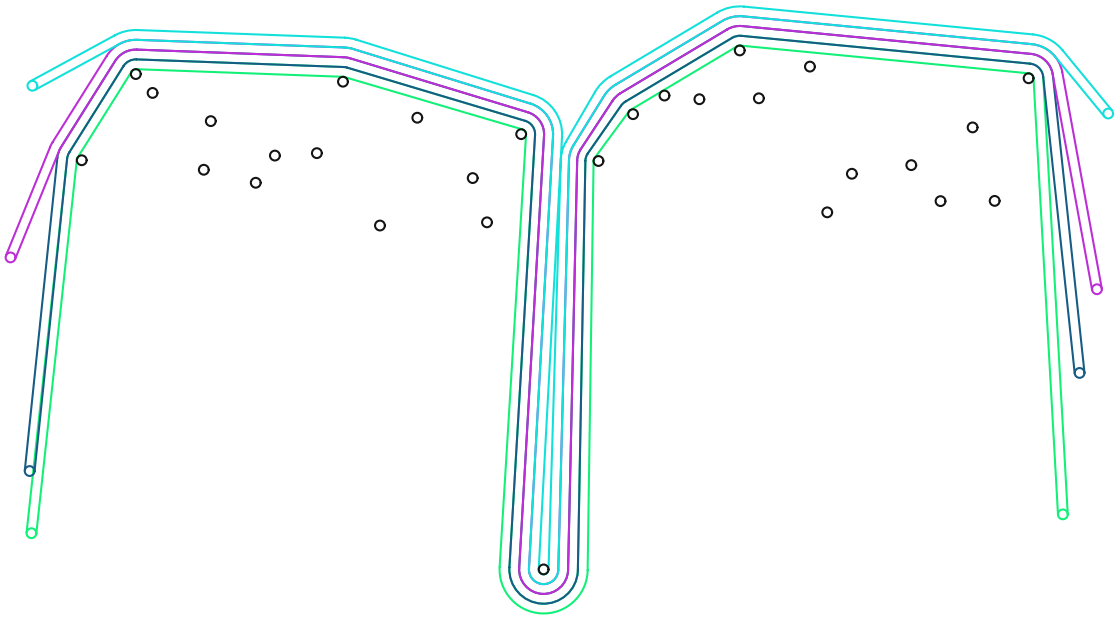


Figure 5.8: Exemplary instance *Drop_0003* for the category Drop, visualized with the alternating hull algorithm

5.3.1 Category: Drop

The instances in this category feature a random number of obstacles that all but one demand the same direction for every set in the instance. This single obstacle always is in the center of the instance. Furthermore, all obstacles demanding the same direction are in close proximity of the same y-coordinate, while the single obstacle with the inverted demand is further away. After routing the set paths through such an instance the visualization somewhat resembles a (water-) drop, as shown by the example instance in Figure 5.8. The direction of this drop is chosen at random.

As is visible in Figure 5.9, the alternating hull algorithm either wins or ties in most of the instances for most of the metrics. Overall the results of the three algorithms are fairly similar for this category, however, there are instances (e.g. *Drop_0007* and *Drop_0009*) in which the expansion method always lacks behind the alternating hull algorithm.

Looking at these instances, and in particular instance *Drop_0007* shown in Figure 5.10, we can observe why the expansion method performs worse. The naive approach used for the expansion method uses only obstacles found with zero-width paths, it is unable to add obstacles to a set path while expanding. Therefore, as shown in Figure 5.10b, the obstacles left and right of the drop cannot expand their orbit radii further, as the sets on their orbits would interfere with the bottom two sets of the instance.

Furthermore, it is visible from the results shown in Figure 5.9, that the usage of the alternating hull algorithm with disjoint orbits does not improve any of the metrics by a significant amount in comparison to the unaltered alternating hull algorithm. On the

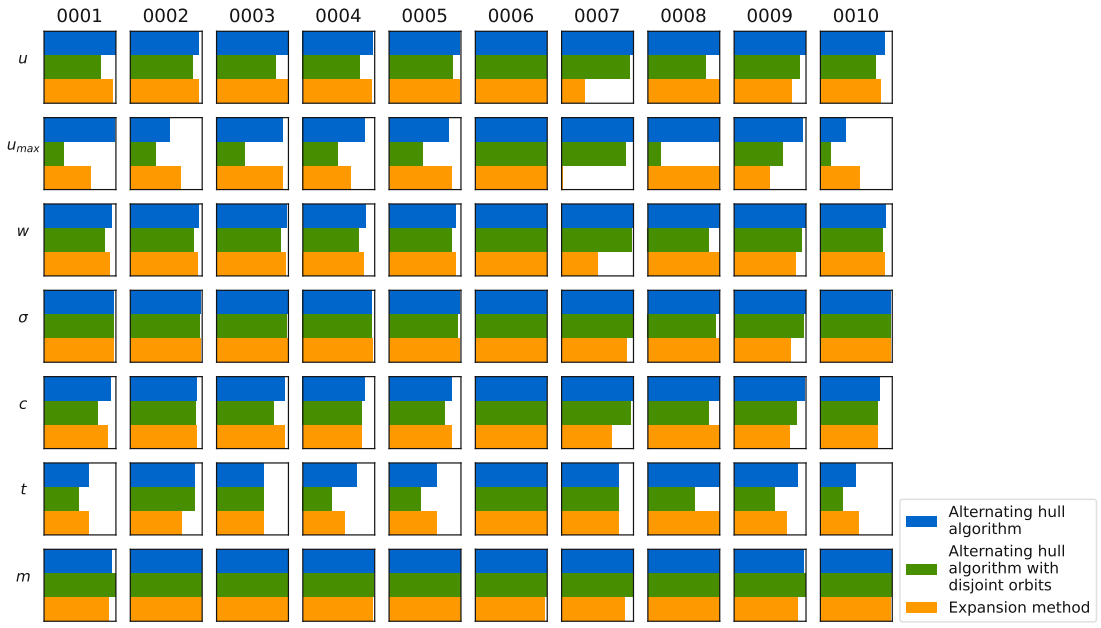
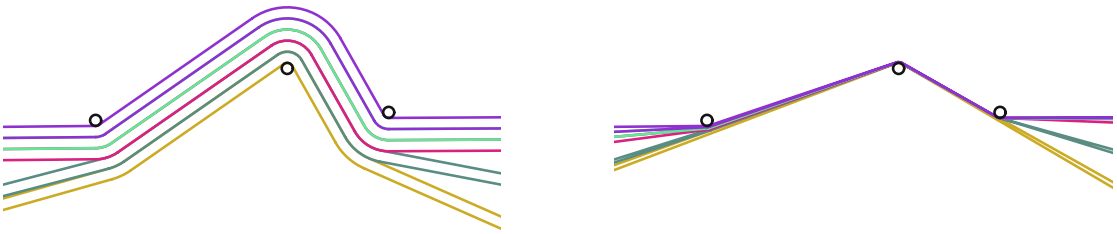


Figure 5.9: Evaluation results for the instances of the category Drop. Bigger bar means better result. Results of metrics u, u_{max}, σ, c and t are inverted and all results are normalized as described in Section 5.2



(a) Excerpt of instance *Drop_0007* visualized with the alternating hull algorithm

(b) Excerpt of instance *Drop_0007* visualized with the expansion method

Figure 5.10: Excerpt of instance *Drop_0007* visualized by two different algorithms

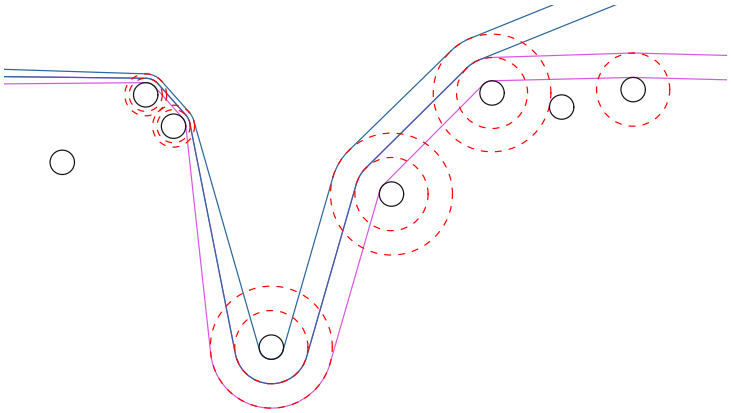


Figure 5.11: Excerpt of instance *Drop_0008* visualized with the alternating hull algorithm with disjoint orbits

contrary, most of the metrics suffer from using disjoint orbits. The reason for this is apparent when looking at the example cutout of an instance shown in Figure 5.11, where it is clearly visible that the orbits of the two leftmost obstacles could be a lot bigger when giving up the constraint of disjoint orbits. Not only would this change increase the overall average (weighted) path width of the instance, but in this case, it would also increase the uniformity of the paths in the instance, which is for example reflected in the worse value of c and u_{max} achieved by the default algorithm with disjoint orbits.

This behaviour of the alternating hull algorithm with disjoint orbits is, however, not strictly related to this specific category and can be observed in most of the other categories as well.

5.3.2 Category: Funnel

This category of instances uses a random number of obstacle pairs that create a funnel that decreases in size from left to right. The top obstacle of each pair thereby demands from all sets to be routed below and the bottom obstacle demands from all sets to be routed above. Routing the set paths through these obstacle pairs creates a visualization that looks like a funnel, as shown in the example Figure 5.12.

Looking at Figure 5.13 it is again visible, that the alternating hull algorithm either matches or outperforms the other two algorithms. A speciality of this category is that none of the algorithms performed particularly well on the metric of average (weighted) path width w. This, however, can be attributed to the metric w and not the algorithms. Routing the set paths through a small funnel only leaves so much space that an algorithm can utilize. Hence, even an optimal visualization of this instance has to include some amount of squishing. The metric w, however, punishes squishing regardless if it happens in an unavoidable situation or not.

One interesting observation can be made when comparing the results of the alternating

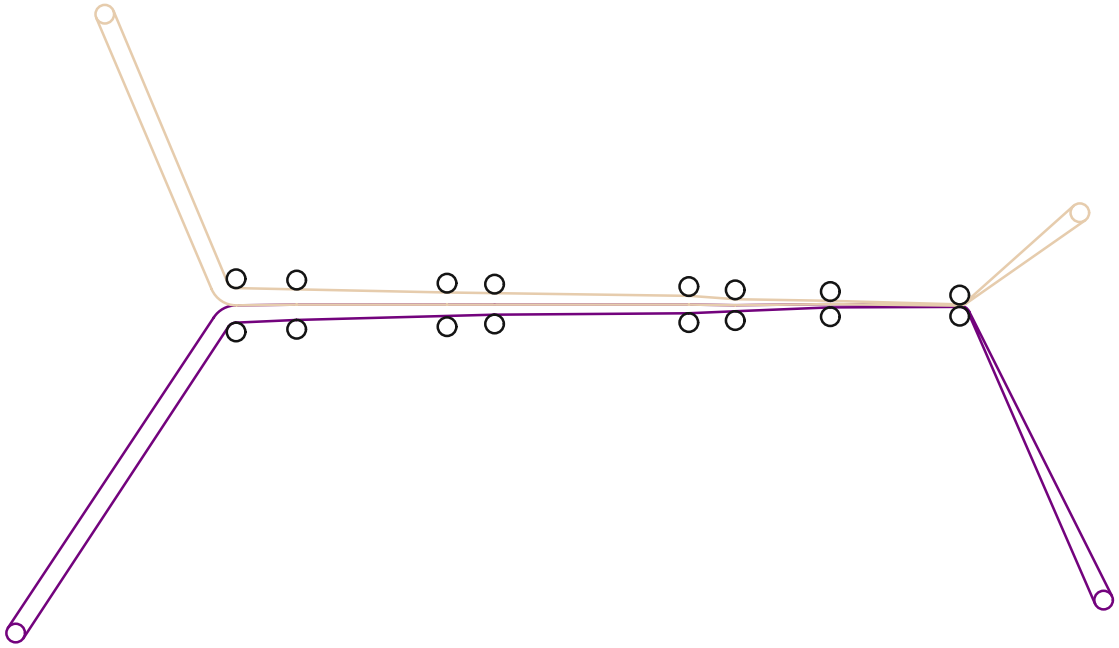


Figure 5.12: Exemplary instance *Funnel_0003* for the category Funnel, visualized with the alternating hull algorithm

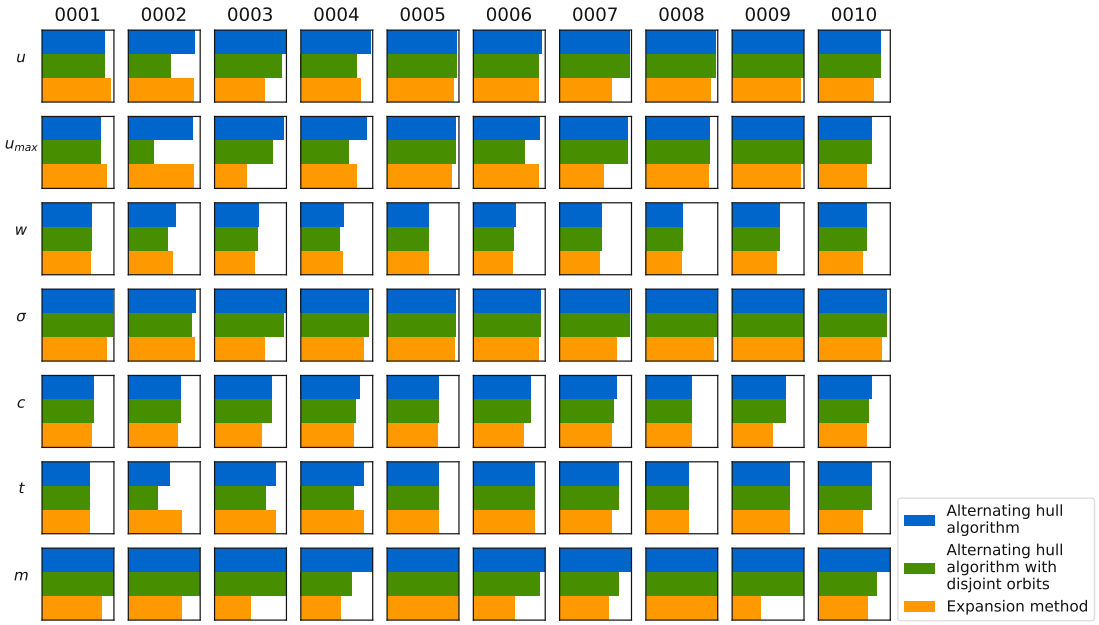
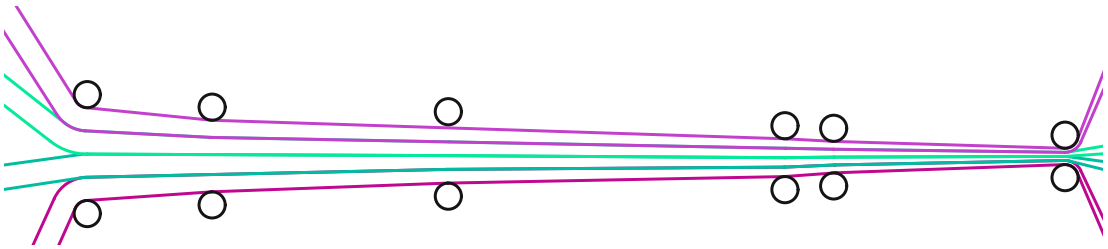
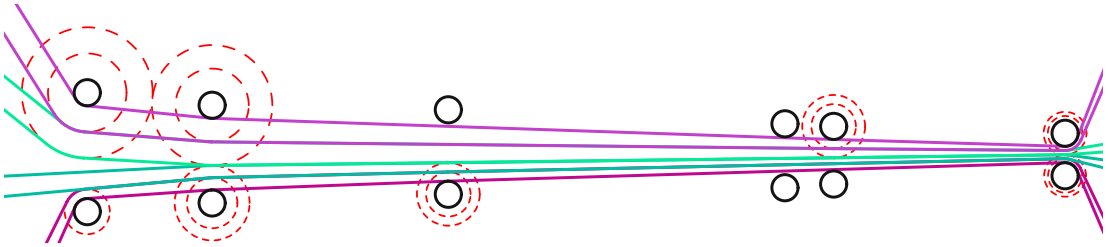


Figure 5.13: Evaluation results for the instances of the category Funnel. Bigger bar means better result. Results of metrics u, u_{max}, σ, c and t are inverted and all results are normalized as described in Section 5.2



(a) Excerpt of instance *Funnel_0007* visualized with the alternating hull algorithm



(b) Excerpt of instance *Funnel_0007* visualized with the expansion method

Figure 5.14: Excerpt of instance *Funnel_0007* visualized by two different algorithms

hull algorithm and the expansion method on the instances *Funnel_0003* and *Funnel_0007*. Here it is visible, that despite them having an almost identical average (weighted) path width w, the expansion method falls behind in the standard deviation of the average (weighted) path width σ. The reason for this can be observed in Figure 5.14, where it is visible that between most of the obstacle pairs the same amount of space is occupied by set paths in both variants, however, the distribution between the sets varies a lot.

The reason for the imbalance of weight distribution with the expansion method, shown in Figure 5.14b, stems from the orbits this naive approach chooses for the zero-width paths that are expanded. These orbits are also visualized in Figure 5.14b. Since the two topmost sets of the instance are routed along the orbits of the second to last orbit pair to the right they can be expanded further at this point, thereby limiting the growth of the two sets at the bottom throughout the rest of the instance.

5.3.3 Category: Long Shot

This category of randomly generated instances has similarities with the category Drop in that it features randomly placed obstacles that all have the same demand from all sets except one, which inverses its set demands. However, as seen in the example Figure 5.15, and in contrast to the category Drop, the single obstacle with these inverted set demands is to the far right of the instance and not in the middle.

Looking at the results of this category, shown in Figure 5.15, it is visible that the alternating hull algorithm, as well as its variant using only disjoint orbits, both consistently outperform the expansion method by huge margins compared to the other categories. The reason for this is similar to the problem observed in the Drop category and stems

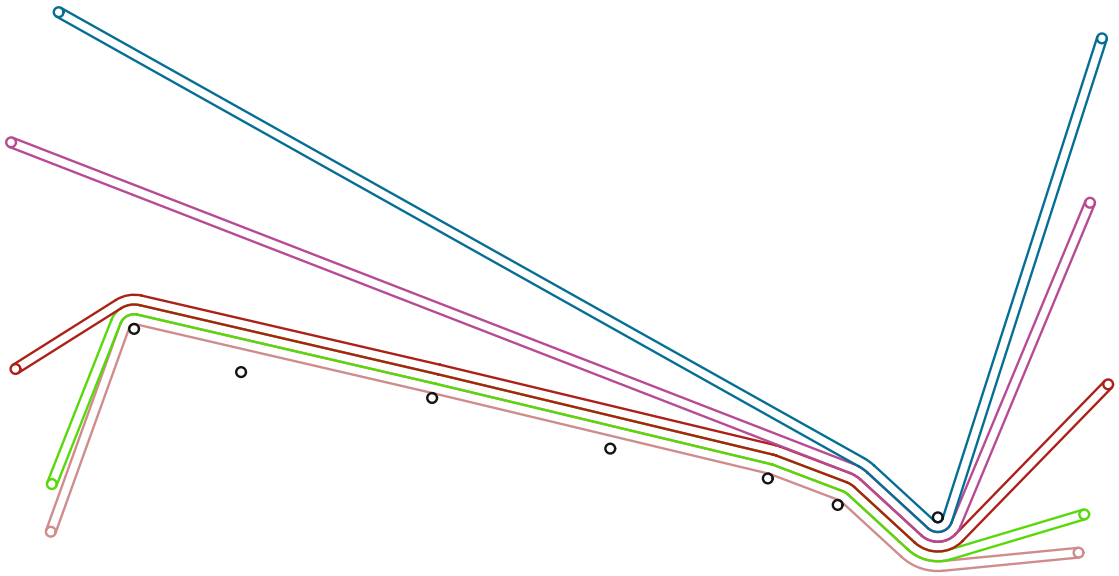


Figure 5.15: Exemplary instance *LongShot_0001* for the category Long Shot, visualized with the alternating hull algorithm

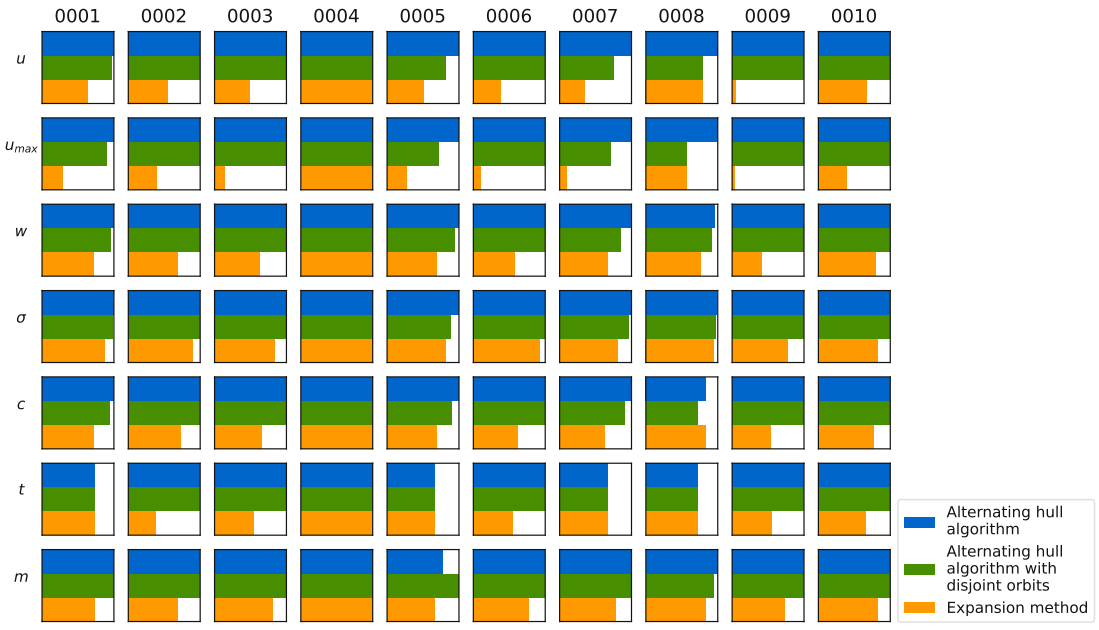


Figure 5.16: Evaluation results for the instances of the category Long Shot. Bigger bar means better result. Results of metrics u, u_{max}, σ, c and t are inverted and all results are normalized as described in Section 5.2

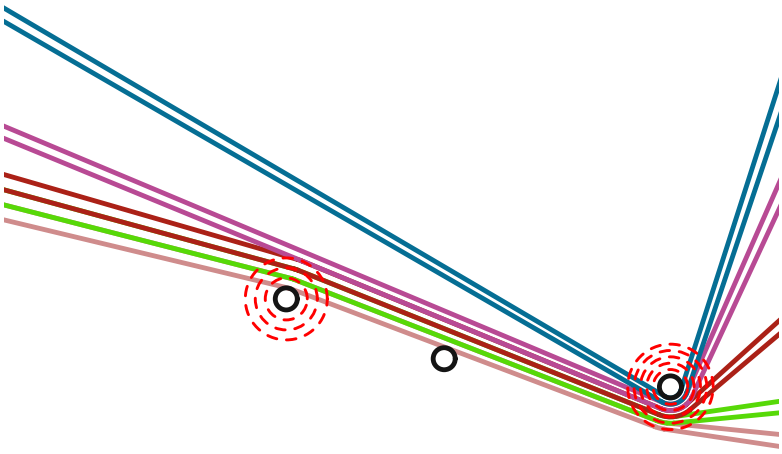


Figure 5.17: Excerpt of instance *LongShot_0001* visualized with the expansion method

from the expansion method's inability to add additional obstacles to set paths after it computed the initial set of used orbits with zero-width paths. This is visualized in Figure 5.17, where the purple set cannot be added to the orbits of the leftmost obstacle, therefore hindering the ability of the obstacle to grow its orbit radii any further without creating intersects between the set paths.

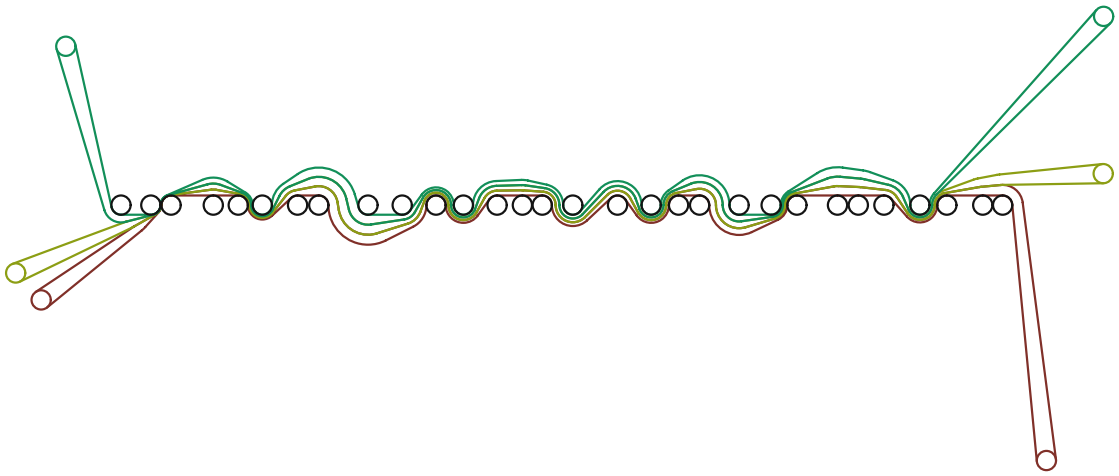


Figure 5.18: Exemplary instance *Snake_0003* for the category Snake, visualized with the alternating hull algorithm

5.3.4 Category: Snake

This category places all obstacles on a horizontal line in the middle of the instance. The distances of the obstacles to each other as well as their set demands are random, although if an obstacle demands from a set to be routed above, it is assumed that this demand holds for all other sets as well. Instances created with these specifications look like the example shown in Figure 5.18 and resemble a snake.

Interestingly for this category, the results of the expansion method and the alternating hull algorithm, shown in Figure 5.19, are a lot closer to each other with the alternating hull algorithm with disjoint orbits losing to both in almost every instance and metric except the average (weighted) change in path width c. The similarity between the results of the alternating hull algorithm and the expansion method is explained by the routing of zero-width paths.

This routing led to worse performance for the expansion method in other categories and especially in the category Long Shot described in Section 5.3.3, as it differed in crucial areas from the routing used by the alternating hull algorithm. However, in this category, the zero-width routing already includes most of the important orbits and crucially (by the design of the category) most of the obstacles either are not used at all or are used by all sets anyways. Therefore the expansion method can expand each obstacle without running into the issue of intercepting another set that does not use the obstacle.

The poor performance of the alternating hull algorithm with disjoint orbits stems from the same reason as already discussed in Section 5.3.1, however, this category of instances is especially prone to it since the obstacles are so close to each other. Therefore this algorithm limits the size of the orbits quite drastically compared to the other two algorithms, as exemplarily shown in Figure 5.20. The one advantage of this algorithm is, that since the width of the set paths is so limited, their ability to change in width is also

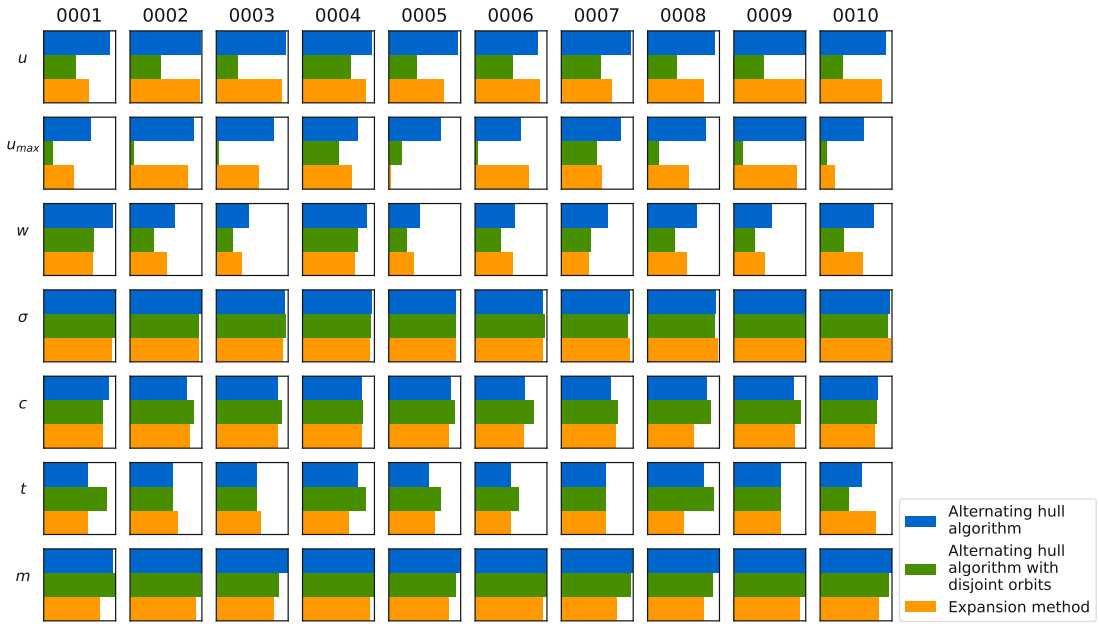


Figure 5.19: Evaluation results for the instances of the category Snake. Bigger bar means better result. Results of metrics u, u_{max}, σ, c and t are inverted and all results are normalized as described in Section 5.2

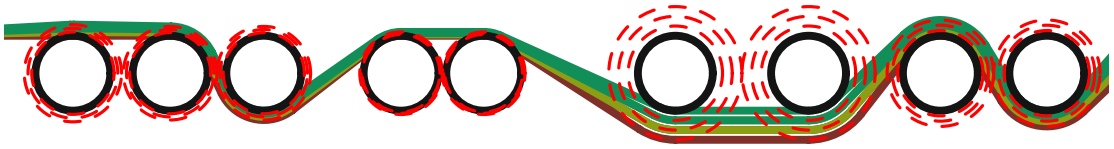


Figure 5.20: Excerpt of instance *Snake_0003* visualized with the alternating hull algorithm with disjoint orbits

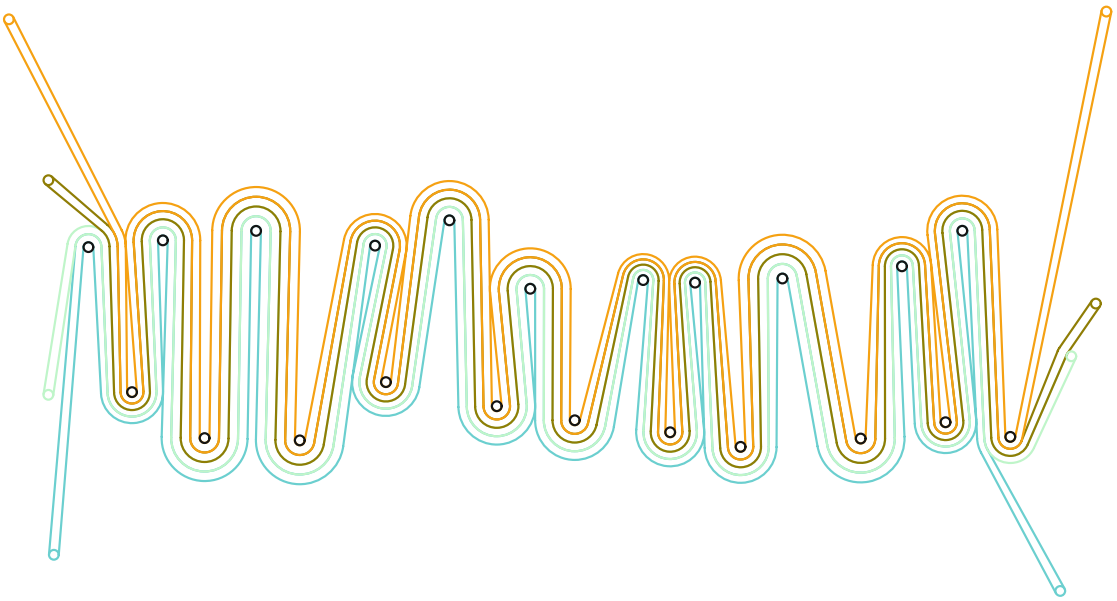


Figure 5.21: Exemplary instance *UpDown_0006* for the category Up Down, visualized with the alternating hull algorithm

limited, therefore leading to a better performance in the average (weighted) change in width of the set paths c.

5.3.5 Category: Up Down

For this category, the obstacles are randomly placed at roughly the same distance to a horizontal line in the middle of the instance, but alternating between being above the horizontal line while demanding from the sets to be routed above, and being below the horizontal line while demanding from the sets to be routed below. The set paths, therefore, traverse up and down between the obstacles, creating a visualization that looks like the one in Figure 5.21.

The evaluation results of this category, presented in Figure 5.22, show that all of the algorithms are comparable for this category. For the expansion method, this can be explained with the zero-width paths again already containing the same routing as used by the other algorithms, like for the category Snake in Section 5.3.4. The alternating hull algorithm with disjoint orbits is also able to produce comparable results to the unaltered alternating hull algorithm in this category, since in most cases (by the design of the category) the orbits of the obstacles are disjoint anyways.

One interesting exception to the overall very similar results in this category can be seen in the metric of u_{max}, particularly in instance *UpDown_0002* and *UpDown_0008*, where the alternating hull algorithm as well the version with disjoint orbits lack behind the expansion method. The reason for this can be observed in Figure 5.23. Specifically,

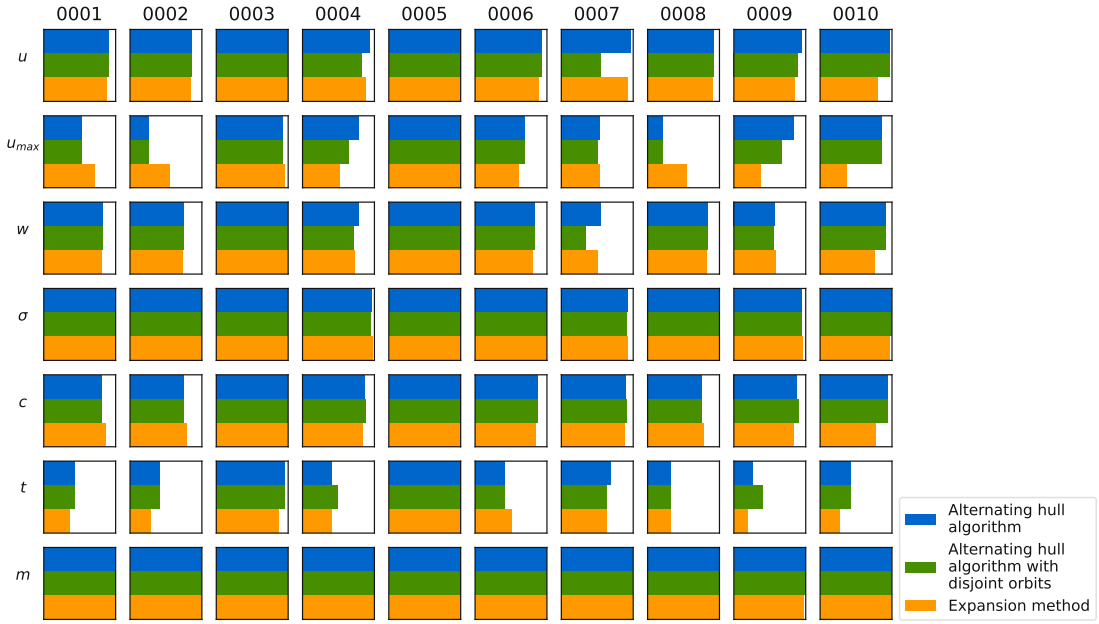
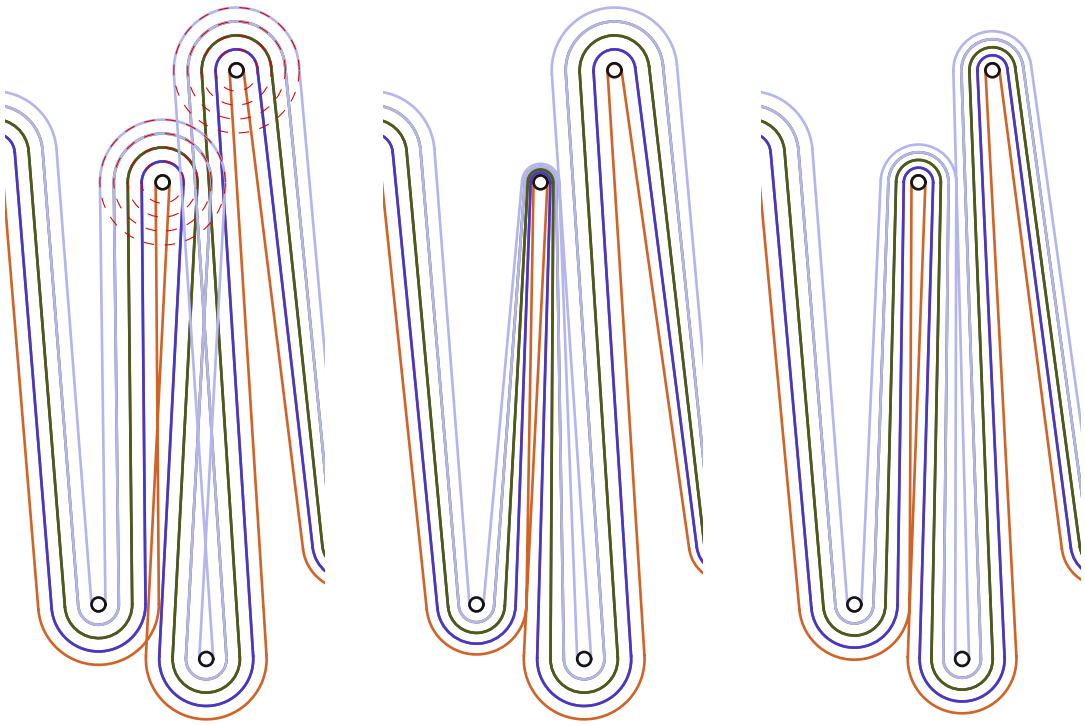


Figure 5.22: Evaluation results for the instances of the category Up Down. Bigger bar means better result. Results of metrics u, u_{max}, σ, c and t are inverted and all results are normalized as described in Section 5.2

Figure 5.23a, which displays the visualization using the alternating hull algorithm before the post-processing, shows that the base algorithm itself is not capable of drawing the set paths without intersection, even though the orbits of the two topmost obstacles are disjoint. Therefore, the post-processing applies inverse pushing as explained in Section 3.3.1 and produces the visualization seen in Figure 5.23b.

In comparison to that the expansion method increases the orbit radii of the two topmost obstacles evenly and therefore creates a more uniformly distributed visualization, as seen in Figure 5.23c. This deviation from the optimal distribution using the alternating hull algorithm is the cause for the worse performance in the metric u_{max}, however, such cases are sparse which is why it is not reflected in the metric u.



(a) Alternating hull algorithm (before post-processing)

(b) Alternating hull algorithm (after post-processing)

(c) Expansion method

Figure 5.23: Excerpt of instance *UpDown_0008* visualized with the alternating hull algorithm and the expansion method

5.3.6 Category: Random

The final category uses entirely random obstacles throughout its instances. Each obstacle can demand different directions from different sets (however, unavoidable crossings between the sets remain prohibited) and the only spatial restriction is that all the obstacles are in between the set items, which are roughly at the left and the right border of each instance. An example of how such an entirely randomized instance can look can be seen in Figure 5.24.

Because of the random nature of the instances evaluated in this category, there is also no clear pattern visible in the evaluation visualization shown in Figure 5.25. Most importantly though it can be said that for most instances and most metrics, the alternating hull algorithm either outperforms or is on par with the other two algorithms.

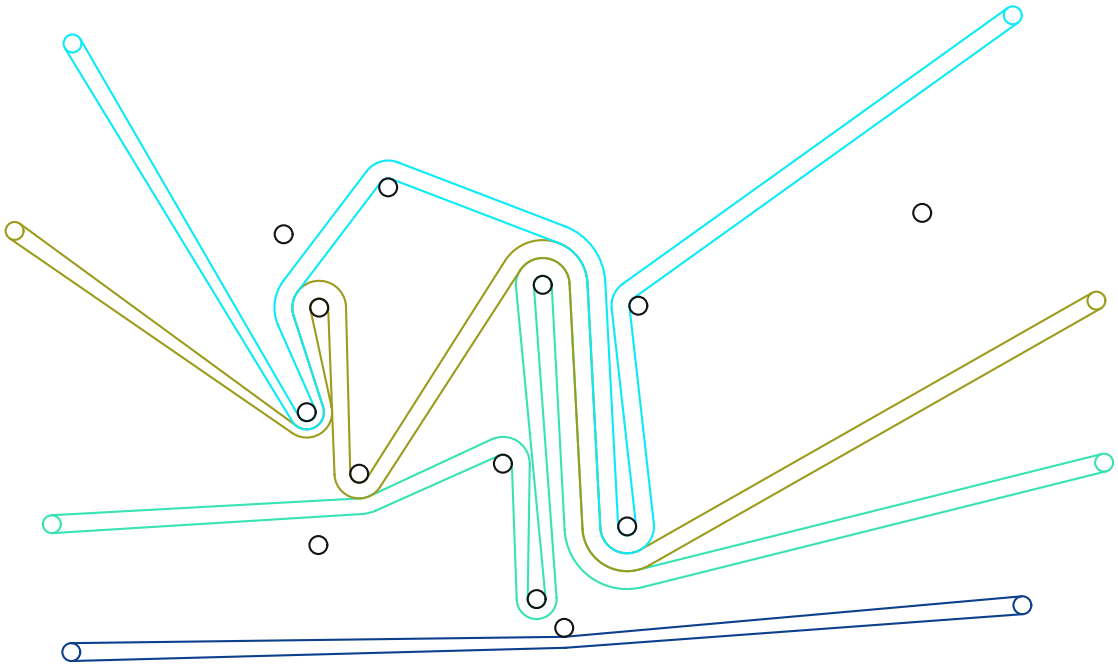


Figure 5.24: Exemplary instance *Random_0010* for the category Random, visualized with the alternating hull algorithm

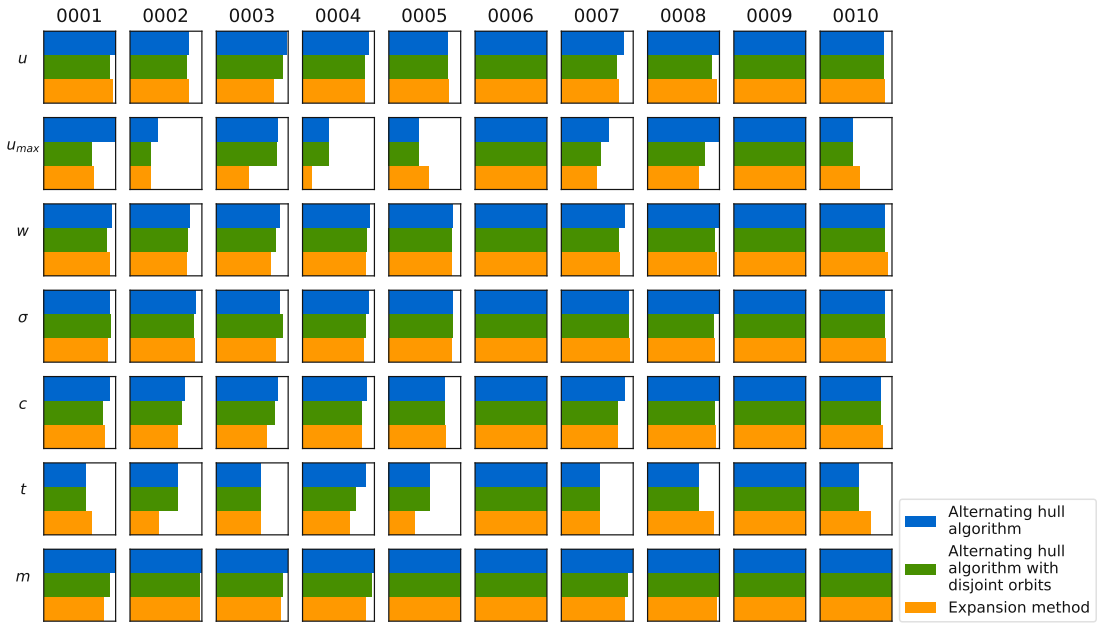
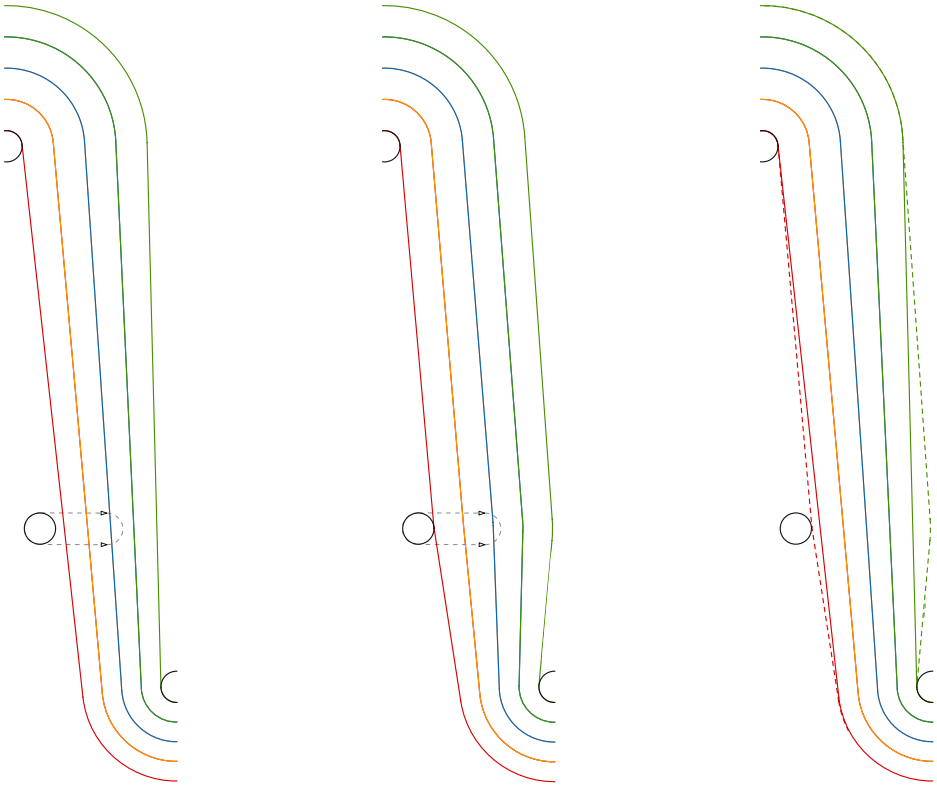


Figure 5.25: Evaluation results for the instances of the category Random. Bigger bar means better result. Results of metrics u, u_{max}, σ, c and t are inverted and all results are normalized as described in Section 5.2



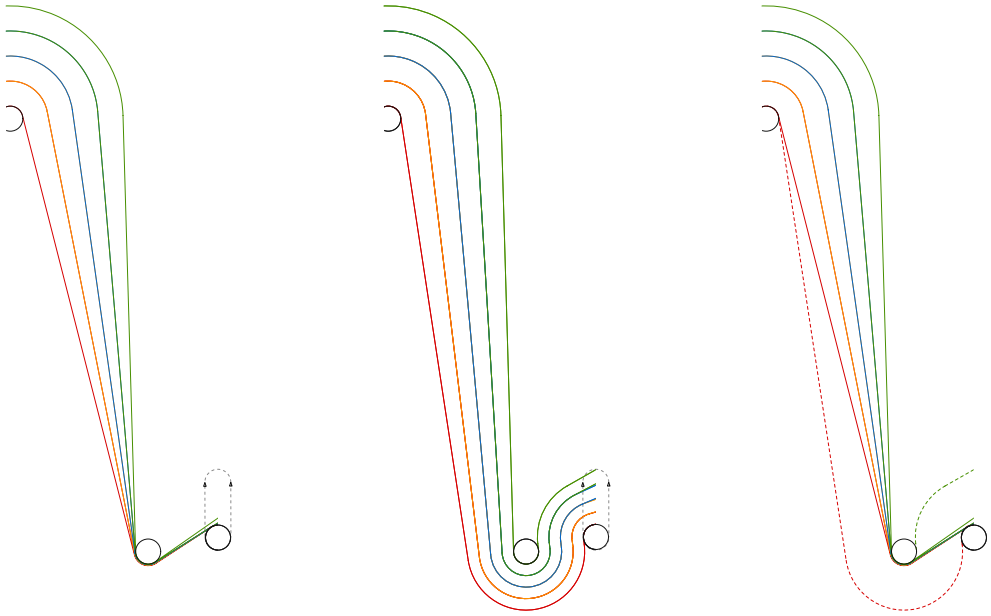
(a) Frame right before the jump **(b)** Frame right after the jump **(c)** The affected area of the discrete change is calculated by measuring the difference between the dashed and the solid red and green lines

Figure 5.26: Instance showcasing a discrete jump of type 1

5.4 Kinetic Evaluation Metrics

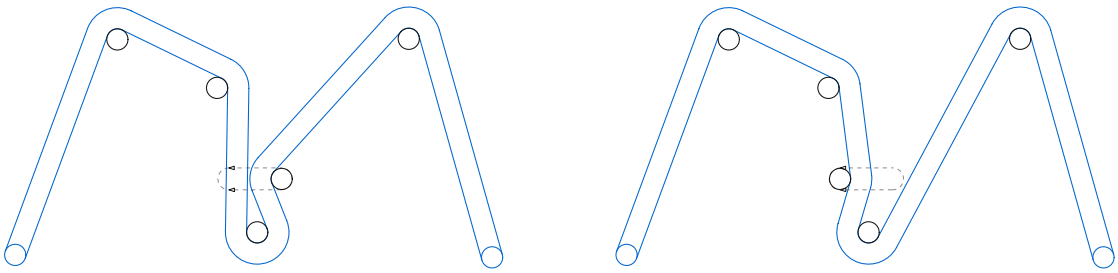
The kinetic evaluation metrics used to assess the quality of a movement visualization are described below. In contrast to the static evaluation metrics described in Section 5.2, the metrics used in kinetic instances do not focus on the uniformity or thickness of the set paths of a given instance. Instead, they aim to meaningfully and objectively judge a visualization based on the perceived smoothness of a movement. Therefore the chosen evaluation metrics are centred around the issue of discrete jumps that can occur during a visualization and their impact.

There are three types of discrete changes that can occur, however, only two of them are relevant for the evaluation as the third type cannot be realistically avoided with any of the algorithms presented in this thesis and therefore is also by design not present in any of the test instances. The three possible reasons/types for a discrete jump are the following:



(a) Frame right before the jump **(b)** Frame right after the jump **(c)** The affected area of the discrete change is calculated by measuring the difference between the dashed and the solid red and green lines

Figure 5.27: Instance showcasing a discrete jump of type 2



(a) Starting frame of the movement **(b)** End frame of the movement

Figure 5.28: Instance showcasing a discrete jump of type 3

1. **Jump because an obstacle is added/removed from a set path.** These jumps can occur because the alternating hull algorithm tries to maximize the width of the set paths without considering if an obstacle in a kinetic environment already touches the set paths (or should already be removed from them). This can be the cause for set paths to form a bottle shape (as seen in Figure 5.26b), which is responsible for this type of jumps. The origin of these discrete changes is explained in more detail in Section 4.1 and an example of such a change can be seen in Figure 5.26.

2. **Jump because an obstacle changes its orbit radii abruptly.** This type of discrete change happens if an obstacle changes its orbit radii abruptly from one frame to the next during the visualization. One possible explanation for why this type of jump occurs is that the zero-width path routing can change from one frame to the next. The consequence of this, at least for the expansion method, is that the space in which the orbit radii can expand changes significantly from one frame to the next, leading to this type of jump. The reason for this is explained in more detail in Section 4.1 and an example for it can be seen in Figure 5.27.

3. **Jump because an obstacle moves in such a way that the x-monotone path cannot be adapted without a discrete change.** This type of discrete change cannot be avoided by any of the algorithms presented in this thesis, as our assumption is that we deal with x-monotone paths. Therefore, the test instances do not feature any discrete change of this type and they are not of consideration for any of the remaining evaluations. An example of an obstacle movement that causes such a discrete change can be seen in Figure 5.28.

By design of our movement implementation with a fixed time and frames per second (fps) (see Chapter 4) the size of the jumps that occur throughout a movement is also dependent on the chosen time and fps. Therefore we decided to introduce a threshold value (in our case 1), which is the minimum size a jump needs to make in order to be counted as such. For discrete changes of type 1 this value is computed as the absolute difference between the sum of the widths of the set paths before and after the change at the position where the obstacle is added/removed. For discrete changes of type 2 this value is computed as the absolute difference between the sum of the orbit radii before and after the change.

To objectively judge the impact of the discrete changes we not only count the number of jumps that occur throughout the movement but also measure the area affected by the changes. This area is calculated differently for the two types of discrete changes and computed as follows:

1. To measure the area a type 1 jump affects we first measure the area of the set paths between the two obstacles that enclose the area where an obstacle is added/removed from a set path. This area is again measured after the jump and the absolute

difference between the two areas is equal to the affected area by this change, as is visualized in Figure 5.26c.

2. To measure the area a type 2 jump affects we first measure the area of the set paths routed along the affected obstacle, between the affected obstacle and its neighbouring obstacles. Then, after the discrete change, this area is measured again and the absolute difference between the two areas is equal to the affected area by this change, as is visualized in Figure 5.27c.

Finally, with the information on how to recognize discrete jumps and measure the area they affect, we define the evaluation metrics for kinetic instances as follows:

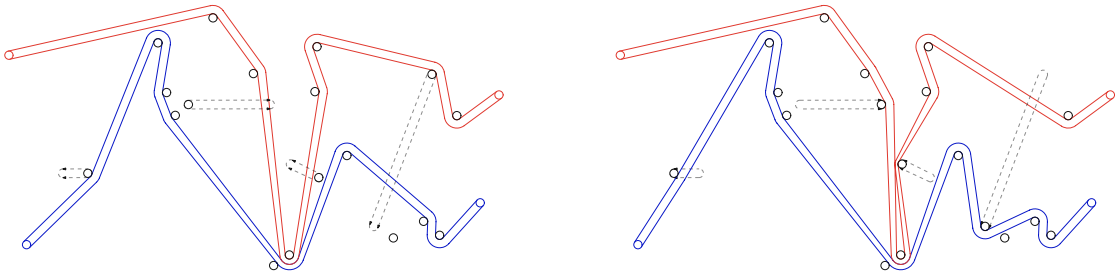
Number of discrete changes j. The amount of discrete changes that happen throughout the movement is one indicator of the smoothness of the visualization. j_1 counts the discrete changes of type 1 and j_2 counts the number of discrete changes of type 2 throughout a visualization. The evaluation metric j for an instance is equal to the sum of j_1 and j_2.

Average area of discrete changes $\mathbf{a_{avg}}$. This metric tries to quantify the impact of the discrete changes that happen throughout the movement. To achieve this it averages the sum of every affected area over the number of discrete changes j. A description of how the affected areas for the different types of jumps are computed is given above. The lower the average area of discrete changes a_{avg} is, the less visual impact the jumps have in the visualization.

Maximum area of discrete changes $\mathbf{a_{max}}$. The average area of discrete changes a_{avg} together with the number of discrete changes j already give a good indication of many discrete changes happen throughout a movement as well as their average impact. However, they do not convey any information about the biggest discrete change. Therefore it can also be of interest to look at the worst-case discrete jump in an instance and measure how much area it affects since one huge change might even be more disturbing to a human observer than many smaller ones. This is measured with the maximum area of discrete changes a_{max}.

5.5 Kinetic Evaluation

For the evaluation itself, we again created ten random instances, similar to the ones described in Section 5.3.6, but with added random movement for a small portion of the obstacles. The maximum number of obstacles and sets is again set to 30 and 7 respectively and the target set path width is set to 20. Furthermore, the moving obstacles of each instance are only allowed to move linear and discrete jumps of type 3 (explained in Section 5.4) are prohibited. An example of such an instance can be seen in Figure 5.29.



(a) Starting frame of the movement **(b)** End frame of the movement

Figure 5.29: Exemplary instance *Move_0001*, visualized with the alternating hull algorithm

Finally, the target frames per second are set to 30 and the time in which the movement has to complete is set to 10 seconds. These movement-specific parameters are important to recreate the achieved results as they can influence the metrics defined in Section 5.4 (with a low enough frame rate, every frame in which an obstacle is added/removed to a set path becomes a discrete change of type 1 as the previously defined threshold is trivially exceeded).

Each movement of the ten randomly generated instances is then tested for the defined metrics on three different movement visualizations. The first two tested approaches are the two different recomputation method implementations discussed in Section 4.1, where either the alternating hull algorithm or the expansion method is recomputed for every frame. The final algorithm tested is the state-aware interpolation approach (with pre-computed critical frames) discussed in Section 4.2.

To visualize the achieved results of the different implementations we again use the style used by Nickel et al. [NSM+22]. Similar to the static case, the instances are represented by columns, the evaluation metrics by rows and the different movement implementations are represented by colour in this visualization. Since all of the kinetic evaluation metrics indicate a smoother visualization if their value is low, we again have to adapt the achieved results for this visualization. The adaptions made are the following:

- **Number of discrete changes j.** The achieved value j for each instance is subtracted from the highest achieved j over all instances and then normalized on a scale from 0 to the highest achieved j over all instances
- **Average area of discrete changes a_{avg}.** The achieved value a_{avg} for each instance is subtracted from the highest achieved a_{avg} over all instances and then normalized on a scale from 0 to the highest achieved a_{avg} over all instances
- **Maximum area of discrete changes a_{max}.** The achieved value a_{max} for each instance is subtracted from the highest achieved a_{max} over all instances and then normalized on a scale from 0 to the highest achieved a_{max} over all instances

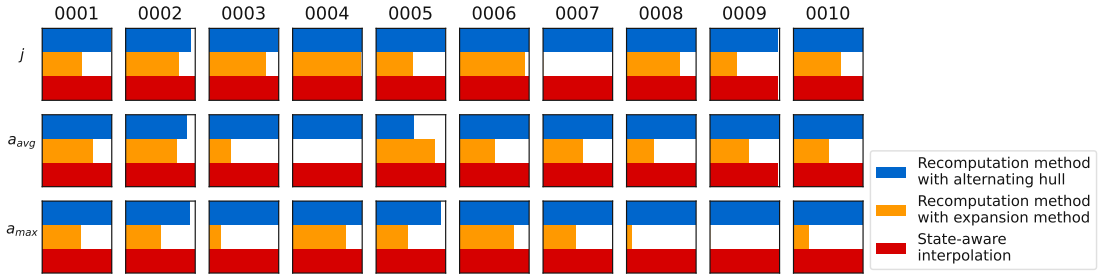
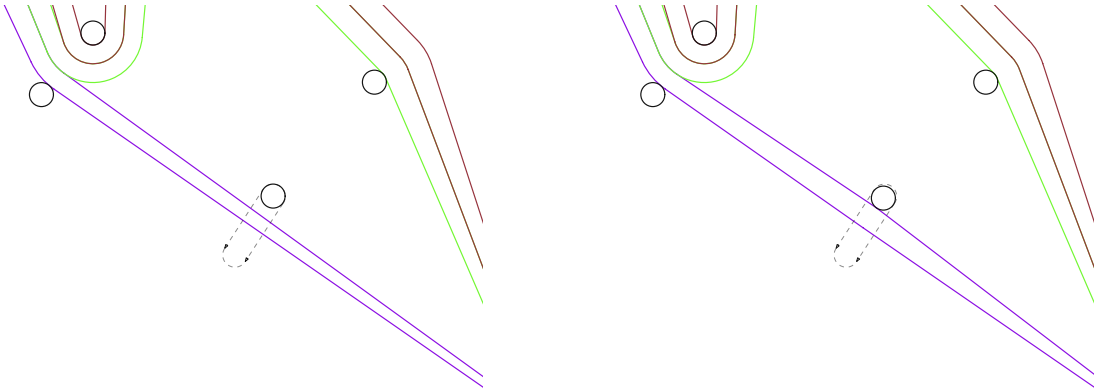


Figure 5.30: Evaluation results for the instances containing obstacle movement. Bigger bar means better result. Results of all metrics are inverted and normalized as described in Section 5.4



(a) Frame right before the discrete change **(b)** Frame right after the discrete change

Figure 5.31: Excerpt of instance *Move_0005*, visualized with recomputation method using the alternating hull algorithm

From the results shown in Figure 5.30 it is visible that the naive movement approach using the expansion method clearly performs the worst of the three. The reason for this is that this approach creates a huge amount of type 2 jumps compared to the other two approaches, as the zero-width path routing changes frequently from one frame to the next, creating big differences in the possibilities for the obstacles to expand their orbit radii. The origin of these type 2 jumps using the expansion method are explained in detail in Section 4.1.

Furthermore, we can see that the state-aware interpolation either improved or tied its results with the naive approach using the alternating hull algorithm, leading to an overall smoother movement.

Finally, one can observe that using the recomputation method that utilises the alternating hull algorithm also leads to only a small number of discrete jumps. However, what is interesting here is that if at least one discrete change occurs using this method, the

average and maximum affected area a_{avg} and a_{max} are meaningfully impacted as well. It can therefore be concluded, that this approach is good in circumventing most of the discrete jumps the recomputation method using the expansion method is prone to do. However, if a discrete change still occurs, it is likely to be a substantial one, as the example displayed in Figure 5.31 shows. Although the recomputation method using the alternating hull algorithm, therefore, can create significant jumps, they are not as big as some of the discrete changes that take place using the expansion method.

CHAPTER 6

Conclusion and Future Work

In this work, we proposed a new visualization style for hypergraphs, that takes inspiration from existing hypergraph visualization techniques like KelpFusion [MRS+13] and Kelp Diagrams [DvKSW12] and combines them with the graph drawing concept of fat edges [DEKW06]. In contrast to other state-of-the-art approaches, we designed our approach with so-called obstacles in mind from the start. These obstacles are not only helpful in a static environment, where they can be used to alter the routing of hyperedges but are also a great tool to manipulate the visualization outcome of an instance in a kinetic environment, where the obstacles are able to move.

To efficiently create a visualization, we came up with an algorithm that utilizes the properties of the convex hull to find a routing for multiple hyperedges through a plane of obstacles in a restricted environment of static instances. A noteworthy property of this approach is that it uses concentric rings around obstacles (so-called orbits) to adjust the width of the edges already throughout the routing process. This allows the algorithm to use fat edges while still properly handling obstacles that are only reached by the hyperedges if they have a certain thickness. In addition to that we introduced a variant of this algorithm which uses only disjoint orbits and a naive alternative approach that first creates a set path routing and then expands its paths. This naive approach, named the expansion method, also makes use of the orbit model but relies on a path routing that is created using only zero-width paths.

To easily test these algorithms we created a simple GUI that allows for basic instance generation and manipulation. This tool was further used to create a set of semi-randomly generated test instances, on which we evaluated the achieved visualizations of the algorithm against a more naive approach that treats the hyperedges as zero-width paths during the routing process and expands them after the fact.

For the evaluation itself, we defined metrics that aim to objectively judge the visualization outcomes depending on how closely they represent our aesthetic goals (thickness and

uniformity). In the majority of test instances, our algorithm was able to succeed in both of these criteria, especially when compared to the naive expanding approach as well as a slightly modified variant of the alternating hull algorithm that only allows for disjoint obstacle orbits.

Finally, we applied the learnings from the static visualization in a kinetic environment, where we allowed the obstacles of an instance to move in straight lines under certain conditions. To visualize this movement we proposed an approach that interpolates between different states throughout the movement. The interpolation between these states is straightforward, but the states, between which we interpolate, represent significant changes of the instance that we cannot naively interpolate over. Therefore, the frames in which these states occur are referred to as critical frames and need to be treated specially. This is done by either recomputing them with a static algorithm (stateless) or using a pre-computed, altered version of the frame, which can help to reduce the amount of discrete changes (state-aware).

Our approach to the visualization of kinetic instances is not too different from other state-of-the-art approaches that handle time-varying data. However, instead of prematurely computing the critical frames, like for example kinetic data structures [Bas99] do with their events, we opted to use a naive technique that preemptively computes every single frame throughout the movement to identify these critical frames. This method allowed us to demonstrate the possible capabilities of such an interpolation approach without the need to implement such a kinetic data structure (which we consider out-of-scope for this thesis).

Evaluation results of our approach to movement visualization, where we focused on smooth motion and the avoidance of discrete changes, showed that our interpolation approach compared favourably against a naive approach that utilizes different static algorithms introduced in this paper to recompute the visualization in every frame. Especially the introduction of state-awareness through the altering of critical frames helped with reducing not only the number of discrete changes but also the area they impact when compared to the naive approach that is stateless.

The main contribution of this work, therefore, is the proposal of a new visualization technique for hypergraphs, that draws its hyperedges as fat edges and is not only capable of routing its hyperedges around obstacles, but rather makes them an integral part of the visualization itself with the introduction of orbits. Furthermore, we showed that this visualization technique has the potential to be used in a kinetic environment in an efficient way.

Future Work. The possibilities for future work that connects to this thesis are numerous. For one, we limit our visualization efforts in this thesis to a restricted subset of hypergraph instances. Extending the static algorithm with its core ideas to work on general instances of hypergraphs would therefore be a significant albeit not trivial addition to the work presented in this thesis.

Another addition we envisioned is the manipulation of orbit sizes in order to achieve

different visualization styles with the same base algorithm. The alternating hull algorithm favours the thickness of the routed set paths over uniformity. However, with the concept of orbits, introduced in this thesis, it is easy to imagine a variant of the routing algorithm that aims to keep the set paths as uniform as possible. To that end, it is conceivable that all orbits used by a set path adapt to the smallest orbit used by the path and even that we would allow orbits of the same obstacle to take on different radii, to prevent unwanted implications of such a change.

In addition to that, even more distinct visualization styles can be achieved by adapting the orbit sizes. Only allowing for a single turn (tipping point between a set becoming thicker and thinner or vice versa) between set items is another example, that can conceivably be achieved by adjusting the sizing of the orbits and would lead to a different visualization outcome. To quantify the quality of the different visualization styles we imagine a case study that empirically evaluates them.

Finally, the state-aware interpolation method for kinetic instances introduced in this thesis is a great starting point to efficiently visualize the movement of obstacles. However, as discussed in Section 4.2, the implementation presented in this work relies on preemptively computing all frames of the movement to identify the critical frames of an instance. To replace this naive technique with a more sophisticated one akin to kinetic data structures [Bas99] we would need more insight into which pair of obstacles are relevant at any given point in time. While we conjecture that this structure can be captured in a planar graph that changes over time, defining this graph is highly non-trivial.

List of Figures

List of Algorithms

Bibliography

[AMA+16] Bilal Alsallakh, Luana Micallef, Wolfgang Aigner, Helwig Hauser, Silvia Miksch, and Peter J. Rodgers. The state-of-the-art of set visualization. *Comput. Graph. Forum*, 35(1):234–260, 2016.

[ARRC11] Basak Alper, Nathalie Henry Riche, Gonzalo A. Ramos, and Mary Czerwinski. Design study of linesets, a novel set visualization technique. *IEEE Trans. Vis. Comput. Graph.*, 17(12):2259–2267, 2011.

[Bas99] Julien Basch. *Kinetic data structures*. PhD thesis, Stanford University, USA, 1999.

[Bes03] Sergei Bespamyatnikh. Computing homotopic shortest paths in the plane. *J. Algorithms*, 49(2):284–303, 2003.

[Cha96] Timothy M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discret. Comput. Geom.*, 16(4):361–368, 1996.

[CPC09] Christopher Collins, Gerald Penn, and Sheelagh Carpendale. Bubble sets: Revealing set relations with isocontours over existing visualizations. *IEEE Trans. Vis. Comput. Graph.*, 15(6):1009–1016, 2009.

[CvGM+19] Thom Castermans, Mereke van Garderen, Wouter Meulemans, Martin Nöllenburg, and Xiaoru Yuan. Short plane supports for spatial hypergraphs. *J. Graph Algorithms Appl.*, 23(3):463–498, 2019.

[DEKW06] Christian A. Duncan, Alon Efrat, Stephen G. Kobourov, and Carola Wenk. Drawing with fat edges. *Int. J. Found. Comput. Sci.*, 17(5):1143–1164, 2006.

[DvKSW12] Kasper Dinkla, Marc J. van Kreveld, Bettina Speckmann, and Michel A. Westenberg. Kelp diagrams: Point set membership visualization. *Comput. Graph. Forum*, 31(3pt1):875–884, 2012.

[Gra72] Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inf. Process. Lett.*, 1(4):132–133, 1972.

[JWKN21] Ben Jacobsen, Markus Wallinger, Stephen G. Kobourov, and Martin Nöllenburg. Metrosets: Visualizing sets as metro maps. *IEEE Trans. Vis. Comput. Graph.*, 27(2):1257–1267, 2021.

[Meu19] Wouter Meulemans. Geometrycore. `https://github.com/tue-alga/GeometryCore`, 2019. Commit: 4a69b087ab55ace20337e7555a4e3e9cfaebe4f2.

[MNKF90] Sumio Masuda, Kazuo Nakajima, Toshinobu Kashiwabara, and Toshio Fujisawa. Crossing minimization in linear embeddings of graphs. *IEEE Trans. Computers*, 39(1):124–127, 1990.

[MRS+13] Wouter Meulemans, Nathalie Henry Riche, Bettina Speckmann, Basak Alper, and Tim Dwyer. Kelpfusion: A hybrid set visualization technique. *IEEE Trans. Vis. Comput. Graph.*, 19(11):1846–1858, 2013.

[MSVW17] Wouter Meulemans, Bettina Speckmann, Kevin Verbeek, and Jules Wulms. A framework for algorithm stability. *CoRR*, abs/1704.08000, 2017.

[MW06] Petra Mutzel and René Weiskircher. Bend minimization in planar orthogonal drawings using integer programming. *SIAM J. Optim.*, 17(3):665–687, 2006.

[NSM+22] Soeren Nickel, Max Sondag, Wouter Meulemans, Stephen G. Kobourov, Jaakko Peltonen, and Martin Nöllenburg. Multicriteria optimization for dynamic demers cartograms. *IEEE Trans. Vis. Comput. Graph.*, 28(6):2376–2387, 2022.

[SA08] Paolo Simonetto and David Auber. Visualise undrawable euler diagrams. In *12th International Conference on Information Visualisation, IV 2008, 8-11 July 2008, London, UK*, pages 594–599. IEEE Computer Society, 2008.

[Sch21] Marcus Schaefer. Rac-drawability is $\exists \mathbb{R}$-complete. In Helen C. Purchase and Ignaz Rutter, editors, *Graph Drawing and Network Visualization - 29th International Symposium, GD 2021, Tübingen, Germany, September 14-17, 2021, Revised Selected Papers*, volume 12868 of *Lecture Notes in Computer Science*, pages 72–86. Springer, 2021.

[VPF+14] Jevgenijs Vihrovs, Krisjanis Prusis, Karlis Freivalds, Peteris Rucevskis, and Valdis Krebs. An inverse distance-based potential field function for overlapping point set visualization. In Robert S. Laramee, Andreas Kerren, and José Braz, editors, *Proceedings of the 5th International Conference on Information Visualization Theory and Applications, IVAPP 2014, Lisbon, Portugal, 5-8 January, 2014*, pages 29–38. SciTePress, 2014.

[WCW+22] Yunhai Wang, Da Cheng, Zhirui Wang, Jian Zhang, Liang Zhou, Gaoqi He, and Oliver Deussen. F2-bubbles: Faithful bubble set construction and flexible editing. *IEEE Trans. Vis. Comput. Graph.*, 28(1):422–432, 2022.