Karlsruhe Institute of Technology

Connecting Points with Low-Complexity Polynomial Curves in a Polygon

Master Thesis of

Fabian Klute

At the Department of Informatics
Institute of Theoretical Computer Science

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. rer. nat. Peter Sanders
Advisors:	Dipl.-Inform. Thomas Bläsius
	Dr. Tamara Mchedlidze
	Dr. Ignaz Rutter

Time Period: 1. November 2014 – 30. April 2015

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, 24th April 2015

Abstract

Adding an edge to an already existing graph drawing can be done in numerous ways. In this thesis we explore the idea of drawing the new edge using a polynomial curve. Generalizing the k-link-path problem to curves, we want to find a curve connecting two points, while respecting a given bounding polygon. For degree 2 curves we present a direct algorithmic solution. For any curve of higher degree the problem is more difficult to solve. Sampling the control points is the logical next step, but with a naive approach one needs a large amount of sample points. For this reason we look into the application of ε-nets, which allow us to reduce the size of the sample.

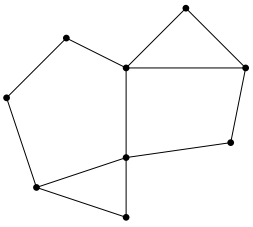
Deutsche Zusammenfassung

Eine Kante zu einer schon bestehenden Graphzeichnung hinzuzufügen kann auf viele Arten erfolgen. Die Darstellung solch einer Kante durch eine polynomielle Kurve ist Gegenstand dieser Arbeit. Eine andere Sichtweise ist, das Problem einen k-link Pfad zu finden, auf Kurven zu verallgemeinern. So betrachtet kann man das Problem auf das Finden einer Kurve, die gegebenen Start- und Endpunkt verbindet und innerhalb eines Polygons bleibt, übertragen. Für quadratische Kurven wird eine direkte, algorithmische Lösung angegeben. Kurven mit höherem Grad erweisen sich in der Betrachtung als wesentlich schwieriger. Die Kontrollpunkte der Kurve zu samplen ist eine Möglichkeit höhergradige Kurven zu handhaben. Der naive Ansatz braucht allerdings eine zu große Samplemenge. Als letzten Schritt betrachtet man ε-Netze, die es erlauben, die Größe der benötigen Sample einzuschränken.

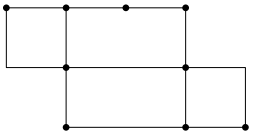
Contents

1. Introduction

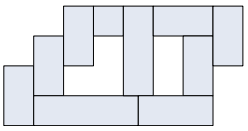
Most times a drawing of a graph is seen as something static. But there are cases when we want to add elements, extending an already existing drawing. For example, if the graph represents a data set and this set changes at some point, but only a smaller portion of the graph is affected. It then seems natural to conserve the untouched part, because it makes it easier for a viewer to track changes, then just delete the changed parts and finally insert the newly needed nodes and edges. Another case might be when a graph drawing is not satisfactory if drawn using a specific algorithm, but if we decompose it into several parts we can draw the whole graph nicely by starting with one part and then add the remaining elements later.
Taking the last case a step further we might even want to add the later parts in a different style. Over the years a lot of techniques to draw a graph were developed and they vary in how edges or nodes are represented. Figure 1.1 presents just a little excerpt. Some of the earliest ideas are force directed methods. The idea is to assign forces to the nodes and edges, then we iterate the system until the change from one step to another is very small. Figure 1.1(a) shows a graph created by a force directed layout algorithm. Kobourov recently presented a survey of such methods [Kob12]. Another popular method is to position the nodes on a grid and draw the edges with polylines, which are allowed to have bends of some predefined degrees. In Figure 1.1(b) we use bends of 90°. The result is a so called orthogonal drawing. Other methods completely changes the representation of nodes and edges. An example for this is shown in Figure 1.1(c). Each box represents a node and two nodes are connected if the two corresponding boxes touch each other. Finally, Figure



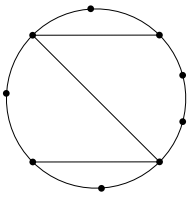
(a) Force directed layout.



(b) Orthogonal drawing.

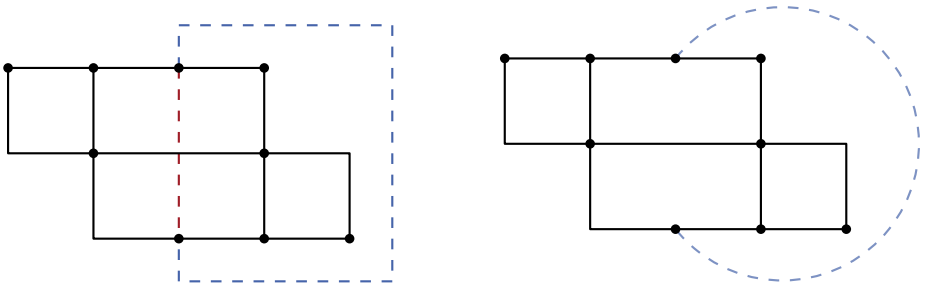


(c) Contact representation.



(d) Draw some edges as curves.

Figure 1.1.: A simple graph consisting of two triangles and two cycles, one of length four and one of length five. Figure 1.1(a) and 1.1(b) are generated with yEd [yWo15]. The Figures 1.1(c) and 1.1(d) are drawn by hand.



(a) Using polyline edges. (b) Add the edge as a curve.

Figure 1.2.: Adding one edge to an existing drawing. Figure 1.2(a) shows the trade-off between bends and crossings we have to keep in mind. The other drawing in Figure 1.2(b) shows how a curve could be used to draw the edge.

1.1(d) displays a mix of drawing styles. While a part of the graph is drawn with straight lines, some nodes are connected by curves. For a collection of papers and introductory material to graph drawing have a look into the paper of Di Battista et al., who collected a comprehensive list of papers on graph drawing [DETT94].
Deciding for a style of drawing often defines what we want to optimize, but there are some more general criteria a lot of techniques share or aim to fulfil. The perhaps most often tracked metric is the number of crossings between a pair of edges, though finding a crossing-minimal drawing is shown to be $\mathcal{NP}$-hard [GJ83]. Another metric we often try to maximize is the angular resolution between two edges at a node. If the angle between two edges is too small it is very hard to distinguish them as they enter the node. If we draw the edges as polylines like in Figure 1.1(b) the number of bends is interesting to us. Drawing an edge with a high number of bends suggest it is routed overly complex and it makes it hard for the user to follow it from one node to another.
Often we need to make a trade-off between different metrics. Figure 1.2 shows an example of a graph where we add an edge to an orthogonal drawing. While the red edge introduces a crossing, the blue one has a lot of bends. In fact Chan et al. [CFG$^+$14] have shown that if the graph is drawn with straight lines and extended by edges, drawn as polylines with bends, the number of bends we need is increasing linear with the size of the already drawn graph, if we want the drawing to be planar. In the end it is hard to say one drawing is better than the other. However in Figure 1.2(b) shows another variant, in which we draw the edge as a curve. Comparing the curved edge to the other two we can still talk about crossings, but we would need some notion on curvature. Again it is a very subjective thing to decide if a drawing is better than another, however the picture suggest the mixing of two styles is an interesting way of drawing the graph.
Finding a curve between two nodes is the task we are looking into in this thesis. To find just a random curve is not hard, but if we apply the above constraints again we get a more interesting problem. Focusing on the crossing minimization we have to find a way of restricting the curves, such that they never intersect if it is unnecessary. A possible way of doing this, is to find boundary polygons in the drawing, containing the two nodes we want to connect by an edge. If we do this in a way, such that the polygons do not intersect and the curves stay inside their polygon we can guarantee that there is no crossing created by adding new edges.
The problem we can derive from this, is to find a curve between two given points (the nodes) in the plane and restricting it to a given boundary polygon, which contains the two nodes connected by the edge. The type of curves we use to draw the edges are so called polynomial curves. A polynomial curve is defined by control points and weights for each point, making it fairly easy to handle. Further we can build a complex and smooth curve

from curves of lower degree. This makes it possible for us to work with low degree curves and still be able to form more complex shapes.
Looking for existing algorithms to draw a polynomial curve such that it respects a given bounding polygon yields a few results. The approaches though rely on a previously computed set of points or polyline and then an approximation is found, fitting the curve to this set or line [Sch90, DGKN97]. If there is not a really good set of points or polyline we approximate the approach runs into a dead end. Especially the case of adding multiple edges as curves, while guaranteeing to add no unnecessary crossings is difficult to realize, because we have no control over where the curve lies exactly and making it respect a boundary polygon gives us overly complex curves.
To formalize further we look at an older, already well studied problem. Given two points u,v and some obstacles in the plane, a k-link path is a polyline consisting of k segments, such that the path connects u and v. Further we say the path is not allowed to intersect any obstacle. The original k-link path problem was given by Toussaint and is solvable in linear time as shown by Suri [Sur86]. A related problem definition asks for a path with minimal euclidean length. This is a very well studied problem as well [LP84, SS86, Mit91]. Combining the two metrics was studied by Mitchell et al. [MRW92] and it yields short paths in both senses.
We aim to generalize the definition of a minimum k-link path to incorporate curves, calling it k-curve path.

Definition 1.1 (***k***-curve path)**.** *Given two points $u, v \in \mathbb{R}^2$, a set of obstacles in the $\mathbb{R}^2$ and an integer $k \in \mathbb{N}_+$, find a path of degree d polynomial curves $C_1, ..., C_k$, such that C_1 starts at u, C_k ends at v and no curve $C_i, i = 1, ..., k$ intersects an obstacle.*

Setting $d = 1$ gives the original problem of a k-link path. The above definition has no restriction on how the single segments of the curve are connected. On the one hand this allows us to very freely route the single curves, but on the other hand the whole curve, consisting of $C_1, ..., C_k$, should be smooth. To guarantee this we extend the above definition.

Definition 1.2 (***k***-curves smooth path)**.** *Given two points $u, v \in \mathbb{R}^2$, a set of obstacles in the $\mathbb{R}^2$ and an integer $k \in \mathbb{N}_+$, find a k-curves path connecting u and v, such that the curve consisting of C_i, C_{i+1} is $d - 1$ times differentiable for all $1 \leq i \leq k$.*

Contribution and Outline

Solving the general k-curve path is beyond the time constraint of this thesis and we focus on solving the problem for $k = 1$. Finding a solution to the general problem remains as future work.
The thesis is structured in the following manner. In Chapter 2 we will formally introduce polynomial curves, some other geometric terms and concepts and talk about graphs, which we later need to demonstrate an application. The next two chapters, 3 and 4, contain the main theoretical solution to the 1-curve path problem. First, we build an analytical solution for the case of $d = 2$ by looking at all points which result in an intersection between graph and curve. Afterwards, in Chapter 4, we consider the possibility of sampling the control points in an intelligent way, which does not need too many sample points. After introducing ε-nets we try to apply them to our problem, but meet some problems, when bounding the VC-Dimension proves to be harder than initially thought. Nonetheless we can show that, if it is possible to find the needed bounds, small samples can be guaranteed. Especially these sample sets do not depend on the diameter of the area containing all control points we want to avoid. In Chapter 5 we return to the starting point of the introduction. Taking a look at a scenario, where we want to add edge as curves to an existing drawing. Finally we finish with the conclusion and future work in Chapter 6.

2. Preliminaries

In this section we introduce some general terms and concepts. The first part is Section 2.1, where we introduce some general graph terminology. They are mostly used in the application chapter in the end of this thesis, but we need to begin with them, because several notions on polygons are defined using graphs in Section 2.2. Finally we conclude the preliminaries by introducing polynomial and, as a special case, Bézier curves in Section 2.2.2.

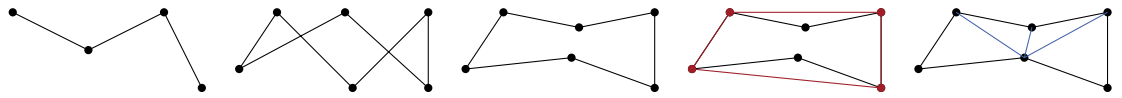
2.1. Graphs

A *graph* G is a pair of two sets. The first set we call V, the set of *nodes*, and the second one E, the set of *edges*. While V is a set of singular elements, E is a set of pairs of nodes from V, formally $E \subseteq \binom{V}{2}$. We write (u, v) for the edge between the nodes $u \in V$ and $v \in V$. In (u, v) we call u the *start* node and v the *end* node. A graph in which we find multiple edges between the same nodes is a *multigraph*. Two edges $e, f \in E$ that have the same start and end node are called *parallel*. If two nodes are connected by an edge, we say they are *adjacent* and with $\text{adj}(u)$ we denote the set of nodes adjacent to u. A node v that is adjacent to u is called a *neighbour* of u. If a node is either the start or end of an edge we say the edge is *incident* to the node. With $\text{inc}(u)$ we denote the set of all edges incident to $u \in V$. Further $\text{in}(u)$ is the set of all edges with u as end node and $\text{out}(u)$ is the set of all edges that have u as their start node. The *degree* of a node u is denoted with $\deg(u) := |\text{inc}(u)|$. The *indegree* of a node u is $|\text{in}(u)|$ and the *outdegree* is $|\text{out}(u)|$.
The graph G is said to be *undirected*, if for every edge $(u, v) \in E$ we find the reversed edge (v, u) in E and *directed* if this is not the case. An edge is a *loop*, if it has the same start and end node and a node with $\text{adj}(u) = \emptyset$ is said to be *isolated*.
A graph $G' = (V', E')$ is called a *subgraph* of G if the two sets V' and E' are subsets of V and E respectively. The subgraph G' is called *induced*, if for every two nodes $u, v \in V'$ we find an edge $(u, v) \in E'$ if and only if (u, v) is an edge in E.
A path is a collection of edges $e_0, ..., e_n$ where the end node of e_i is the starting node of e_{i+1} for $i \in [0, n-1]$. The path $e_0, ..., e_n$ leads from the start node of e_0 to the end node of e_n. We forbid an edge or node to appear twice in a path. This means no two edges share the same start and end node. If a node u is part of a path and neither start node of e_0 nor end node of e_n, it appears exactly once as start node of an edge e_i and as end node of the edge e_{i+1}, with $i = 1, ..., n-1$. A path is a *cycle* if the start node of e_0 and the end node of e_n are the same. Finally we say a node $v \in V$ is *reachable* from a node $u \in V$ if the two

nodes are connected by a *path*.
An undirected graph G is *connected* if for any pair of nodes in V we find a path between them. A directed graph is *weakly connected* if adding all missing reversed edges (which is the undirected version of the graph) results in a connected graph. We call it *strongly connected* if for every pair of nodes u, v we find a path between them, without introducing any new edges. A graph is said to be *k-edgeconnected* if the removal of $k-1$ edges is not enough to make it disconnected.
Throughout the thesis we need a few operations on graphs, namely *deleting*, *inserting* and *contracting*. To avoid confusion, let us define them in this section. Formally, applying an operation to a graph G results in a new graph $G' = (V', E')$ with a changed set of nodes and edges. We start with deleting edges or nodes. To delete a node $u \in V$ means we set $V' := V \backslash \{u\}$ and $E' := E \backslash \text{inc}(u)$. If we delete an edge $e \in E$ we set $E' := E \backslash \{e\}$ and leave $V' := V$. Next inserting a new node u' into G means we set $V' := V \cup \{u'\}$ and leave $E' := E$. Inserting an edge between two nodes $u, v \in V$ means to set $E' := E \cup \{(u, v)\}$ and $V' := V$. Finally contracting an edge $e := (u, v) \in E$ gives $V' := V \backslash \{v\}$ and $E' := E \backslash \text{inc}(v) \cup \{(u, v') | (v, v') \in \text{in}(v)\} \cup \{(v', u) | (v', v) \in \text{out}(v)\}$.
If not stated otherwise, all graphs we consider are *simple* and connected. A simple graph is no multigraph, has no loops and is undirected.

2.1.1. Planar Graphs

This section offers a quick overview of *planar graphs* and introduces a few basic properties we use in later sections. A planar graph is one that can be drawn without two edges crossing each other.
We define a *drawing* of a graph, as the mapping of nodes to points in the $\mathbb{R}^2$ and of edges to Jordan curves. We assume no two nodes are mapped to the same point, an edge always starts at the coordinate of its start node and ends at its end points coordinate and no edge includes a point already associated with a node (besides its start and end point). If the graph is planar we can draw it without any intersections between the curves. Such a drawing is called a *planar drawing*.
If we delete all edges and nodes in a planar drawing the remaining connected regions are called *faces*. The *boundary* of a face f is the sequence of edges bounding the connected region of f in the $\mathbb{R}^2$. Two faces are *adjacent* if they share a common edge along their boundaries. An *embedding* of a planar graph are all the planar drawings preserving the cyclic order of the boundaries around the faces. For the same graph we can get different sets of faces if we change the embedding.
The *dual graph* $G^\star = (V^\star, E^\star)$ of an embedded graph is the graph with $V' := \{u | u \in F\}$ and $E' := \{(u, v) | u \text{ adjacent to } v\}$. So for every edge on a boundary we get an edge in the dual graph and we say the edge on the boundary is crossed by the edge in the dual graph. This definition allows for parallel edges, which means the dual graph can be, and most likely is a multigraph. With *primal graph* we denote the graph G, which was used to derive $G^\star$. The *primal edge* $e \in E$ to a dual edge $e^\star \in E^\star$ is the edge that is crossed by $e^\star$ along the boundary.
Finally we want to introduce a special variant of planar graphs, a *tree*. The undirected graph $G = (V, E)$ is an (undirected) tree if and only if it is connected and has no cycles. A node in a tree is called *leaf* if it has degree one and *internal node* if it has degree at least two. A directed graph is a (directed) tree, if its undirected variant is a tree and we find a node u, such that for every other node v we find exactly one path from u to v. We call u the *root* of the tree. Further if (u, v) is an edge in a directed tree we call u the *parent* and v its *child*.



(a) A polyine and its vertices (b) A polygon (c) A simple, non-convex polygon (d) The convex hull of a simple polygon (e) A triangulation of a simple polygon

Figure 2.1.: Polygon Examples

2.2. Geometric Notions

The biggest part of this thesis focuses on finding a polynomial curve inside a polygon. The next two sections present a short introduction to the most commonly used geometric properties we use.

2.2.1. Polygons

Similar to a face in a planar graph we define *polygons* as a collection of points in the $\mathbb{R}^2$ which are connected not by Jordan curves, but lines. The points are *vertices* and the connecting lines we call *edges*, just as in graphs. This definition makes it easy to treat a polygon as a graph, with the vertices being the nodes and the edges of a polygon forming the set of edges in the corresponding graph. It allows us to use the same terminology as defined in Section 2.1.
We assume all polygons are *connected* in the same sense as we assume our graphs are connected, in Section 2.1. In Figure 2.1(b) you can see an example of a polygon. If the polygon is not closed we call it a *polyline* (Figure 2.1(a)). In the example the lines cross each other. It is hard to work with such polygons and we assume that any polygon we encounter is also a planar drawing of a graph. Such a polygon is called *simple*.
We call the collection of edges between vertices the *boundary* and whatever is in between the edges is the *interior* of the polygon. If a part in the interior is surrounded by edge of the polygon we call it a *hole*. A *convex* polygon is a polygon, in which for any two points x and y in the interior we find exactly two crossings of the line going through x and y and the boundary of the polygon. Given a simple polygon we can find the *convex hull* of the vertices, which is the smallest collection of edges, such that the vertices lie on the boundary of a convex polygon or lie in the interior. For an example look at Figure 2.1(d).
Finally we need the notion of a *triangulation*. Given a simple polygon we can triangulate it by interpreting it as a graph and add edges between the nodes until every face of the graph is a triangle (see Figure 2.1(e)).

2.2.2. Polynomial Curves

In this section of the preliminaries we introduce polynomial curves and, as a special case, Bézier curves. We closely follow the introduction to polynomial and Bézier curves presented by Beatty et. al. [BB87, Chapter-2,10]. The for us interesting curves all lie in the $\mathbb{R}^2$, we only talk about two-dimensional polynomial curves. Usually they are represented by parametric equations:

$$Q(t) = (x(t), y(t)),$$

where $t \in [a, b]$, with $a \leq b$, is the parameter of Q and the two functions $x : \mathbb{R} \to \mathbb{R}$ and $y : \mathbb{R} \to \mathbb{R}$ represent the x- and y-coordinate of a point on the curve. For example if we look at a simple unit circle it can be represented as $x^2 + y^2 \leq 1$, which is the so called implicit form, or in its parametric form as $Q(t) := (\cos t, \sin t)$ with $t \in [0, 2\pi]$.
This small example already raises the question of how we find the functions $x(t)$ and $y(t)$. If there is an implicit version given we can perhaps transform it, but if we have to come up

with the function from scratch and if the curve gets more complex, it is not really practical to construct the whole curve at once. The solution is, to break it up into so called *segments* and join them back together.
To break up the curve we have to take a look at the parameter t. Suppose we want to split the curve at a point $t_k \in [a, b]$. A split point is also called a *knot*. Now x and y can be written as piecewise functions, where we start at u, go through the knot t_k and end at v. At the joints we need to make sure the curve is still smooth and not forming any jumps or sharp corners. The usual way to do this, is to require $Q(t)$ to be C^{d-1} continuous.

Bézier Curves

The polynomial curves we are interested in are so called *Bézier curves*. They were developed by designers at Renault and Citroën and are widely used in all areas of design. Bézier curves use the *Bernstein polynomials* $b_{i,n}(t)$ as Basis, with

$$b_{i,d}(t) := \binom{d}{i} t^i (1-t)^{d-i}.$$

A Bézier curve of degree d is then defined as:

$$B_d(t) := \sum_{i=0}^{d} b_{i,d}(t) p_i$$

with the p_i being the control points of the Bézier curve. We call $B_d(0) = p_0 = p_s$ the *start point* and $B_d(1) = p_d = p_e$ the *end point* of the curve. The other control points p_i are called *inner control points*. It is important to note a Bézier curve does not pass through the inner control points, but just comes close to them.

Joining Bézier Curves

As already stated above we often want to join two curves together. Here we demonstrate how to do this with Bézier curves. Suppose our first curve is defined by the points $\{p_0, ..., p_d\}$ and the second one is using $\{q_0, ..., q_d\}$, with $p_d = q_0$. The resulting curve is C^0 continuous. Figure 2.2(a) shows why this is not enough. At the joint the curve can move arbitrarily and it does not look smooth at all. To guarantee C^{d-1} continuity we have to look at the derivatives of the curve. <In general we get that the first derivative of a Bézier curve as:

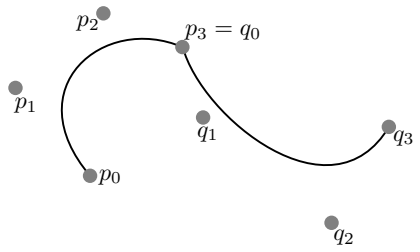
$$\frac{d}{dt} B_d(t) = \sum_{i=0}^{d-1} B_{d-1,i}(t) * d(p_{i+1} - p_i)$$

We can see the derivative is again a Bézier curve with a lower degree and different control points. To write the higher order derivative we define the following notation. Let $i \in \{0, ..., d\}$ and $n \in \{0, ..., d-1\}$, then

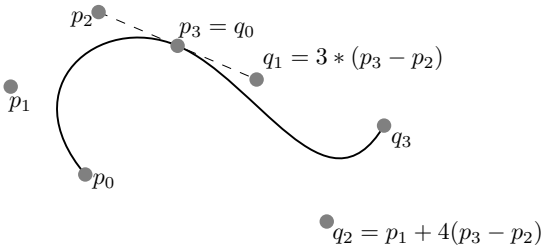
$$D_i^1 = p_i$$
$$D_i^n = D_{i+1}^{n-1} - D_i^{n-1}.$$

In general we can now write the n-th derivative of a Bézier curve as

$$\frac{d^n}{dt^n} B_d(t) = \prod_{j=0}^{n} (n-j) \sum_{i=0}^{d-n} B_{d-n,i}(t) * D_i^n.$$



(a) Joining the two curves without any constraints beside $p_0 = q_0$.



(b) Joining the two curves with constraints for C^2 continuity.

Figure 2.2.: Joining two cubic Bézier curves defined by the control point sets $\{p_0, ..., p_d\}$ and $\{q_0, ..., q_d\}$ together.

Evaluating the derivatives for $t = 0$ and $t = 1$ we get the constraints we have to keep satisfied at the knots. First the ones for C^1 continuity:

$$\frac{d}{dt}B_{i,d}(0) = d(p_1 - p_0)$$
$$\frac{d}{dt}B_{i,d}(1) = d(p_d - p_{d-1}).$$

Geometrically these constraints tell us we have to assure the second to last control point of the first curve, namely p_2, and the first control point of the second curve, namely q_1, lie on a straight line and the knot $p_e = q_s$ is exactly at the midpoint of the segment between p_2 and q_1. Figure 2.2(b) shows an example of this method. If we move the knot, C^1 continuity can be preserved by moving p_2 and q_1 accordingly. This means the curve only changes locally, especially no other segments than the two that are adjacent to the knot have to be modified.
If the curve has to be $n = d - 1$ times continuous at the joints we have to apply the following constraints:

$$\frac{d^n}{dt^n}B_d(0) = \left(\prod_{j=0}^{n}(n-j)\right) D_0^n$$
$$\frac{d^n}{dt^n}B_d(1) = \left(\prod_{j=0}^{n}(n-j)\right) D_1^n.$$

These constraints fix us nearly all control points of the curve. In fact it leaves only the start point p_s and the end point q_e free. This means we can not easily move the knot, because the control points of more than the adjacent segments would have to be modified. Since most applications don't require curves with a degree higher than three we only have to guarantee C^2 continuity at the joints. Again look at Figure 2.2(b) for an example. If we want to move a control point after the initial Bézier curve was constructed maintaining C^{d-1} continuity is nearly impossible, since we run into the problem of changing all control points. In practical cases with degree 3 curves it is often enough to preserve C^1 continuity.

Subdividing Bézier Curves

Given a Bézier curve $B_d(t)$ we might be interested in subdividing the curve into two segments. Both segments should have the same degree as the original curve and the joint segments should form the same shaped curve. Let $p_0, ..., p_d$ be the control points of our curve. First we need the following identity:

$$B_d(t) = (1-t)B_{d-1}^l(t) + tB_{d-1}^r(t), \tag{2.1}$$

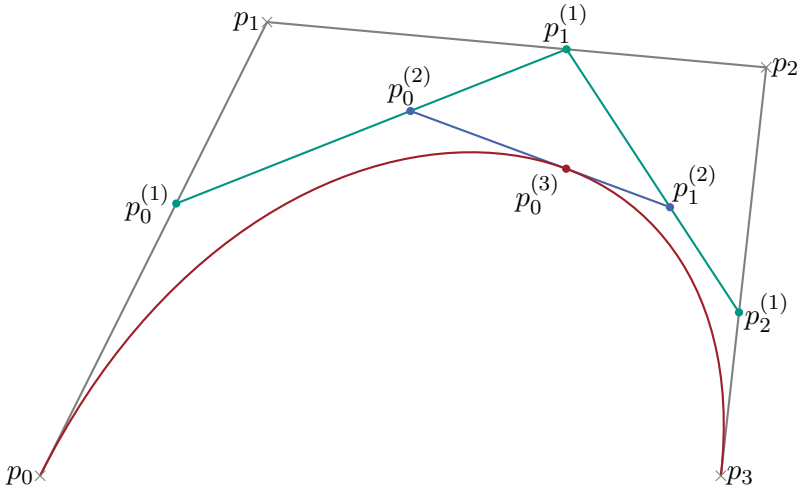


Figure 2.3.: Finding the point $B_n(t)$ with $t = 0.6$.

where B^l is defined on the control points $p_0, ..., p_{d-1}$ and B^r on $p_1, ..., p_d$. Basically the above equation states that the initial Bézier curve is formed as the convex combination of two curves with degree $d-1$. A proof of this identity can be found in [BB87].
To compute new control points we need to rewrite Equation 2.1 as

$$B_d(t) = \sum_{i=0}^{d-1} b_{i,d}(t)((1-t) * p_i + t * p_{i+1}). \tag{2.2}$$

Next we define:

$$\begin{aligned} p_i^{(r)}(t) &:= (1-t)p_i^{(r-1)} + tp_{i+1}^{(r-1)} \\ p_i^{(0)} &:= p_i. \end{aligned} \tag{2.3}$$

We use Equation 2.2 with the just defined recursive Formula 2.3 to subdivide the control points r times:

$$B_d(t) = \sum_{i=0}^{d-r} b_{i,d}(t)p_i^{(r)}(t).$$

What we basically do is to interpolate linearly between every two control points, in the next step between those newly found points and so on. Now set $r = d$ and we end up with:

$$B_d(t) = p_0^{(d)}(t).$$

This means we can find a point on the initial Bézier curve by evaluating Equation 2.3 for $r = d$. An example of this process is shown in Figure 2.3. To come back to our original task, which was to subdivide the curve, we start by noticing that the original start and end point remains the same and the point where we want to subdivide is just $p_0^{(d)}$. This point then is used as end point for the left and as starting point for the right part of the curve. As control points we choose always the last and first point of an iteration step. In the case of a cubic curve this would result in the left curve being formed by the control points $p_0^{(0)}, p_0^{(1)}, p_0^{(2)}, p_0^{(3)}$ and the right curve is defined by $p_0^{(3)}, p_1^{(2)}, p_2^{(1)}, p_3^{(0)}$.

Drawing Bézier Curves

So far we looked at the definition of a Bézier curve and if we wanted to draw one, the straightforward way would be to just evaluate the polynomials. Since this method is not

numerically stable, we want to find a more stable approach. The *de Casteljau* algorithm offers a solution. It is easy to implement, runs in $O(n^2)$ and is numerically stable. Algorithm 2.1 shows the pseudocode of an inplace variant. A call to deCasteljau returns, for a given parameter t_0, the point $B_n(t_0)$. We then can draw the Bézier curve in a loop, where we go from $t_0 = 0$ to $t_0 = 1$ in sufficiently small steps. Lets look at Algorithm 2.1 and find out how it works.
Besides t_0 it takes another parameter $l \in \mathbb{N}$ and a set of points in the $\mathbb{R}^2$. We assume the first and the last point in the set lie on $B_n(t)$. The idea is to subdivide the curve with Equation 2.3, until we reduced the set of points to one point. If there is only one point left we return it (see Line 3). Else we process the points, reducing the size of our point set by one. Instead of allocating new space we just store the current size in l. Finally we recurs in Line 8.

Algorithm 2.1: The deCasteljau algorithm to find a point on a Bézier curve.

```
   input: Set of points in ℝ², Parameter l := |points|,Parameter t₀ ∈ [0,1]
1  Function deCasteljau(t₀, l, points)
2      if |points| = 1 then
3          return points[0]
4      else
5          − − l
6          for k ← 0 to l do
7              points[k] ← (1 − t₀) ∗ points[k] + t₀ ∗ points[k + 1]
8          return deCasteljau(t₀, l, points)
```

3. Fitting Bézier Curves into Simple Polygons

As said in the introduction we can not solve the k-curves path problem right away. Instead we start by solving it for $k = 1$. This means we want to find a Bézier curve $B_d(t)$ with a possibly low degree connecting two points in a polygon. In the following chapter we assume, we are not only given the start and end point, but also $d - 2$ inner control points of a degree d Bézier curve. The task then is to find the last control point, such that the curve does not intersect the polygon. For most of the calculations in this chapter we used Mathematica by Wolfram Research [Wol14].
We formally describe the configuration of our system in Section 3.1. The next part, Section 3.2, is about finding bad control points for one line segment and a Bézier curve. As a running example we will use quadratic Bézier curves. Afterwards we put the polygon back together (Section 3.3). The methods developed in Section 3.2 are finally applied to curves with degree 3 in Section 3.4.

3.1. Configuration

Let $\mathcal{P}$ be a simple polygon with vertices $v_0, ..., v_m \in \mathbb{R}^2$. The assumption of it being simple is necessary, because it guarantees us a boundary exists and allows us to talk about inside and outside of $\mathcal{P}$. Secondly we have all, but the i-th inner, control points of a degree $d \in \mathbb{N}_+$ Bézier curve $B_d(t)$. They are collected in the set P_i, where i indicates the missing control point. The two points $p_s = p_0 \in \mathbb{R}^2$ and $p_e = p_d \in \mathbb{R}^2$ serve as the start and end points of the Bézier curve.
We often need to talk about x- and y-coordinate of a point or the x- and y-component of a parametric function with values in the $\mathbb{R}^2$. Let $g : [0, 1] \to \mathbb{R}^2$ be a function. With $(g(t))_x$ we denote its x- and with $(g(t))_y$ its y-component for parameter $t \in [0, 1]$. For a point $p_i \in \mathbb{R}^2$ with subscript i, we write x_i for its x-coordinate and y_i for its y-coordinate. If p has no subscript we write x_p and y_p.
If a curve intersects a polygon, it has to intersect at least one segment. This observation allows us to reduce the system we have to consider to a line segment and one Bézier curve. Let b and e be two consecutive vertices of $\mathcal{P}$. The segment can be expressed with the parametric equation $L_{b,e}(s) := b + s(e - b) = b + sc, s \in [0, 1]$. Further the system of a segment and control points is rotated and translated, such that the segment lies on the x-axis. Then the formulas get easier, because $y_b = y_e = y_c = 0$. To avoid unnecessary negative signs we say $x_c > 0$ at all times. This is no restriction, because we can rotate the

system by 180 degree to swap the roles of b and e. Further we assume that neither the start nor the end point of the Bézier curve lie on the segment.

3.2. Forbidden control points

The core of this section is to find all forbidden coordinates. That is any coordinate which leads to an intersection between $B_d(t)$ and $L_{b,e}(s)$. Curve and segment intersecting means we find a pair of parameters (s,t) such that $B_d(t) = L_{b,e}(s)$. Since p_s and p_e never lie on the segment we can exclude $t = 0$ and $t = 1$, which gets rid of some special cases where we would have to divide by zero later on. Now a point $p \in \mathbb{R}^2$ is called *forbidden* if and only if using it as the i-th control point of a Bézier curve $B_d(t)$ leads to an intersection between $B_d(t)$ and $L_{b,e}(s)$. Finding these points allows us to compute a curve not intersecting with at least this segment of $\mathcal{P}$.

Theorem 3.1. *Given a Bézier curve with control points $P = \{p_0, ..., p_d\}$ and a line segment $L_{b,e}(s)$. All forbidden points for the i-th control point are given by the following function:*

$$f_i(s,t,P_i,L) = -\frac{\sum_{j=0,j\neq i}^{d} b_{j,d}(t)p_j - (b+sc)}{\binom{d}{i}t^i(1-t)^{d-i}}$$

Proof. In order to find all control points that give us a crossing between $B_d(t)$ and $L_{b,e}(s)$, we need to identify for which p_i's the equality $L_{b,e}(s) = B_d(t)$ holds:

$$\begin{aligned} L_{b,e}(s) &= B_d(t) \\ b + sc &= \sum_{j=0}^{d} b_{j,d}(t)p_j. \end{aligned}$$

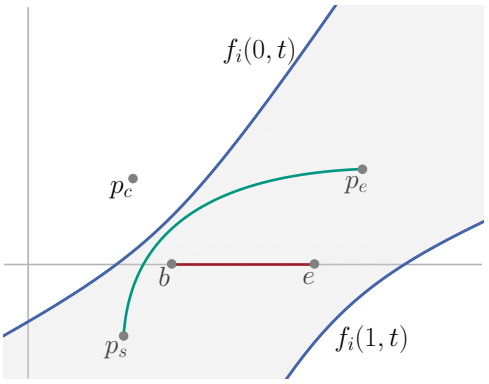
Reduce the equation to the i-th control point and bernstein polynomial of the Bézier curve:

$$\begin{aligned} b_{i,d}(t)p_i &= -\sum_{j=0,j\neq i}^{d} b_{j,d}(t)p_j - (b+sc) \\ p_i &= -\frac{\sum_{j=0,j\neq i}^{d} b_{j,d}(t)p_j - (b+sc)}{b_{i,d}(t)}. \end{aligned}$$

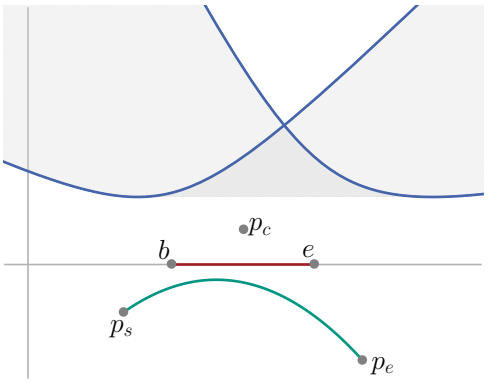
As claimed we find all the forbidden points as values of $f_i(s,t,P_i,L)$. □

Throughout this section P_i and L are considered fixed and we use the shorter $f_i(s,t)$ for $f_i(s,t,P_i,L)$. To get an idea what the values of $f_i(s,t)$ look like and how we might be able to work with them we use curves with $d = 2$ as an example. With quadratic Bézier curves there is only one control point p_c beside p_s and p_e. Figure 3.1 shows two example plots. The light grey area contains all the forbidden control points. The two thicker lines correspond to $f_1(t,0)$ and $f_1(t,1)$. Keep in mind, all plots shown in this chapter are really cuts through a higher dimensional body. For example the set of all forbidden control points in the case of $d = 2$ is a 6-dimensional body if we leave start and end point free. If they are treated as fixed points we reduce the number of dimensions to two. In the case of degree 3 the dimension would again increase by two, because we get another inner control point, and if start and end point of the curve are fixed the forbidden values for p_i still lie inside a four dimensional body.

Getting back to the case of all control points being fixed and only one left variable we can



(a) The case when start and end point of the Bézier curve are on either side of the line.



(b) Second case, start and end point are on the same side of the x-axis.

Figure 3.1.: Intersecting a Bézier curve with a segment. Green is the Bézier curve, red the segment and in blue/grey the area of all the $f(t,s)$.

define the area $\mathcal{F}_i(P_i, L)$ of all forbidden control points. We call it the *forbidden area* and it contains all values of $f_i(s,t)$:

$$\mathcal{F}_i(P_i, L) := \{p \in \mathbb{R}^2 | \exists (t,s) \in (0,1) \times [0,1] : f_i(s,t) = p\}.$$

We are able to create a crossing between a given Bézier curve and a line $L_{b,e}(s)$ by placing the i-th control point at $p_i = f_i(s,t)$ for some $s \in [0,1]$ and $t \in (0,1)$, but one might want to know where the crossing between $B_d(t)$ and $L_{b,e}(s)$ occurs. Suppose we have chosen $p_i = f_i(t_0, s_0)$ with some $s_0 \in [0,1]$ and $t_0 \in (0,1)$. Using p_i as the i-th control point in the Bézier curve $B_d(t)$ we get:

$$\begin{aligned} B_d(t) &= \sum_{j=0, j\neq i}^{d} b_{j,d}(t) p_j + b_{i,d}(t) p_i \\ &= \sum_{j=0, j\neq i}^{d} b_{j,d}(t) p_j + b_{i,d}(t) \left(- \frac{\sum_{j=0, j\neq i}^{d} b_{j,d}(t_0) p_j - (b + s_0 c)}{b_{i,d}(t_0)} \right) \end{aligned}$$

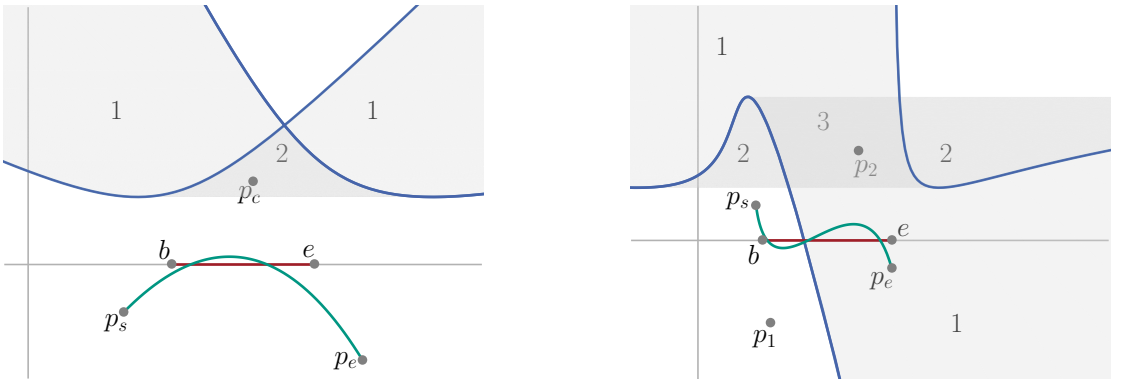
Some point on the curve corresponds to $t = t_0$ and the equation reduces to:

$$B_d(t_0) = b + s_0 c = L_{b,e}(s_0)$$

As one would expect from Theorem 3.1 we find the crossing at $B_d(t_0)$ and $L_{b,e}(s_0)$. So far we only looked at one crossing, but there might be multiple crossings between $B_d(t)$ and $L_{b,e}(s)$. To be exact there can be a maximum of d crossings between curve and segment. Suppose placing the i-th control point somewhere in the forbidden area results in d crossings, then intuitively we have to find d different pairs of (s,t) with equal values when used in $f_i(s,t)$. Otherwise we had found two sets of coordinates for the same control point. The following lemma shows the intuition is correct.

Lemma 3.2. *Given a Bézier curve $B_d(t)$, a line segment $L_{b,e}(s)$ and pairs $(t_0, s_0), ..., (t_j, s_j)$ with $t_k \neq t_l, l \neq k$, such that $L_{b,e}(s_k) = B_d(t_k)$ for all $k = 0, ..., j$, then $f_i(t_k, s_k) = f_i(t_l, s_l)$.*

Proof. Theorem 3.1 tells us for any crossing at $L_{b,e}(s_k) = B_d(t_k)$ we find a control point p_i at $f_i(t_k, s_k)$ for every $k = 0, ..., j$. Suppose $k \neq l$ and $f_i(t_k, s_k) \neq f_i(t_l, s_l)$, then there would be two different control points $p_i \neq p_j$, but then the degree d Bézier curve would have more than $d+1$ control points. □



(a) Degree two Bézier with two intersections and Γ.

(b) Degree three Bézier with three intersections and Γ.

Figure 3.2.: Displaying Γ for degree 2 and 3 Bézier curves. Whenever the control point p_c is placed in a region with $\Gamma(p) = c$ the curve intersects the line c times.

With Lemma 3.2 we can differentiate between cases. Suppose we have no possibility of placing the i-th control point, such that there is more than one crossing between $B_d(t)$ and $L_{b,e}(s)$. Then by Lemma 3.2 there are no two different pairs (s,t) resulting in a crossing between two $f_i(s,t)$. From this we can conclude there need to be two $f_i(s,t)$ which have all the forbidden control points between them. Figure 3.1(a) illustrates this. Placing start and end point above and below the x-axis, connecting them with a quadratic curve, allows for only one crossing. As mentioned the thicker lines represent $f_1(t,0)$ and $f_1(t,1)$ and they seem to bound the region of all forbidden control points. This holds for higher dimensions of Bézier curves as well. Whenever there is a maximum of one crossing between curve and segment the boundary of the forbidden area is consisting of $f_1(t,0)$ and $f_1(t,1)$, but the added flexibility makes the case differentiation a lot harder, as we see with cubic Bézier curves in Section 3.4.
Now suppose we allow more than one crossings between $B_d(t)$ and $L_{b,e}(s)$. Figure 3.1(b) gives an example for $d = 2$. Placing start and end point on the same side allows two intersections between curve and segment. The previously used functions $f_1(t,0)$ and $f_1(t,1)$ are no longer single handily bounding the forbidden area. We find an area seemingly "below" the crossing p_x that lies between $f_1(t,0)$ and $f_1(t,1)$ and it appears to be bounded by a straight line at the bottom.
Lets think about what happens. We follow $f_1(t,0)$ from left to right. It is the boundary up to a certain point, after this point the curve goes upwards to the crossing at p_x. All the time the Bézier curve has to maintain an intersection with b of $L_{b,e}(s)$. Placing the control point at p_x leaves us with the curve intersecting both start and end point of the segment, but as we progressed along $f_1(t,0)$ we had to intersect every other point on $L_{b,e}(s)$. This means $f_1(t,0)$ had to intersect every other $f_1(t,s)$ for $s \in (0,1)$. The same observation can be made for $f_1(t,1)$. From this we can see $f_1(t,0)$ and $f_1(t,1)$ are still bounding some area, not the whole forbidden one but they seem to separate the region where p_c leads to one and where p_c leads to two crossings. The remaining question is if we are able to find the segment bounding the area where we get two crossings.
Placing the control point in the area below the crossing of $f_i(t,0)$ and $f_i(t,1)$. We still get two intersections between $B_d(t)$ and $L_{b,e}(s)$, but when we move the control point down, towards the x-axis, the intersections are pushed closer together as well. This continuous until we reach the end of the region and it seems like the last crossing is on the segment of the area, exactly when $L_{b,e}(s)$ is a tangential to $B_2(t)$.
Formalizing the above observation is the subject of the next paragraphs. First we define

the *crossing number* $\Gamma(f_i(s,t))$ of a point $p \in \mathbb{R}^2$ as:

$$\Gamma(p) := |\{(t_c, s_c) \in (0,1) \times [0,1] : |f_i(t,s) = f_i(t_c, s_c)\}|.$$

The crossing number gives us a natural subdivision of the forbidden area, where every subregion is made up of all $f_i(s,t)$ with equal $\Gamma(f_i(s,t))$. Lets define a crossing region as:

$$\mathcal{C}(c) := \{p \in \mathbb{R}^2 | \Gamma(p) = c\} \text{ ,with } c \in \mathbb{N}_0$$

Figure 3.2 shows two examples of such regions. These regions might not be connected, as you can see in Figure 3.2(b) where two not connected regions with crossing number two exist. We define a *connected crossing region* as a subregion of some $\mathcal{C}(c)$, where for every point in the connected crossing region an ε-region with $\varepsilon > 0$ is inside the region as well or the point lies on the boundary. We denote a connected crossing region with $\mathcal{C}(c,p)$ where p is a point inside the region.
The forbidden area is the union over all the connected crossing regions and if we find their boundaries, we can compute a boundary for the whole forbidden area. Before we can present the theorem we look at an observation, regarding Bézier curves and its roots on the x-axis. With root $[B_d(t)]$ we denote the number of roots a Bézier curve has with the x-axis. Let $\delta, \nu \in \mathbb{R}^2$, with $[\delta, \nu]$ we denote all points in the $\mathbb{R}^2$, laying on a straight line between δ and ν. Next let $B_d(t)$ be a Bézier curve, we define $B_d^\delta(t)$ as the same Bézier curve, but with the i-th control point translated by δ. The equation of this new curve can be written in terms of the old curve:

$$B_d^\delta(t) = \sum_{j=0, j\neq i}^{d} b_{d,j}(t)p_j + b_{d,i}(t)(p_i + \delta) = B_d(t) + b_{d,i}(t)\delta.$$

Figure 3.3 shows what happens if we translate parallel to the x-axis (Figure 3.3(a)) or parallel to the y-axis (Figure 3.3(b)). The figures motivate the following two observations:

(i) If $\delta = \binom{x_\delta}{0}$ with $x_\delta \neq 0$, then $B_d^\delta(t)$ has the same number of roots and they are translated by x_δ.

(ii) A quadratic Bézier curve can have exactly a maximum of two roots. Let $B_d(2)$ be a quadratic Bézier curve with root $[B_d(2)] = 2$ and $\delta \in \mathbb{R}^2$ with root $\left[B_2^\delta(t)\right] = 0$, then there exists a $\delta' \in [0, \delta]$ with root $\left[B_2^{\delta'}(t)\right] = 1$.

Observation (i) says that the number of roots can not change if $y_\delta = 0$, consequently the number of roots of a Bézier curve can only be changed by a translation with $y_\delta \neq 0$.
et δ be a vector with $y_\delta \neq 0$ and we assume the number of roots decreased, root $[B_d(t)] >$ root $\left[B_d^\delta(t)\right]$. There has to be a $\nu \in [0, \delta]$, such that

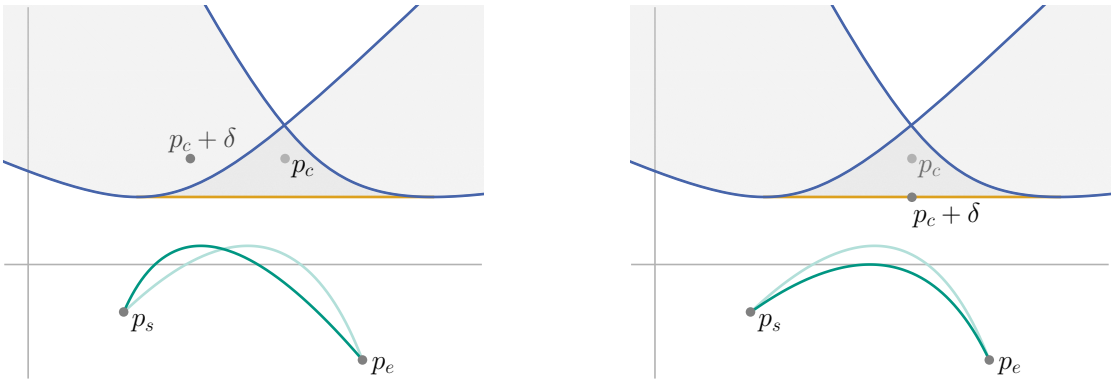
$$\forall \nu' \in [0, \nu] : \text{root}\,[B_d(t)] = \text{root}\left[B_d^{\nu'}(t)\right] \wedge \text{root}\,[B_d(t)] > B_d^\nu(t).$$

For every two roots at parameters t_0 and t_1 with $t_0 < t_1$ we know there exists a parameter $t' \in [t_0, t_1]$ such that

$$\left(\frac{d}{dt}B_d(t')\right)_y = 0.$$

At ν the number of roots changes for the first time by at least one. With Observation (ii) and because there had to be an extreme value between two roots we know there is at least one root where the x-axis is tangential to $B_d^\nu(t)$.
Another component we are going to need are the roots of the first derivative of $f_i(s,t)$. We define the function $t_i : \mathbb{R}^{2d} \to (0,1)^{d-1}$, with L constant, as:

$$t_i(P_i) = \text{root}\left[\frac{d}{dt}\left(f_i(0, t, P_i, L)\right)_y\right].$$



(a) Translation with $\delta = (x_\delta, 0)$.

(b) Translation with $\delta = (0, y_\delta)$.

Figure 3.3.: Translating the control point by a vector δ. In Figure 3.3(a) the control point is translated parallel to the x-axis and in Figure 3.3(b) the translation is parallel to the y-axis.

The next theorem proves the intuitive argument given above to be correct. No arbitrary $f_i(t, s)$ can be on the boundary, but only the ones with $s = 0, 1$ or t leading to $L_{b,e}(s)$ being tangential to $B_d(t)$.

Theorem 3.3. *Consider a Bézier curve $B_d(t)$ and a line segment $L_{b,e}(s)$. The connected crossing regions $\mathcal{C}(c, p)$ with $p \in \mathbb{R}^2$ being a forbidden point and $c \in \mathbb{N}_0$, have a boundary consisting of the curves $f_i(t, 0)$, $f_i(t, 1)$, $f_i(t_i(P), s)$.*

Proof. Let $L_{b,e}(s)$ be a line segment from b to e, $\delta \in \mathbb{R}^2$ and $p_i \in \mathbb{R}^2$ be the i-th control point of a degree d Bézier curve $B_d(t)$ with crossing number $c = \Gamma(p_i) > 0$. The connected crossing region p_i lies in is $\mathcal{C}(c, p_i)$. Further let $B_d^\delta(t)$ be the Bézier curve which we got by using $p_i + \delta$ as the i-th control point in $B_d(t)$. We assume $c \neq \Gamma(p_i + \delta) = c_\delta$, consequently $\mathcal{C}(c_\delta, p_i + \delta) \neq \mathcal{C}(c, p_i)$. Finally we say there is no $\nu \in [0, \delta)$ such that $\mathcal{C}(c, p_i) \neq \mathcal{C}(\Gamma(p_i + \nu), p_i + \nu)$
We consider two cases:

1. ($y_\delta = 0$) In this case the Bézier curve is only translated in the x-direction. From above we know the number of roots stays the same. or the number of intersection however this has not to be true, because we can translate a root such that it lies not on the curve or just do the opposite. Let us assume we reduced the number of crossings. Then there is a parameter $t' \in (0, 1)$, such that $B_d(t')$ is an intersection, but $B_d^\delta(t')$ is none. We know after translating the curve $B_d^\delta(t')$ is still a root and because the translation is a continuous operation on the curve, this means there has to be a $\nu \in [0, \delta]$, such that $B_d^\nu(t') = L_{b,e}(0)$ or $B_d^\nu(t') = L_{b,e}(1)$. Analogously if the number of intersections increased.

2. ($y_\delta \neq 0$) If a translation in y-direction is allowed and δ is the only vector in $[0, \delta]$ changing $\Gamma(p_i + \nu), \nu \in [0, \delta]$ and for no $\nu \in [0, \delta]$ $B_d^\nu(t') = L_{b,e}(0)$ or $B_d^\nu(t') = L_{b,e}(1)$ we find the number of roots had to change. But then we know from above that at δ we find at least one intersection between $B_d^\delta(t)$ and $L_{b,e}(s)$, such that $L_{b,e}(s)$ is tangential to $B_d^\delta(t)$.

We now know the crossing number can only change when there is a point $B_d^\delta(t) = L_{b,e}(0)$, $B_d^\delta(t) = L_{b,e}(1)$ or we changed the numbers of the roots the Bézier curve has with the x-axis. Because we set the segment $L_{b,e}(s)$ to be horizontal, the last step towards proving

the theorem is showing that the roots of $\frac{d}{dt}\left(B_d(t)\right)_y$ are equivalent to the roots of $\frac{d}{dt}f_i(s,t)$. Let $t_0 \in (0,1)$ be a root of $\frac{d}{dt}\left(B_d(t)\right)_y$, then we get

$$\begin{aligned}\frac{d}{dt}\left(B_d(t_0)\right)_y &= \sum_{j=0}^{d} \frac{d}{dt} b_{d,j}(t_0) y_j = 0 \\ -\sum_{j=0, j\neq i}^{d} \frac{d}{dt} b_{d,j}(t_0) y_j &= \frac{d}{dt} b_{d,i} y_i.\end{aligned}$$

Replace the sums in $\frac{d}{dt}f_i(s,t)$ with the just calculated identity and we find that the extreme values of a Bézier curve and the corresponding $\mathcal{F}_i(P_i, L)$ are at the same parameters:

$$\begin{aligned}\frac{d}{dt} f_i(t_0, s) &= \frac{(-\sum_{j=0,j\neq i}^{d} \frac{d}{dt} b_{d,j}(t_0) y_j) b_{d,i}(t_0) - (-\sum_{j=0,j\neq i}^{d} b_{d,j}(t_0) y_j) \frac{d}{dt} b_{d,i}(t_0)}{b_{d,i}(t_0)^2} \\ &= \frac{\frac{d}{dt} b_{d,i}(t_0) y_i b_{d,i}(t_0) - b_{d,i}(t_0) y_i \frac{d}{dt} b_{d,i}(t_0)}{b_{d,i}(t_0)^2} \\ &= 0.\end{aligned}$$

Finally we have to show what happens if δ is not the first time $\mathcal{C}(c, p_i)$ changes. Then there exists a $\nu \in [0, \delta]$, such that for no $\nu'[0, \nu)$ $\mathcal{C}(c, p_i) \neq \mathcal{C}(\Gamma(p_i + \nu'), p_i + \nu')$. Split the interval $[0, \delta]$ into segments $[\nu_0, \nu_1], ..., [\nu_{n-1}, \nu_n]$, where ν_m is the first time $\mathcal{C}(\Gamma(p_i + \nu_{m-1}), p_i + \nu_{m-1}) \neq \mathcal{C}(\Gamma(p_i + \nu_m), p_i + \nu_m)$ with $m = 1, ..., n-1$ and $\nu_0 = 0$, $\nu_n = \delta$.

□

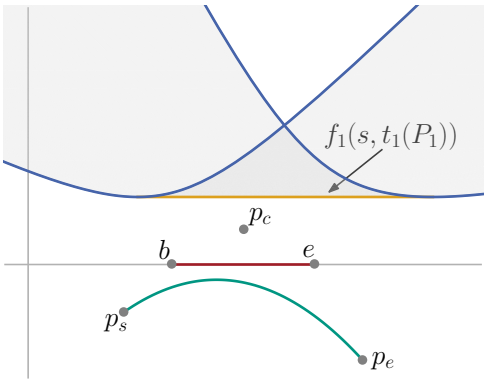
In the case of $d = 2$ we are now able to compute the complete boundary of the forbidden area. We consider only the control point of the curve, leaving the start and end point as variables. The biggest part is already known as it consists of $f_1(t, 0)$ and $f_1(t, 1)$. Remaining is $f_1(t_1(P_1), s)$. With such a small degree we can directly compute the roots of the derivative for $f_1(s, t)$:

$$\begin{aligned}\frac{d}{dt}\left(f_1(t, s)\right)_y &\stackrel{!}{=} 0 \\ \frac{y_s(1-t)^2 - y_e t^2}{2(1-t)^2 t^2} &= 0 \\ y_s(1-t)^2 &= y_e t^2\end{aligned} \tag{3.1}$$

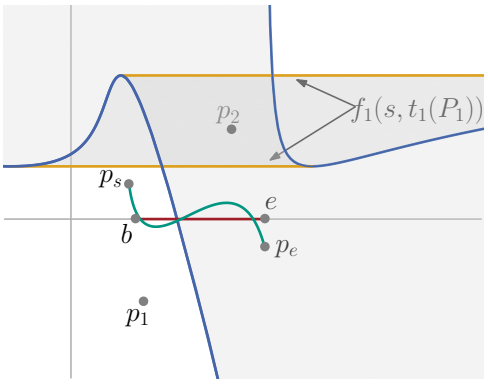
The solution of Equation 3.1 gives us the concrete formula for $t_1(P_1)$. Since we are handling only a quadratic curve, there is always a maximum of one solution, which means for the given positioning of p_s and p_e there exists either one $f_i(t_i(P_i), s)$ or $t_i(P_i)$ has no solution. At this point we make a case differentiation over were we find p_s and p_e.

Case 1: Let p_s and p_e be on the same side of the x-axis. We get three possibilities of how the two points can be positioned relative to each other. With Mathematica we compute the following solution:

$$t_1(P_1) = \begin{cases} \frac{1}{2} & , \text{if } y_s = y_e \\ \frac{y_s}{y_s - y_e} + \sqrt{\frac{y_s y_e}{(y_e - y_s)^2}} & , \text{if } |y_s| < |y_e| \\ \frac{y_s}{y_s - y_e} - \sqrt{\frac{y_s y_e}{(y_e - y_s)^2}} & , \text{if } |y_s| > |y_e| \end{cases}$$



(a) Degree two Bézier curve with $t_1(P_i)$ plotted in orange



(b) Degree three Bézier curve with $t_2(P_i)$ plotted in orange

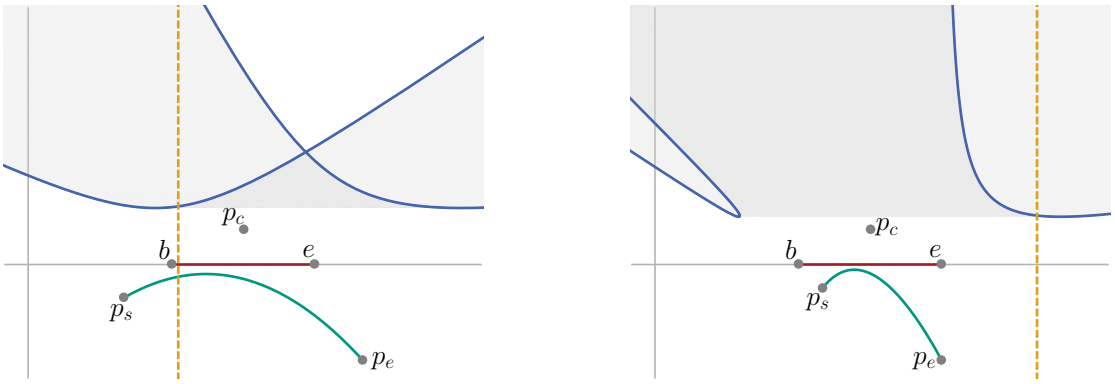
Figure 3.4.: Two examples for $t_i(P)$. For degree 2 (Figure 3.4(a)) the solution is unique and only one $f_1(t_1(P), s)$ exists. For degree 3 there are eventually multiple solutions, see Figure 3.4(b)

Case 2: If p_s and p_e are on different sides of the x-axis, one has a negative and one has a positive y-coordinate. Without loss of generality we assume $y_s > 0$ and $y_e < 0$. Now if we reconsider Equation 3.1 we get:

$$y_s(1-t)^2 = y_e t^2$$
$$\frac{y_s}{y_e} = \frac{t^2}{(1-t)^2}.$$

In the last line the left side is definitely smaller zero, while the right side is bigger zero, because $t \in (0, 1)$. Then there exists no solution for this case.

Figure 3.4 shows a plot, with $f_i(t_i(P_i), s)$ plotted in orange. In the quadratic case, shown in Figure 3.4(a), we get a lower and an upper boundary. The lower one is coming from the left, using $f_1(t, 0)$, at parameter $t = t_1(P_1)$ it is completed by the values of $f_1(P_1, s)$ and then continues to the right with $f_1(t, 1)$. As a lookahead the Figure 3.4(b) shows the plot for degree 3, where the solution is not unique and we get two different $t_i(P_i)$ for the same i. Putting together the boundary can now be done by combining all the boundaries of all $\mathcal{C}(a, p)$. Lets denote the set of all points on the boundary of the forbidden area by $\mathcal{B}(\mathcal{F}_i(P_i, L))$. A point is in $\mathcal{B}(\mathcal{F}_i(P_i, L))$ if and only if it lies on the boundary of one $\mathcal{C}(a, p)$, that is any ε-region around it contains a point q with $\Gamma(q) = 0$. The set of all points on the boundary is really containing two piecewise functions, forming a kind of upper and lower boundary. Distinguishing between them can be done by orienting the involved functions. For the two functions $f_1(t, 0)$ and $f_i(t, 1)$ we orient from big values of t to small values for t. The third function, $f_i(t_i(P_i), s)$ is oriented from small to big values for s. Then we get one set of points, one where the region with crossing number is to the left and one where it is to the right.
In case when p_s and p_e are on different sides of the x-axis we do not need to do any of this. As seen above in the case differentiation we do not get a solution for $t_i(P_i)$ and with Theorem 3.3 we know the boundary consists of $f_1(t, 0)$ and $f_1(t, 1)$.
Up to this point we only looked at the y-coordinate of curve and line. But, looking at the x-coordinate, we find one more differentiation. Figure 3.5 shows two plots for $d = 2$. First Figure 3.5(a) shows the case we already saw beforehand. In Figure 3.5(b) though start and end point moved such that they lie not left and right of b and e, but in between them. The crossing between $f_1(t, 0)$ and $f_1(t, 1)$ disappeared, which is expected, since there can not



(a) p_e is right of the orange line. Functions $f_1(t,0)$ and $f_1(t,1)$ cross.

(b) p_e is left of the orange line. Functions $f_1(t,0)$ and $f_1(t,1)$ do not cross.

Figure 3.5.: The orange line depicts the boundary of two cases. If the end point p_e is right of the orange line we get a crossing between $f_1(t,0)$ and $f_1(t,1)$, if it is left of the line there is none.

be any control point p_c with two crossings between Bézier curve and segment at b and e. Such a point would mean the Bézier curve has to bend outwards in both directions and then come back together, something impossible for a quadratic curve, because the first derivative of the y-component would have three roots. The last differentiation we have to make is distinguishing cases. In case one two boundaries exist, while in case two only one of the boundaries is existing. Using Mathematica we can compute for which values the equation

$$f_1(t,0) = f_1(t,1)$$

has a solution and for which it does not. If there is no solution one side of the forbidden region has no boundary, because if p_s and p_e are on one side there is a non empty crossing region $\mathcal{C}(2,p)$ for some control point $p \in \mathbb{R}^2$ and by Theorem 3.3 this regions boundary is consisting of $f_1(t,0)$, $f_1(t,1)$ and $f_1(t_1(P_1),s)$. There can be at most one intersection between each combination of these three curves, because we are using a quadratic Bézier curve. Now if there is no crossing between $f_1(t,0)$ and $f_1(t,1)$, the connected crossing region $\mathcal{C}(2,p)$ stretches indefinitely far into one direction, while being bounded on the other. Since we are looking at a quadratic curve there is at most one connected crossing region with crossing number two. Solving for x_e we get:

$$x_e > \frac{x_e(x_s - x_b)}{y_s} + x_b + x_c. \tag{3.2}$$

If Equation 3.2 is satisfied, there exists a value t_0 for $f_1(t,0)$. Using the same case differentiation over the y-coordinates of p_s and p_e. Assuming p_s and p_e are on the same side of the x-axis we compute the three cases with Mathematica:

$y_s = y_e$

$$t_0 := \frac{y_e(x_s - x_b) + y_s(-x_e + x_b + x_c)}{2(y_e(x_s - x_b - x_c) + y_s(-x_e + x_b + x_c)}$$

$|y_s| < |y_e|$

$$t_0 := \sqrt{\frac{y_e y_s(y_e(x_s - x_b) + y_s(-x_e + x_b))}{(y_e - y_s)^2(y_e(x_s - x_b - x_c) + y_s(-x_e + x_b + x_c))}} - \frac{y_s}{y_e - y_s}$$

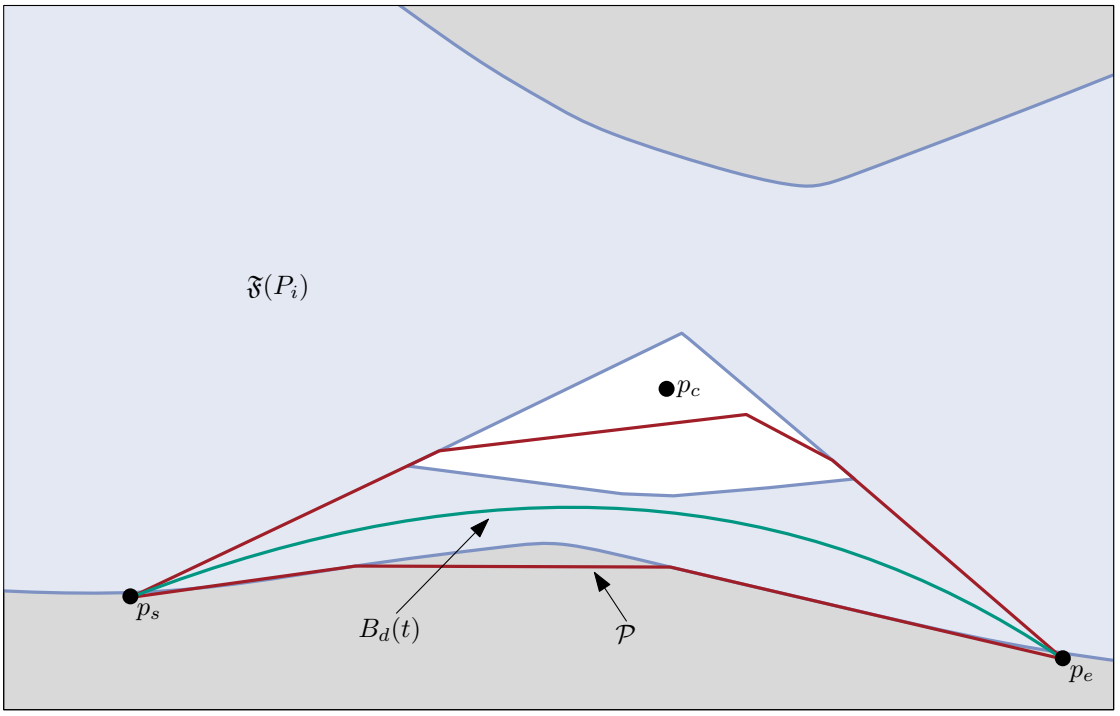


Figure 3.6.: Plot of the forbidden areas of a polygon. For each segment there is an orange area and we are allowed to place the control point anywhere in the white area. Placing it above or below the orange regions would result in no crossing, but the curve would lie completely outside the polygon.

$|y_s| > |y_e|$

$$t_0 := -\sqrt{\frac{y_e y_s(y_e(x_s - x_b) + y_s(-x_e + x_b))}{(y_e - y_s)^2(y_e(x_s - x_b - x_c) + y_s(-x_e + x_b + x_c))}} - \frac{y_s}{y_e - y_s}.$$

The parameter t_1 for $f_1(t, 1)$ is depending on t_0 and with that a solution only exists if the condition for finding a t_0 are met. If t_0 in fact exists we can compute t_1 as:

$$t_1 := \frac{y_s - y_e t_0}{y_s + y_e t_0 - y_s t_0}.$$

Looking back at Figure 3.5 we now know there is no solution for t_0 and t_1 if and only if x_e is left or on the orange line. This last bit concludes the $d = 2$ variant. To sum this section up we saw a general method to find the i-th control point if all but the i-th are given. We now have the whole boundary for $d = 2$. With Theorem 3.3 we can find the boundary of the forbidden area for a general degree, but its complexity increases with the degree of the curve. For the quadratic it should be possible to implement an algorithm that finds p_c, by approximating the boundary of the forbidden area, compute the arrangement and then find the face where the non-forbidden control points lie.

3.3. From Lines to Polygons

After discussing intersections between a single segment and a single Bézier curve we turn back to polygons. Given a start and an end point on the boundary of a simple polygon, find a Bézier curve connecting them, with as few control points as possible such that the curve never intersects with a segment of the polygon. After Section 3.2 the general approach is

to split the polygon into its segments, compute the forbidden regions for each segment and the i-th control point and then intersect them with each other. If there is some point left in the intersection choose it as the i-th control point and return the curve.
Figure 3.6 gives an example for a simple polygon and a quadratic Bézier curve. We can see how the $f_i(s,t)$ intersect and form an area of forbidden points. Still one small problem remains. Above and below the intersection region there are areas where we can place a control point without the Bézier curve crossing any segment of the polygon, but with the curve lying completely outside the polygon.
Formally the area of all forbidden points is the union of the areas $\mathcal{F}_i(P_i, L)$ for each segment:

$$\mathfrak{F}_i(P_i) := \bigcup_{\forall s \in \text{seg}(\mathcal{P})} \mathcal{F}_i(P, s).$$

Computing the whole area is not very practical. What we really want to do is using the boundaries of all the $\mathcal{F}_i(P_i, L)$, intersect them and get the complete boundary of $\mathfrak{F}_i(P_i)$. This can be done just as above. Compute the boundaries, extract all points p with some point q inside an ε-region around p such that $\Gamma(q) = 0$ for any $\mathcal{F}_i(P_i, L)$. While this is a very general description it is surely possible to distinguish the two boundaries in the case of quadratic Bézier curves.

3.4. Degree Three Bézier Curves

The last section in this chapter deals with the case of Bézier curves with degree 3. The main problem we discuss here is finding the roots of the derivative. Curves of higher degree are most likely not solvable, but luckily cubic curves are enough for most applications.
We focus on finding the forbidden second control point, which means we consider p_1 as given and p_2 as the point we want to find. Keep in mind, any plot shown here is again a slice of a body in higher dimensions. With Theorem 3.1 we get the formula for $f_2(s,t)$ with $d = 3$:

$$\begin{aligned} f_2(s,t) &= -\frac{\sum_{j=0, j\neq 2}^{3} b_{j,3}(t) p_j - (b + sc)}{\binom{3}{2} t^2 (1-t)^{3-2}} \\ &= -\frac{(1-t)^3 p_s + 3(1-t)^2 t p_1 + t^3 p_e - (b + se)}{3(1-t)t^2} \end{aligned}$$

By Theorem 3.3 the boundary of $\mathcal{F}_i(P_i, L)$ consist of some segments of $f_2(t,0)$, $f_2(t,1)$ and $f_2(t_2(P), s)$. The main difference between degree 2 and 3 is the possibility of self intersecting Bézier curves, like the one in Figure 3.7, which especially allow the forbidden area to have one hole. For the general description this makes no difference, but the implementation gets a little more tricky.

The main thing we concern ourself with is $t_2(P)$. To compute it we need the roots of the first derivative of $(f_2(s,t))_y$:

$$\begin{aligned} \frac{d}{dt} \left(f_2(s,t)\right)_y &\stackrel{!}{=} 0 \\ \frac{2(1-t)^3 y_s + 3(1-t)^2 t y_1 - t^3 y_e}{3(1-t)^2 t^3} &= 0 \\ 2(1-t)^3 y_s + 3(1-t)^2 t y_1 &= t^3 y_e \end{aligned}$$

Again we use Mathematica to solve this Equation. The solution is more complex than in case of $d = 2$. The case differentiation gets bigger. Here we split it up into several decision trees. First we have to differentiate between the relative positions of p_s and p_e:

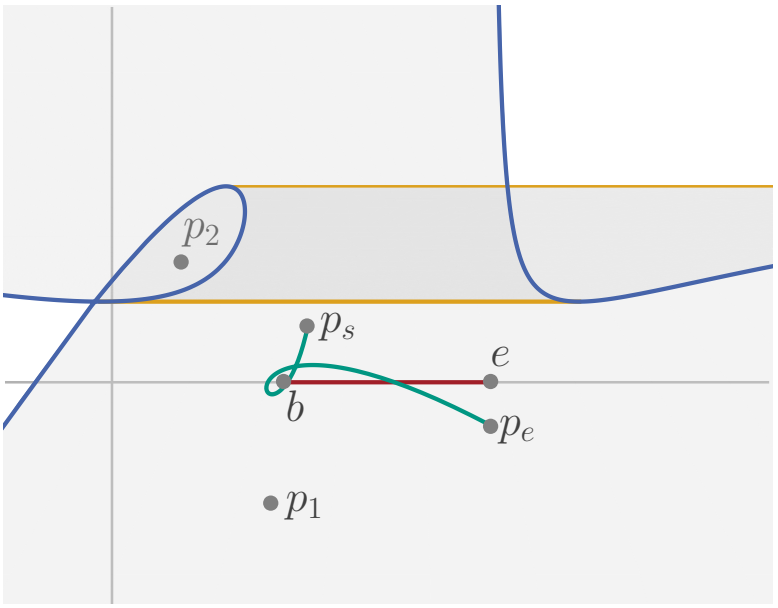
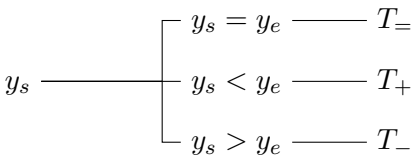
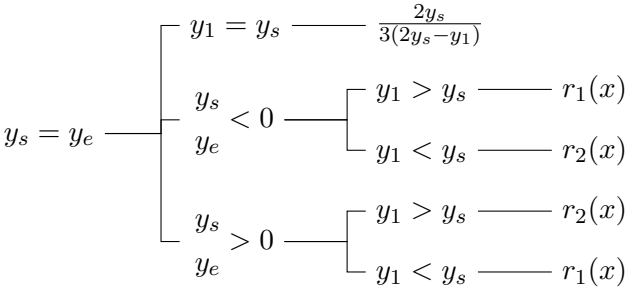


Figure 3.7.: Example of a self-intersecting degree 3 Bézier curve.



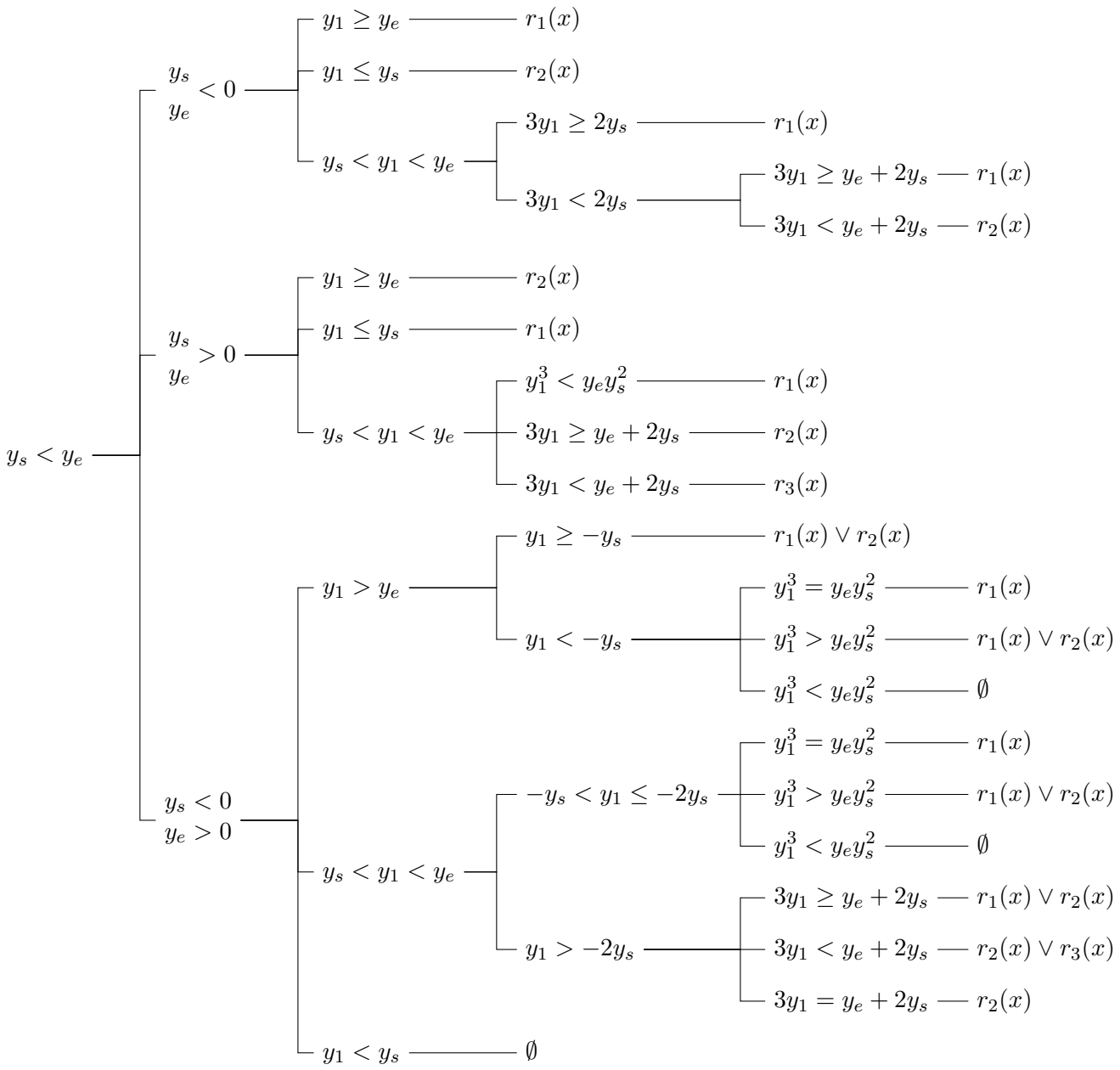
The T's are again trees. The smallest is $T_=$. In it we differentiate the side where start and end point are and the position of the p_1 relative to $p_s = p_e$:



The first leaf of this tree is clear, but the others need some explanation. Instead of a closed formula we denote with $r_i(x)$ the i-th solution for $x \in \mathbb{R}$ of:

$$r_i(x) := 2p_s(-1+x)^3 + x(-3p_1(-1+x)^2 + p_e x^2) \stackrel{!}{=} 0$$

While the first solution is real, the second and third might be complex. So far it was not possible to calculate for which $x \in \mathbb{R}$ this is the case. Finally the two trees T_+ and T_- have the same structure, but in T_- all the comparison operators have to be flipped. The differentiation is quite big and uses not really intuitive bounds. Especially noteworthy is the possibility of multiple solutions, which we already saw in Figure 3.4(b). Here we show only T_+. Computing the full formulas can be done using the Mathematica code provided in the appendix.



If we know where all the loops are on the boundary curves we can exclude those parts, by jumping over them. Finding the whole boundary can now be done just as above. Look at $f_2(t, 0)$, $f_2(t, 1)$ and $f_2(t_2(P), s)$ and pick any point which has a point with crossing number zero in any ε-region around it.

4. Sampling with ε-Nets

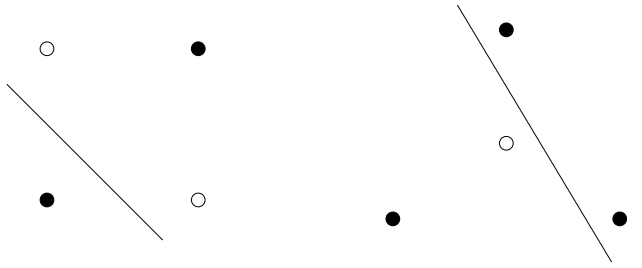
In the previous chapter we saw how one can choose a control point for a Bézier curve, such that the curve does not intersect a given polygon. But so far we have to choose the other $d-1$ control points randomly. This does not seem to be of much practical use, because choosing good control points for the whole polygon by chance seems very unlikely. This chapter investigates the idea of sampling all control points at once using ε-nets and their relation to VC-Dimension.

Exploring a more complex sampling method is necessary, because a naive grid based approach proves to be impractical. Suppose we had a bounding box around the allowed area, then sampling a grid of points, with granularity $\varepsilon \in [0,1]$ depends on the diameter of said bounding box. A set of points sampled this way contains $O\left((\Delta/\varepsilon)^{2d-2}\right)$ sample points. If ε-nets are applicable we do not longer depend on the size of the used bounding polygon. To understand how ε-nets work in general we look at an example, exploiting the concept of VC-Dimension and its relation to sampling, before getting back to our original problem.

Assume we want to solve the following problem. Given a disk C with area one, find a sample of points inside C such that if we consider another disk C', which intersects C and the area of the intersection is big enough, we find a point in the sample set lying in C and C'. The two main gaps we need to fill are what big enough intersection means and by what criteria we choose the sample. Both questions can be answered with the ε-net theorem, which we will introduce in Section 4.1.

An ε-net is a set N of points, chosen from a so called ground set X. In the above case C is the set we want to choose from. Restricting ourselves to C is done using a probability measure μ. Let X be the $\mathbb{R}^2$, then the probability measure is zero for any set outside the disk C and one for the whole disk C. The disks we intersect with C are collected in a set T, that is T is a family of sets where every element of T is a set containing the points of one disk C'. Now N is an ε-net if and only if for every C', with $\mu(C') > \varepsilon$ and $\varepsilon \in [0,1]$, we get a non empty intersection between C' and N. In other words for every C' intersecting C and the area of the intersection being big enough if measured by μ, we find a $p \in N$ such that $p \in C'$ and $p \in C$.

Finding such a net seems not an easy task. Surprisingly the ε-net theorem guarantees the existence of an ε-net for a lot of different sets. The main ingredient needed to prove the theorem is the notion of *bounded VC-Dimension*. We introduce the concept in the following section. Afterwards, in Section 4.2, we note two lemmas giving us the ability to bound the VC-Dimension for a broad set of geometric shapes and the last part, Section 4.3, draws a



(a) Convex position.

(b) Non-Convex position.

Figure 4.1.: The two different possibilities to place four points in the plane.

connection between ε-nets and our basic problem of finding the control points to a degree d Bézier curve.

4.1. VC-Dimension

To define the VC-Dimension, let $A \subseteq X = \mathbb{R}^2$ be a set of points in general position in the plane. Further let $A = A_1 \cup A_2$ be a partition of A. We say a half plane H supports this partition, if $A_1 \subseteq H$ and $A_2 \cap H = \emptyset$. Choosing T as the family of sets where every set is a half plane, we say T *shatters* A if and only if every subset $A' \subseteq A$ is supported by a $H \in T$. The VC-Dimension of a family of sets T is defined as the biggest A still shattered by T.
If T contains all half planes in the $\mathbb{R}^2$ we can find the VC-Dimension by looking at a set of four points. Figure 4.1 shows the possible positions of four points in the plane. In both cases we can not separate the white disks from the black ones with a half plane. Looking at three points in general position, we can always shatter any subset of those three points with a half plane. This tells us the VC-Dimension of T is three and we write $\dim(T) = 3$.
In general we say the VC-Dimension $\dim(T)$ is bounded if and only if there is a $v \in \mathbb{N}_+$, such that $\dim(T) \leq v$. Otherwise we say the VC-Dimension is unbounded.
The above notion of half planes can be generalized to degree d. Radon's theorem says, any set A of $d+2$ points in $\mathbb{R}^d$ can be partitioned into two disjoint sets A_1 and A_2, such that the intersection of their convex hulls is non-empty. Applying this to the system of points and half planes no set of size $d+2$ or bigger can be shattered. Else there would be a half plane $H \in T$, such that for any partition $A = A_1 \cup A_2$, with $A_1 \subseteq H$ and $A_2 \cap H = \emptyset$, the convex hulls of A_1 and A_2 do not intersect, a contradiction to Radon's theorem. Contrary a set of $d+1$ points in general position can be shattered. This gives us a bound for the VC-Dimension of the set T of all half planes in $\mathbb{R}^d$ with $\dim(T) = d+1$.
The VC-Dimension of more complex objects than half-planes is often hard to bound if done by foot. In Section 4.2 we present two lemmas that give us great tools to bound the VC-Dimension of sets described by one polynomial inequality or are composed of sets with bounded VC-Dimension.

4.2. ε-Nets and bounded VC-Dimension

The connection between VC-Dimension and ε-nets is expressed in the so called ε-net theorem. The idea is that we can guarantee the existence of an ε-net of small size, for any set with bounded VC-Dimension.

Theorem 4.1. *Let X be a set with probability measure μ and T a family of μ-measurable subsets of X with bounded VC-Dimension $\dim(T) \leq v$. Then there exists an $\varepsilon = \frac{1}{r}$-net with $r \geq 2$ and size at most $O(vr \ln r)$.*

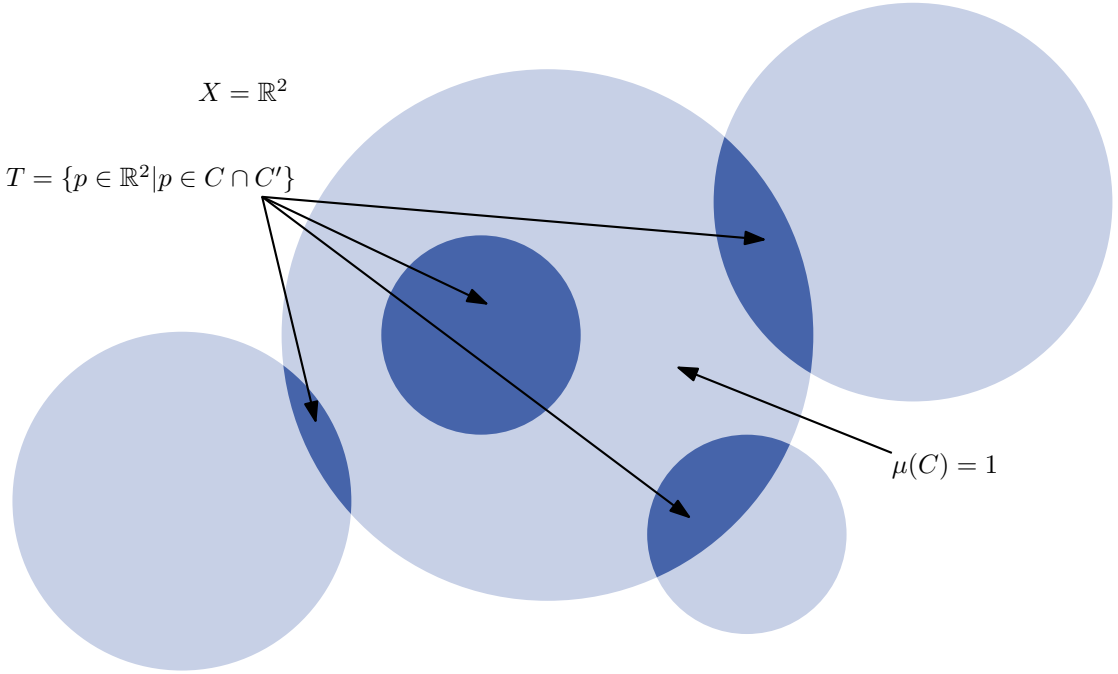


Figure 4.2.: System of intersecting circles. The dark blue areas are the ones we want to hit with points in the ε-net. The light blue area is the ground set.

A proof of Theorem 4.1 can be found in the book "Lectures on Discrete Geometry" by Jiří Matoušek, see [Mat02, Chapter-10]. From the proof a basic method to find an ε-net with high probability can be derived. Let $s = O(vr \ln r)$, we make s independent draws from X, picking each element of X according to μ. A sample N that we drew in this manner, is an ε-net with probability greater $1/2$.
Coming back to the example of intersecting disks. Recall that C is a disk in the plane intersected by other disks C'. We choose X as the plane and set $\mu(C) = 1$. The elements of T represent the other disks C'. Figure 4.2 shows what an instance of this problem might look like. Now we want to find an ε-net N, that is a set of points where we find for every disk C' with $\mu(C') > \varepsilon$ a point inside N. Theorem 4.1 gives us exactly such a sample, if we are able to bound the VC-Dimension of disks. This can be accomplished using the following lemma.

Lemma 4.2. *Let $\mathbb{R}[x_1, x_2, ..., x_d]$ be the set of all real polynomials in $d \in \mathbb{N}_+$ variables with maximum degree $D \in \mathbb{N}_+$, then a sets of the form*

$$P_{d,D} := \{\{x \in \mathbb{R}^d : p(x) \geq 0\} : p \in \mathbb{R}[x_1, x_2, ..., x_d]\}$$

has bounded VC-Dimension of $dim(P_{d,D}) \leq \binom{d+D}{d}$.

This lemma can be proven very elegantly, using a mapping to the monomials, which reduces the case of general polynomials to half-planes, again see the proof can be found in the book of Matoušek [Mat02, Chapter-10]. Lemma 4.2 gives us a bound for the VC-Dimension of disks. We can describe the unit disk in the plane with:

$$x^2 + y^2 - z \leq 0.$$

This is a polynomial inequality in three variables with a maximum degree of two. Applying the formula from Lemma 4.2 we get an upper bound for the VC-Dimension of the set T containing all disks in the $\mathbb{R}^2$:

$$\dim(T) \leq \binom{3+2}{3} = 10.$$

This bound is not tight and can be lowered significantly. A more careful analysis yields a VC-Dimension of five, by using a better mapping and not map to all combinations of monomials. The other lemma, which will be interesting for us later on, bounds the VC-Dimension of a set composed of sets with bounded VC-Dimension.

Lemma 4.3. *Let S be a set system with VC-Dimension $dim(S) = d < \infty$ on a ground set X and with $G(S_1, S_2, ..., S_k)$ we denote a set-theoretic expression using union, intersection and difference. Then the set*

$$\mathcal{T} := \{G(S_1, S_2, ..., S_k) : S_1, ..., S_k \in S\}$$

has bounded VC-Dimension of $dim(\mathcal{T}) = O(kd \log k)$.

Using Lemma 4.2 and 4.3 we can bound the VC-Dimension for any geometric shape describable by finite number of polynomial equations.

4.3. Using ε-Nets to Find Control Points

The ε-net theorem offers us a way to prove that small samples are enough to find all the control points of a Bézier curve. First we think about what the sets (X, T) are going to be. With X we identify the $\mathbb{R}^{2d-2}$. When we pick a point in $2d-2$ dimensions, we can interpret it as a vector, containing all $d-1$ inner control points. The set T contains all allowed points, that are all control points not leading to an intersection between the Bézier curve and the polygon. Since we are working in $2d-2$ dimensions every point we pick can be seen as a vector, encoding the $d-1$ inner control points of our curve and consequently we can interpret X as the set of all curves connecting a fixed start and end point.
Finally the probability measure has to be restricted to a certain region. This can be done by asking ourselves what the minimum distance is, a control point has to be pulled outwards, such that curve and bounding box have to intersect. We then choose a box in the $\mathbb{R}^{2d-2}$ containing the allowed area and the polygon and set the probability measure μ to 1 on this bounding box.
To illustrate the above have a look at Figure 4.3. For a quadratic curve the space we work with is the $\mathbb{R}^2$, because we only can pick one control point. All allowed control points lie in the white area, finding a diameter a we can restrict the probability measure to a rectangle around the allowed area and polygon containing p_s and p_e. If we are dealing with curves of degree d this figure shows only a slice through the space, where we fixed all but one control points. The following theorem connects the 1-curve path problem with ε-nets. Unfortunately we did not manage to prove the full theorem in the scope of this thesis.

Theorem 4.4. *Let X be the $\mathbb{R}^{2d-2}$, T be the family of all sets of allowed points with $dim(T) \leq v, v \in \mathbb{N}_+$ and μ is a probability measure on X. Then there exists an ε-net of size $O(v \cot r \log r)$ with $\varepsilon = \frac{1}{r}$ and $r \geq 2$.*

Remaining are two tasks, first we have to find the described bounding box around the allowed area and secondly we have to show that the VC-Dimension of the chosen T is bounded. For us this says we have to show the allowed area has a bounded VC-Dimension.

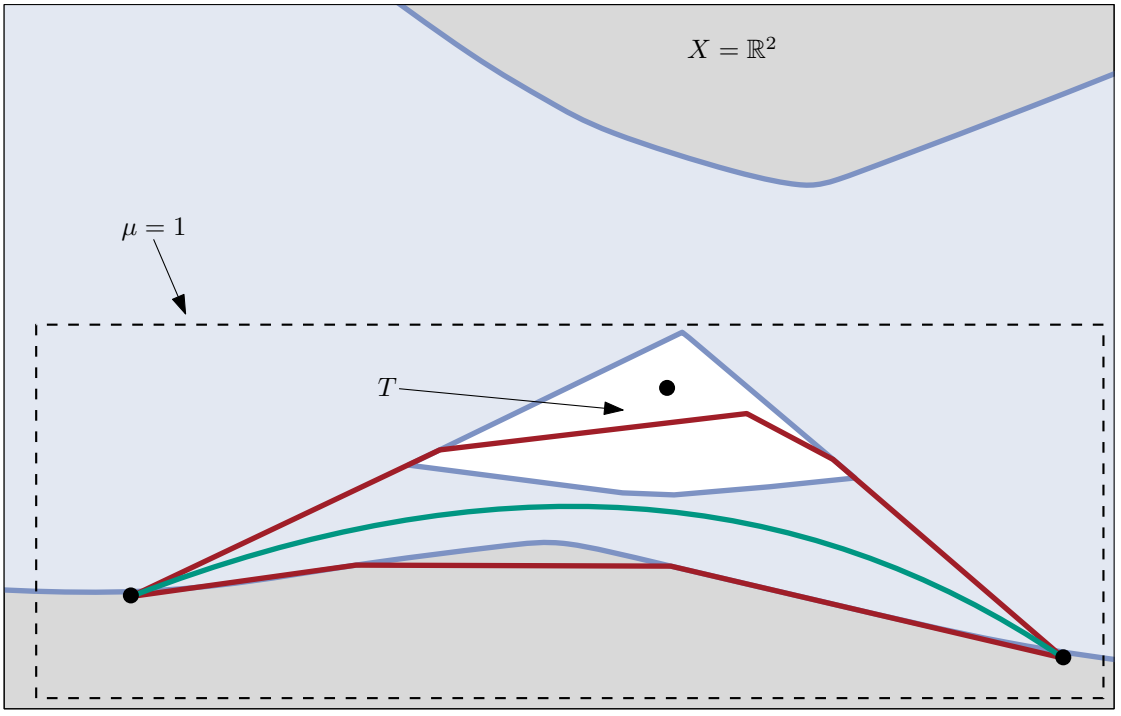


Figure 4.3.: Illustration of the ground set X and set system T in our configuration. The dashed rectangle is part of the bounding box and the white area is the one we want to hit. With blue we denote the forbidden area, while the grey area is not forbidden by the $f_i(s,t)$ but it is excluded, because the whole curve is outside $\mathcal{P}$ it the control point is placed there.

Finding the Ground Set

Let $B_d(t)$ be a Bézier curve with control points $P := \{p_0, ..., p_d\}$, we assume without loss of generality $p_0 = (0,0)$ and $p_d = (x,0), x \neq 0$. We consider the polygon $\mathcal{P}$, with p_0 and p_d being points inside or on the boundary of $\mathcal{P}$. For any two points $u, v \in \mathbb{R}^2$ we say $\text{dist}(u,v)$ is the distance between them. The maximum $\text{dist}(u,v)$ between any two points $u, v \in \mathcal{P}$ is called Δ.
Observe that for any point $p = B_d(t)$ the distance between p_0 and p is at most Δ. In other words any set of control points leading to $\text{dist}(B_d(t), p_0) > \Delta$ is definitely outside the region we want to use as bounding box since it leads to a crossing. Let $t' \in (0,1)$ be some parameter and $u' = B_d(t')$ be the corresponding point on the curve. Suppose the curve intersects $\mathcal{P}$, we want to show either $\text{dist}(u', p_0) > \Delta$ or $\text{dist}(u', p_d) > \Delta$, if a certain control point is pulled out far enough.
For every point on the curve some control point p_i has the biggest influence on it, which just says that the value of the Bernstein polynomial is bigger for this control point than for any other control point. If p_i has the biggest influence this translates to

$$\forall_{j \in \{0,\dots,d\} \setminus i} \; b_{d,i}(t') \; > \; b_{d,j}(t').$$

The case where we can place the control point the farthest away, is exactly occurring if p_i pulls into one direction and all other control points pull the curve exactly in the opposite direction.
This means we are looking for a constant $a \in \mathbb{R}$, such that a control point being pulled out by more than $a\Delta$ forces an intersection between $B_d(t)$ and $\mathcal{P}$. If we find such an a, we place a bounding box around the polygon with side length depending on a and Δ. This means we only need to look at control points being pulled out along the x- or y-axis and

can set the other coordinate to zero. Further the calculation yields the same result for both directions and we only need to consider the case of the y-coordinate being set to zero. Setting the i-th inner control point to $(-a\Delta, 0)$ and all other inner control points to $(a\Delta, 0)$ is enough. We can use the same a for both control points, because if one is pulled out more another part of $B_d(t)$ will intersect the boundary. Start and end point are placed differently. By definition p_s has to lie at $(0, 0)$ and since Δ is the diameter of the polygon the worst case placement of the end point would be at $(\Delta, 0)$, else it surely lies outside $\mathcal{P}$.

Lemma 4.5. *Let $\mathcal{P}$ be a polygon in the $\mathbb{R}^2$ with diameter Δ and $B_d(t)$ be a Bézier curve of degree d with start p_0 and end point p_d laying inside $\mathcal{P}$ and inner control points $\{p_1, ..., p_{d-1}\}$. Then there exists an $a \in \mathbb{R}$, such that $B_d(t)$ intersects the boundary of $\mathcal{P}$ if a control point p_i is placed at (x, y) with $|x| > |a\Delta|$ or $|y| > |a\Delta|$.*

Proof. Let $u' \in \mathbb{R}^2$ be a point on the Bézier curve. Then there exists a parameter $t' \in [0, 1]$, such that $B_d(t') = u'$. $B_d(t)$ intersects the $\mathcal{P}$ if $\text{dist}(p_0, u') > \Delta$ or $\text{dist}(p_0, u') > \Delta$. We only consider the distance between u' and p_0 the case of p_d can be done analogously. Since all the y-coordinates of the control points are zero the distance $\text{dist}(u', p_0)$ is $|\sum_{j=0}^{d} b_{d,j}(t')x_j - (0, 0)|$. Replacing the x-coordinates with the worst case values from above we get:

$$\left|\sum_{j=0}^{d} b_{d,j}(t')x_j\right| \geq \left|\left(\left(\sum_{j=1,j\neq i}^{d-1} b_{d,j}(t') - b_{d,i}(t')\right)a + b_{d,d}(t')\right)\Delta\right|.$$

We find ourselves in one of three cases. First if the sum and $b_{d,i}(t')$ are equal, then the equation is reduced to just $b_{d,d}(t')\Delta$. Seemingly we get a contradiction to our assumption of the distance between p_0 and u' or p_d and u' being big. Luckily this case is not important, because no control point beside the last and first play any role in it, which translates to u' laying on a straight line between p_0 and p_d. Such a line is definitely in any bounding box that surrounds the polygon.
The other two cases are either the sum is bigger as $b_{d,i}(t')$ or smaller. Without loss of generality we assume the later:

$$\begin{aligned}&\left|\left(\left(\sum_{j=1,j\neq i}^{d-1} b_{d,j}(t') - b_{d,i}(t')\right)a + b_{d,d}(t')\right)\Delta\right| \\ =&\left(\left(b_{d,i}(t') - \sum_{j=1,j\neq i}^{d-1} b_{d,j}(t')\right)a - b_{d,d}(t')\right)\Delta.\end{aligned}$$

We want to show there is a value a such that the distance is bigger than Δ:

$$\begin{aligned}\left(\left(b_{d,i}(t') - \sum_{j=1,j\neq i}^{d-1} b_{d,j}(t')\right)a - b_{d,d}(t')\right)\Delta > \Delta \\ \left(b_{d,i}(t') - \sum_{j=1,j\neq i}^{d-1} b_{d,j}(t')\right)a - b_{d,d}(t') > 1 \\ \frac{1 + b_{d,d}(t')}{b_{d,i}(t') - \sum_{j=1,j\neq i}^{d-1} b_{d,j}(t')} < a.\end{aligned}$$

Solving this equation for a specific d yields the result. □

In most cases degree Bézier curves are enough. Using the above equation we can compute a and t' using Mathematica as $a \approx 3.4305$ for $t' \approx 0.215$. Here we used the first control point. Using the second control point changes only t' and we end up with $a \approx 3.4305$ for

$t' \approx 1 - 0.215$. Finally we restrict the ground set to a box around the polygon with side length $(2 \cdot 3.4305 + 1)\Delta$ in all direction of the $\mathbb{R}^4$.

Bounding the VC-Dimension

The other important condition we need to verify before the ε-net theorem can be used is that the VC-Dimension of the set system T is bounded. In our setting T contains the allowed points. The immediate idea is to bound the VC-Dimension of the set of all forbidden control points:

$$A := \{p \in X | p \notin \mathfrak{F}_i(P_i)\}.$$

Unfortunately this direct approach only works if we can convert the $f_i(s,t)$ into implicit equations. If we just continue the decomposition we see why. Using the definition of the $\mathfrak{F}_i(P_i)$ we get the condition for a point to be inside or outside a forbidden area. It can be expressed as a polynomial equation. Let p_i be the i-th control point:

$$A = \{p \in X | \forall l \in \text{seg}(\mathcal{P}) \forall t \in (0,1) \forall s \in [0,1] : f_i(s,t,p,l) - p_i \neq 0\}.$$

This set can be split up again if we express the $\neq$ as the union of the set with all points greater zero

$$A_> := \{p \in X | \forall l \in \text{seg}(\mathcal{P}) \forall t \in (0,1) \forall s \in [0,1] : f_i(s,t,p,l) - p_i > 0\}$$

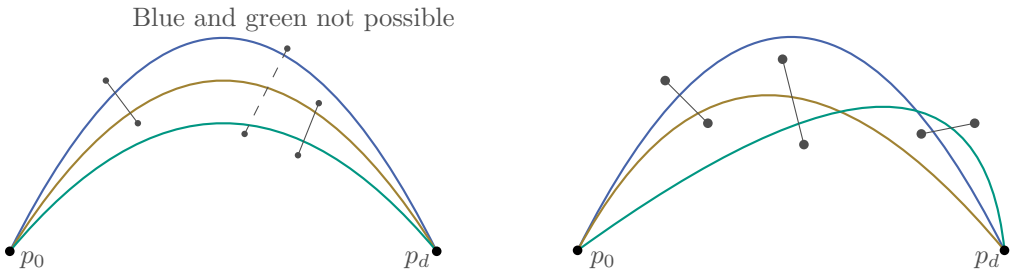
and smaller zero:

$$A_< := \{p \in X | \forall l \in \text{seg}(\mathcal{P}) \forall t \in (0,1) \forall s \in [0,1] : f_i(s,t,p,l) - p_i < 0\}.$$

The union of these sets is again the set of all points not in a forbidden area A. Both, $A_<$ and $A_>$, can be split up again as the intersection of all the allowed points over the segments:

$$A_> = \bigcap_{l \in \text{seg}(\mathcal{P})} \{p \in X | \forall t \in (0,1) \forall s \in [0,1] : f_i(s,t,p,l) - p_i > 0\}$$
$$A_< = \bigcap_{l \in \text{seg}(\mathcal{P})} \{p \in X | \forall t \in (0,1) \forall s \in [0,1] : f_i(s,t,p,l) - p_i < 0\}.$$

At this point we want to apply Lemma 4.2, but then we fix the parameters where the crossing is located. This translates to the VC-Dimension of points leading to a crossing at $t = t_0$ and $s = s_0$ being bounded. If we want to get back the whole set we have to quantify over all parameters, which does not any longer fit with Lemma 4.3, because we need a union over infinitely many sets.
To make the approach work we have to cut the number of curves we are considering. The first step is to only look at the curves on the boundary, given from Theorem 3.3. Applying Lemma 4.2 we can build the allowed space by set operations on the boundary curves.
Each of the curve now only depends on one parameter and unfortunately on a set of given control points. Bringing them to the other side does not solve the problem as well, because then we map a parameter to an implicit equation. Before we can implicitize the boundary curves we have to parametrize the equation depending on all the control points. Converting a rational parametric one to an implicit equation is always possible [SAG84, SG86], finding a parametric equation though is a more difficult task.
In the case of quadratic Bézier curves we can demonstrate this approach, because we only have one inner control point and do not have to fix any more than start and end point of the curve. The boundary we are interested in consists of $f_1(s,t)$ for $s = 0$, $s = 1$ and



(a) No possbility to hit every set of curves.

(b) It is possible to hit every combination.

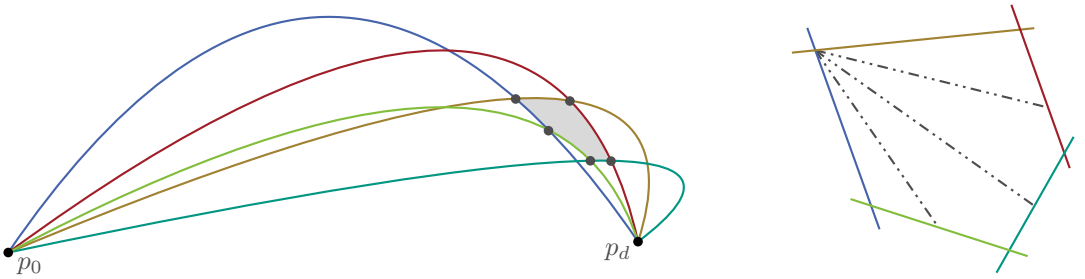
Figure 4.4.: For three curves it is possible to arrange them in a way, such that every combination of them can be intersected by a segment. This is equivalent to shattering three control points.

$t = t_1(P_1)$. Solving the $f_1(s,t)$ for t and s respectively, we get implicit equations in two variables, the coordinates of the control points, and maximal degree 2. Now we are able to apply Lemma 4.2 and get the VC-Dimension for each of the boundary curves as $\binom{2+2}{2} = 6$. We know from Section 3, there can be only crossing regions with crossing number one and two, and they can be created from the three boundary curves with union and intersection. This allows us to apply Lemma 4.3. The biggest set equation can use all three boundary curves, resulting in a bounded VC-Dimension of $O(6 \cdot 3 \log 3) \approx 18$.

Translating the Problem to Colouring

The ideas described above seem hard to translate to a higher degree. Another view on the problem of bounding the VC-Dimension of the allowed area is necessary. If we identify with X all points in the $\mathbb{R}^{2d-2}$, then each point represents a curve between the fixed points p_0 and p_d. The sets we put into T are the sets of forbidden points. Choosing a subset $C \subseteq X$ is equivalent to picking a set of curves in the plane. We shatter the set C with T if and only if we find for every subset $C' \subseteq C$ an $S \in T$ such that $C' = S \cap C$.

Interpreting this geometrically we pick a set of curves, draw them according to their control points and then try to intersect every combination of them with a segment. A set of control points is then shattered by T if and only if for every combination of curves we find a segment, such that it intersects this combination. Figure 4.4 shows an example with three curves. While it is not possible to intersect the blue and green curve in Figure 4.4(a),



(a) Arrangement of five curves with a pentagon.

(b) Choosing every combinaion of three colours in the pentagon.

Figure 4.5.: In the case of a set of five quadratic curves we can arrange them in a way, such that a pentagon is one of the faces. Every corner has two colours, while the other three edges of the pentagon have the other three colours.

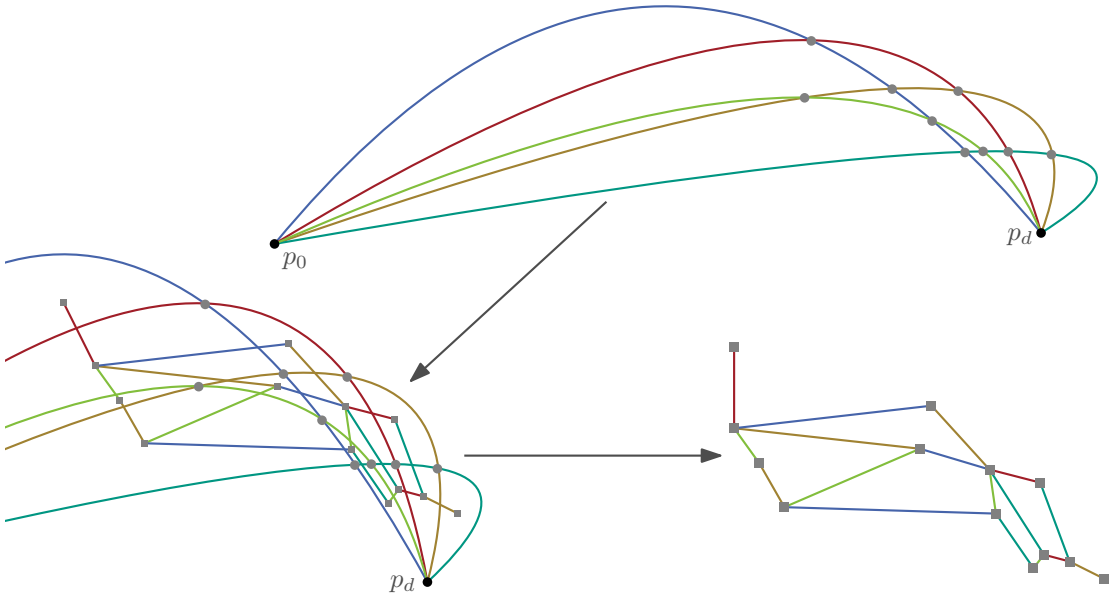


Figure 4.6.: Converting the drawing of curves into a graph, where the problem of finding a segment, such that it intersect a combination of curves is reduced to finding a path with these colours.

we can rearrange the curves in a way allowing us to intersect every possible combination (see Figure 4.4(b)).
Instead of bounding the allowed area we now bound the VC-Dimension of the forbidden points. This is no problem, because Lemma 4.3 can be used to transform it back into the allowed area. To do this we take the difference between the set of all curves or control points and the forbidden area.
Earlier in this section we concluded an upper bound for the VC-Dimension of quadratic curves is 18. A lower bound, that is a set we are still able to shatter, can immediately be derived from the presented conversion. Figure 4.5 shows an arrangement of five curves, such that we can intersect any combination of curves with a segment. This can be seen very well if we consider the pentagon, which we form with the curves. It basically allows us to pick every set of three colours, which is the only problematic size of curve combinations. This means the VC-Dimension of the forbidden area is at least 6 and at most 18.
Taking the colouring approach a step further, we can interpret the intersections of curves as nodes V and the segments between them as edges E. The resulting graph $G = (V, E)$ is planar and the drawing of the curves gives us an embedding. Figure 4.6 shows the process. Every edge gets assigned a colour, depending on the colour of the curve segment it represents. From G we then compute the dual graph $G^\star$ and assign every edge in $E^\star$ the colour of its corresponding primal edge. The only problem we have in this construction is the possibility of parallel edges. Suppose there are two parallel edges in the graph. From the drawing of the curves we derived an embedding, which is represented by the cyclic order of edges around the nodes. For every two parallel edges, next to each other in the cyclic order, we introduce a node to the dual graph and connect it to the two adjacent faces or the nodes of more parallel edges. The colours are derived from the cyclic order of the edges around the node.

Looking at the dual graph we can rephrase the problem of intersecting all possible combinations of curves. Given a set of colours Ω we want to find a path in $G^\star$ containing $|\Omega|$ edges and for every colour $\omega \in \Omega$ the path contains an edge with colour ω. So far we did not manage to find a arrangement for six curves, such that we can intersect every

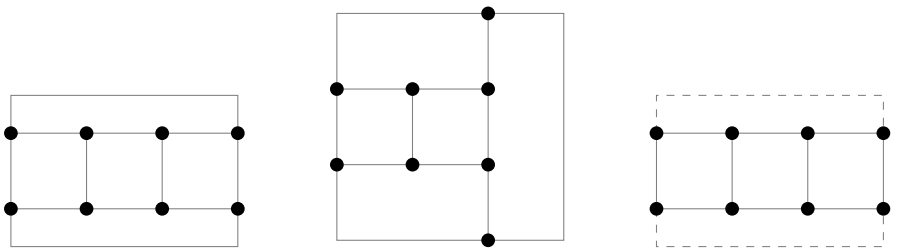
combination of curves and to us it seems likely that the VC-Dimension of the forbidden area is six.
Bounding the VC-Dimension for a general degree d needs more work. Nonetheless it seems intuitive that the VC-Dimension of the allowed area has in fact a bounded VC-Dimension and if proven we can directly derive the existence of small ε-nets.

Conjecture 4.6. *Let X be the set of all Bézier curves of degree d, defined by fixed start point p_0 and end point p_d and $d-1$ inner control points. Let T be the family of sets, where every set is a set of forbidden control points. It exists a $v \in \mathbb{N}_+$, such that $dim(T) \leq v$.*

Using Lemma 4.5 and Conjecture 4.6 we can prove Theorem 4.4. The resulting ε-nets then have size $O(v' \cdot k \log k)$, with v' being the VC-Dimension of an allowed area for one Bézier curve and one segment and k being the number of segments in the polygon.

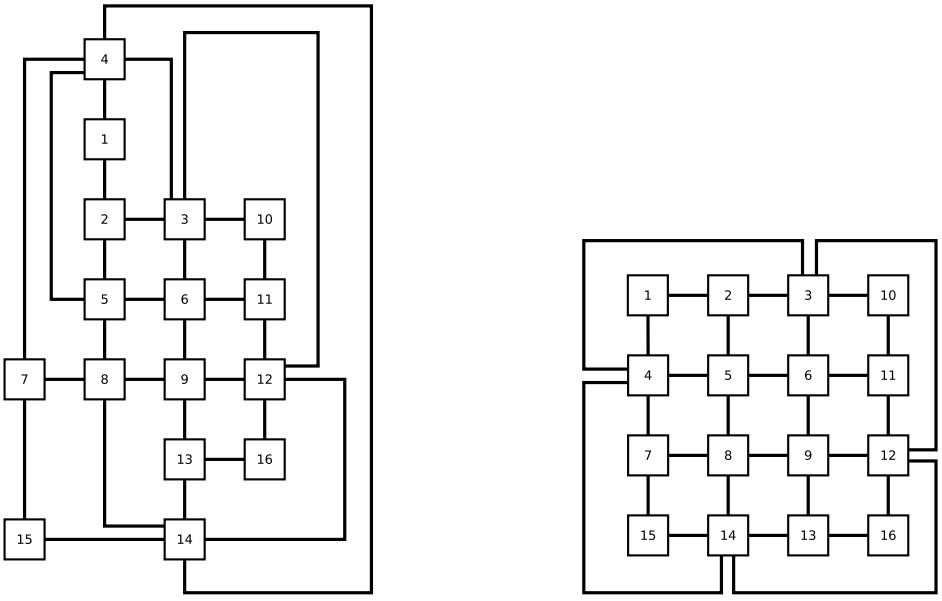
5. A Graph Drawing Application

Look at the graph in Figure 5.1. The leftmost drawing looks pretty nice. All nodes are in a grid and two edges are going around the graph. If we let a standard orthogonal layouter draw the graph, we get the result in the middle. The nodes are no longer in a grid, which leads to a less well visible structure. This observation leads to the idea we pursue in this chapter.
Figure 5.1(c) shows the same graph again, but compared to the drawing in Figure 5.1(a) two edges, the ones going around the graph, were deleted. The remaining part is a simple grid and using the same orthogonal layout algorithm as in Figure 5.1(b), we get a drawing of the remaining part showing exactly that. Finally we reinsert the edges. This produces a drawing that shows the underlying structure of the graph better, than if we use the layout algorithm on the whole graph in one sweep.
In Section 5.1 we discuss the question, which edges have to be removed, such that if we draw the remaining graph the structure becomes better visible. Afterwards, in Section 5.2 we investigate how the removed edges can be inserted into the already constructed drawing. As a running example we use the graph shown in Figure 5.2(a). This initial layout shown was computed with yEd [yWo15] and can still be improved substantially, when we compare it to the hand-drawn variant in Figure 5.2(b). If drawn optimally the example graph is just a grid with a few edges going around the outer face. All the drawings of the running example were generated by a self developed tool, using the "Open Graph Drawing Framework" OGDF [CGJ+11] and yEd.



(a) Nice Drawing (b) Ugly Drawing (c) Delted the Outer Edges

Figure 5.1.: Depiction of the basic idea. If drawn automatically the drawing is distorting the grid structure. Deleting the edges and reinserting them can help.



(a) Orthogonal Drawing by yEd.

(b) Drawing of the graph by hand in yEd.

Figure 5.2.: The initial layout of our continuous example drawn by yEd. Figure 5.2(a) shows a graph drawn by the orthogonal layouter of yEd. The other drawing, Figure 5.2(b) was drawn by hand.

5.1. Deleting Edges

Given a planar, connected graph $G = (V, E)$ we want to find edges $E_d \subseteq E$ such that, after we delete them, the remaining graph is nicely drawable by a given layout algorithm. We call the graph, remaining after we delete the edges, the core graph $G_c = (V_c, E_c)$, with $V_c := V$ and $E_c := E \backslash E_d$.
When finding edges for E_d we have to make sure deleting it does not destroy a lot of structure, at the same time we don't want to choose too few edges, such that there is nothing won and the layout still looks bad. One metric we can track and which is important in any case is the connectivity of G. By assumption G is at least 1-edge-connected and we want to preserve this. In case we disconnected G there would be at least two independent components and each of them has to be drawn independently. In the end the underlying structure is most likely lost completely. This means the minimal thing we want to guarantee, is to keep G connected. The algorithm we present here achieves even more, by keeping the graph 2-edge-connected, if it was at least 2-edge-connected in the beginning and otherwise does not construct any new *bridges*. A bridge is an edge whose removal disconnects the graph. So any graph with a bridge is 1-edge-connected.

A motivation to what kind of edges we want to delete can be given by looking at the edges we just saw in the examples. In both, the small and the big, examples we found edges which forced the drawing to bend the graph. This can be seen as a general problem. Suppose we have a rather dense graph, but it can be drawn very regularly, an example would be a grid with diagonals in the cells. If there are edges between nodes on the outer face a layouter will be forced to incorporate them. For example with orthogonal layouts we minimize the bends on the edges and if we can distribute the bends better among the edges by bending the whole graph we do this instead of a few more bends on the edges.
Using this motivation we can look at this a little differently. If we stand in the outer face f_o of the graph and pick a primal edge e, adjacent to f_o and another face f_i, and delete it, we merge f_o with f_i. Continuing this process we possibly get rid of the edges forcing

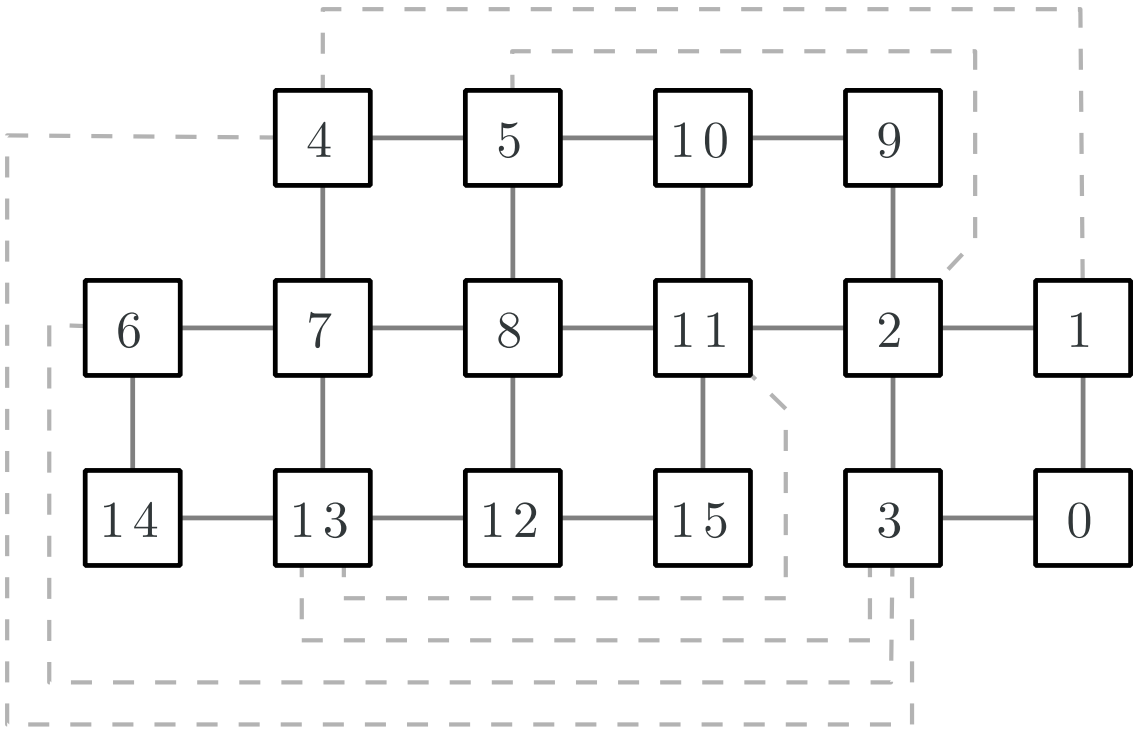


Figure 5.3.: The reduced version of our continuous example. The grid structure is well visible and the drawing is more compact.

us to bend the graph. This intuition basically asks for a tree in the dual graph and then we delete all primal edges, corresponding to the dual edges in the tree. Still we need to guarantee the connectivity properties from above.

To check if a graph is k-edge-connected we can use the dual graph $G^\star$ of G. Let C be a set of edges, forming a cycle in $G^\star$. Deleting all primal edges, corresponding to dual edges in C, disconnects the graph. Hence the size of the cycle corresponds to the number of edges we have to delete, such that G is disconnected. Let C be the shortest cycle in $G^\star$. Then G is exactly k-edge connected with k being the length of C. Otherwise there would be a set of fewer edges, such that deleting them disconnects G. In return we then find be a cycle in the dual graph of this length. For $k = 1, 2$ this says if the graph is 1-edge-connected there is a loop at the node representing the outer face in $G^\star$ and if $k = 2$ we find two parallel edges at the outer face node in the dual graph.

With the above we see, if the set E_d forms a tree in $G^\star$ and the tree does not contain any edges that are part on a cycle of length one or two we found a viable set of edges. To make E_d somewhat big at the same time we want to maximize the size of the tree. An already existing problem is to find a *maximal induced subtree* in a graph. More specific we ask for a set of nodes $V' \subseteq V$ whose induced subgraph is a tree in G and V' has maximal cardinality. Unfortunately finding the optimal solution is $\mathcal{NP}$-complete. A proof can be found in the Bachelorthesis of Simon Bischof [Bis14]. Still there might be a fast and good approximation for such trees.

After we know what we want, lets turn to the algorithms. We can use depth-first-search (DFS) and Breath-first-serach (BFS) with the same modification. We always start the search from the dual node corresponding to the outer face. Whenever we visit a node $u \in V^\star$, we mark all the neighbours from which we can reach an already visited node, using a different edge, than the one between u and that neighbour. Then we pick the next node. To decide if $v \in V^\star$ can be picked as next node we have to check if it was already visited

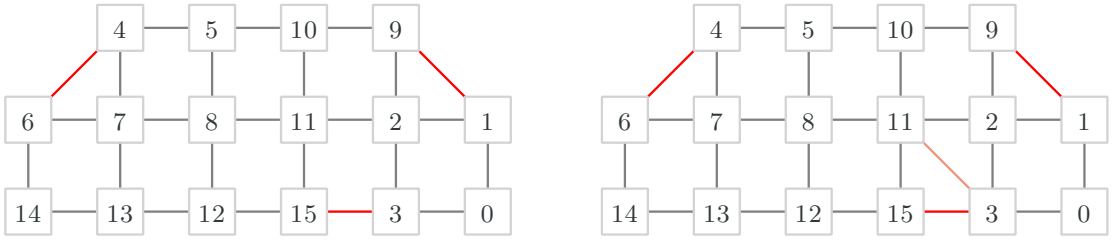
or if it is on a cycle of length two. BFS and DFS both forbid us to choose a visited node, else the result would not be a tree, and we add the constraint, that no marked node can be chosen. If either is the case we choose a different node. Checking if the node was marked is enough, because if there is a cycle of length two and we at one point pick one of the two nodes, we mark the other one. In the end we return the set of primal edges E_d which where crossed by dual edges in the induced subtree and delete them from G. For our ongoing example take a look at Figure 5.3. After applying the BFS version of the reduction we find a more grid like structure.
When we want to draw G_c, we have to keep in mind that the deleted edges in E_d have to be reinserted later on. Preserving the embedding or at least keep the same nodes on the outer face would be nice, since then we can draw all the edges around the graph. Otherwise we might create unnecessary crossings. From now on we will assume that this is the case, so for any edge in E_d we find the start and end node on the outer face of G_c. To draw the core graph we can use any method we like. In the example pictures we used an orthogonal layout algorithm provided by the OGDF library.

5.2. Reinserting Edges

After generating the drawing for G_c we have to reinsert the edges from E_d. As said above, for every edge $e \in E_d$ we assume the start and end node are on the outer face and we only consider paths for the edges lying completely in the outer face. For us the most important property of the final drawing is to not introduce any unnecessary crossings, which are only introduced because we drew an edge from E_d in a certain way. At this point we finally see the application of the previously discussed Bézier curve properties. When we can come up with some bounding polygons, such that two polygons never intersect, unless it is completely necessary, we can draw the edges in any way we like, keep them inside the boxes and guarantee no new crossings are introduced.
Chapter 3 and 4 talked about how we are able to find a curve inside a polygon. This reduces the remaining task in this section, to coming up with a construction of said bounding boxes. The basic idea we suggest here is based on two observations. If the outer face is a convex polygon, we can duplicate the shape and then route all edges in the space between the two boundaries. Assume all the faces we are interested in are triangulated, then a reinsertion path between two nodes can be found as the shortest path in the dual graph. With shortest we mean the number of nodes is minimal. Such a path is not necessarily the shortest path by euclidean length of the edge in the drawing, but it should be somewhat close to it, if the faces we create between the two copies of the outer face are somewhat equally sized.
The necessary steps we need to fill in are how do we find the convex hull (Section 5.2.1) of the outer face, triangulate all the newly created faces (Section 5.2.2), find the shortest paths in the dual graph and finally specify the bounding box for each edge (Section 5.2.3).

5.2.1. Finding the Convex Hull

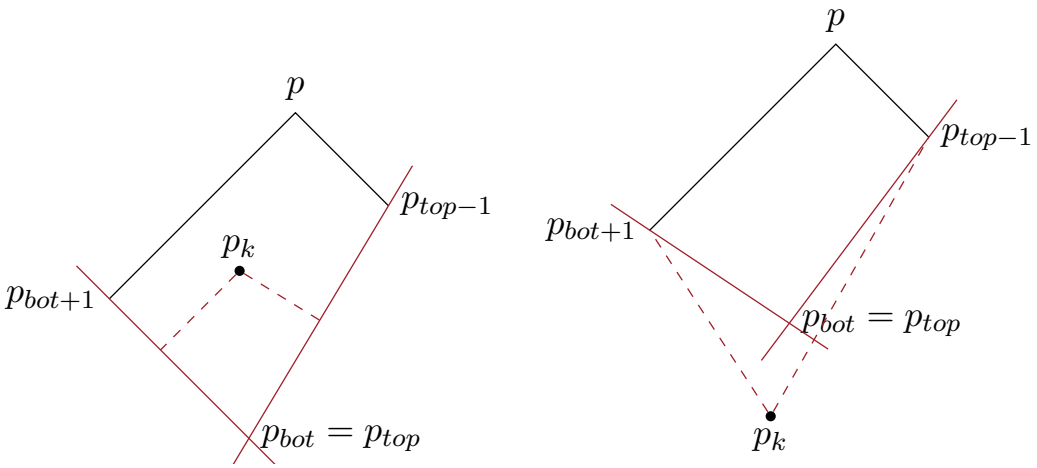
To calculate the convex hull we use an algorithm by Melkman [Mel87], which runs in $O(n)$. The only drawback is that the polygon needs to be simple. In our case we can assume the outer face of G_c to be such a polygon, because G is planar. Here we discuss the algorithm very quickly, for a formal description and correctness proof look into [Mel87]. In general the algorithm works with a simple polyline $P := \{p_0, p_1, ..., p_n\} \in \mathbb{R}^2$ and a double ended queue $Q := \{q_{bot}, ..., q_{top}\}$. As suggested by the name, Q has a front and a back and we can insert/delete an element in $O(1)$ on both sides.
Starting with p_0 we process the points in order of their index and at all times we find in Q_k the convex hull after processing k points. Consequently $q_{bot} = q_{top}$ at all times, because



(a) The outer face is now convex.

(b) Triangulating the new faces.

Figure 5.4.: Finding the convex hull and triangulate the new faces.



(a) p_k lies inside.

(b) p_k lies outside.

Figure 5.5.: Calculating the convex hull of a simple polygon. p_k is the next point we consider.

the convex hull is a closed polygon. Suppose we have looked at $k-1$ points and are now processing p_k. Either we find p_k to lie inside the current hull Q_{k-1} and we don't need to do anything or it lies outside. In the latter case we have to remove the points which are inside the polygon if p_k is added and add p_k to both ends of Q in the end.

To test if a point p_k is inside the current convex hull we check if it is to the right of both the line going through q_{bot} and q_{bot+1} and the line going through q_{top-1} and q_{top}; see Figure 5.5. In the case of both being true the point lies between the two lines and hence, because P is simple, it can not lie outside of the already computed convex hull. If p_k lies to the left of either line we have to add it to the convex hull. When we insert p_k there can be points in Q_{k-1} and therefore on the convex hull, which after p_k was inserted, lie inside the hull. For the top half of Q this can be solved by popping points as long as we find that p_k is to the left of the line through q_{top-1} and q_{top}. Analogous for the bottom half, but we have to check if p_k is to the left of the line through q_{bot} and q_{bot+1}. Finally we insert p_k as the bot and top point of Q putting it on the convex hull, satisfying $q_{bot} = q_{top}$

Applying this algorithm to the polygon of the outer face, gives us the segments we have to add between nodes on the outer face. We think of them as edges, so we can insert them in our graph. They are marked as helper edges. In Figure 5.4(a) you can see the example graph with its convex hull. The edges we had to insert are marked red. From now on the outer faces polygon of G_c is regarded as convex. In the process three new faces where created. The next step now is to triangulate them. In the example there is only one face we have to triangulate.

5.2.2. Triangulating the new Faces

For every face in G_c, which we just added by computing the convex hull, we have to find a triangulation. This is necessary for the techniques used in section 5.2.3. Let f be a face with a helper edge on its boundary. Then we need to check if it is a triangle and if not triangulate it.
The method we implemented is using ear clipping. An ear is a set of three vertices in a polygon v_0, v_1 and v_2 such that there is no other vertex on the segment between v_0 and v_1 and the segment between v_0 and v_2. If we introduce the segment between v_1 and v_2 we separated the polygon into two. One of them is a triangle, hence we cut off or sliced off an ear.
It is possible to prove the existence of an ear in any simple polygon [EET93] and if an ear is cut off the remaining polygon is still simple. Continuing this process we find in every step an ear and cut it off until there are only triangles left. A careful implementation with a greedy approach yields a running time of $O\left(n^2\right)$, where n is the number of vertices in the polygon. We only implemented a naive version running in $O\left(n^3\right)$, which is enough to demonstrate the application.

5.2.3. Routing the edges

As said above we need to find an area, where we can root the edges. We create it by duplicating the outer face, which by now is a convex polygon. For each edge $e \in E_c$ which is adjacent to the outer face of G_c we introduce a new node v, place it on the midway point of e and move it away from the core graph. The distance between the node and the outer face is chosen constant for all nodes. Two in this way inserted nodes are connected if and only if the two edges they represent had a node in common.
We end up with a convex polygon or in graph terms cycle P_c surrounding the core graph. What's left to do is the triangulation of the area between the outer face of G_c and P_c. This is done by looking for every node in P_c at the edge it represents and connect it to both the start and the end node. Figure 5.6 gives an example of what this looks like. We call this expanded graph G_e and the set of all faces we created, which are the ones with a helper edge on their boundary minus the outer face of G_e, is called H. Further we collect all edges introduced by the convex hull, triangulation and the copying of the outer face in E_H.
Next we have to find for every edge $e \in E_d$ the path it takes from its start node to its end node. Switching to the dual graph allows us to find a path between the faces in H. To prevent any edge from going right through the core graph we forbid any dual edge corresponding to a primal edge that is an original graph edge. Additionally we forbid to cross any edge connected to the node representing the outer face of G_e. In the end we can only route through dual nodes and edges corresponding to the faces in H and primal edges on their boundary.
Finding the shortest path between the start and end node for every edge $e \in E_d$ makes it necessary that we we add the start and end node of e to $G_e^\star$. We call these two new nodes s and t. Then we connect s to all the dual nodes, which represent faces on whose boundary we find s. The end node t is connected to $G_e^\star$ in the same way. A BFS search is then used to find the path between s and t. Such a path gives us a set of dual edges. Taking the corresponding primal edges we found all edges e has to cross. All these edges lie on the boundary of some face in H. Doing this for all edges we get for every primal edge $f \in E_e$ the number of edges from E_d we route over this edge.
Before we can compute the bounding boxes for each edge we have to find an order of the edges in E_d, such that we get no crossings or as few as possible. We assume all edges to have their start and end node on the outer face it is enough to just look at the intervals of nodes between start and end node. For an edge $e \in E_d$ there can are two intervals around the outer face. We name them r_e and l_e. The first, r_e, contains all nodes if we go around

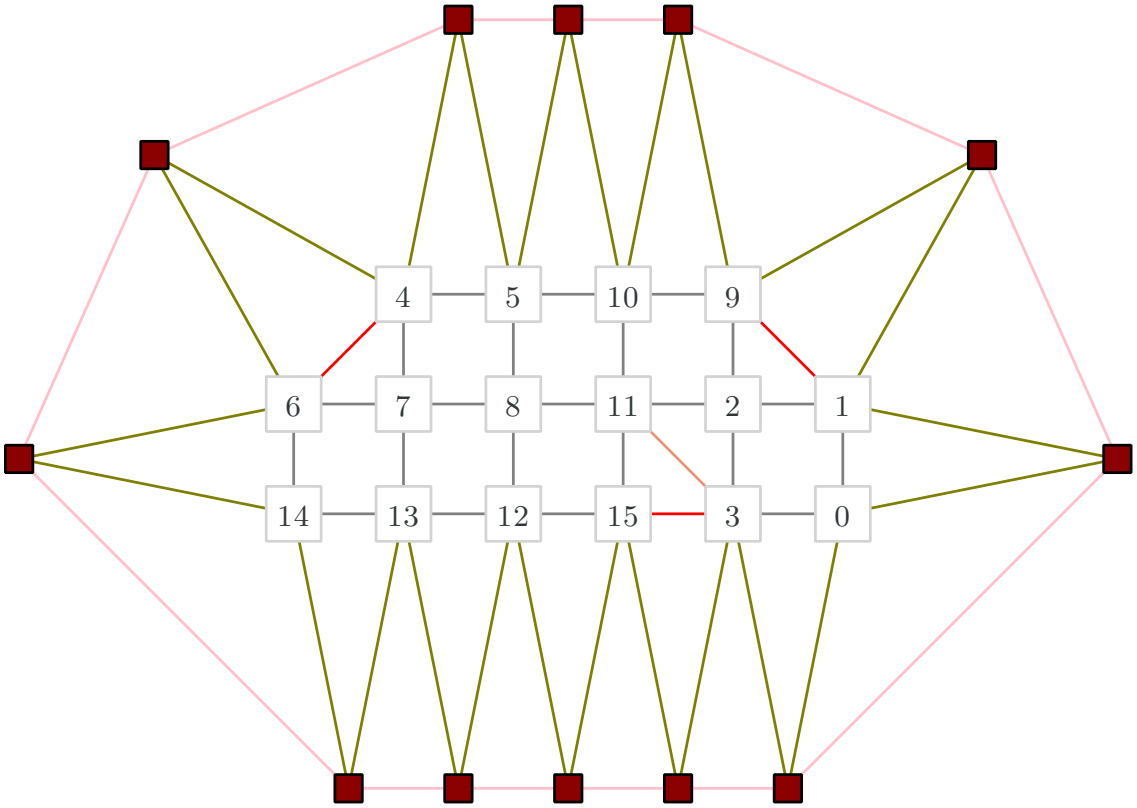


Figure 5.6.: The graph G_e after we introduced the copied outer face P_c and triangulated the area between P_c and the outer face of G_c.

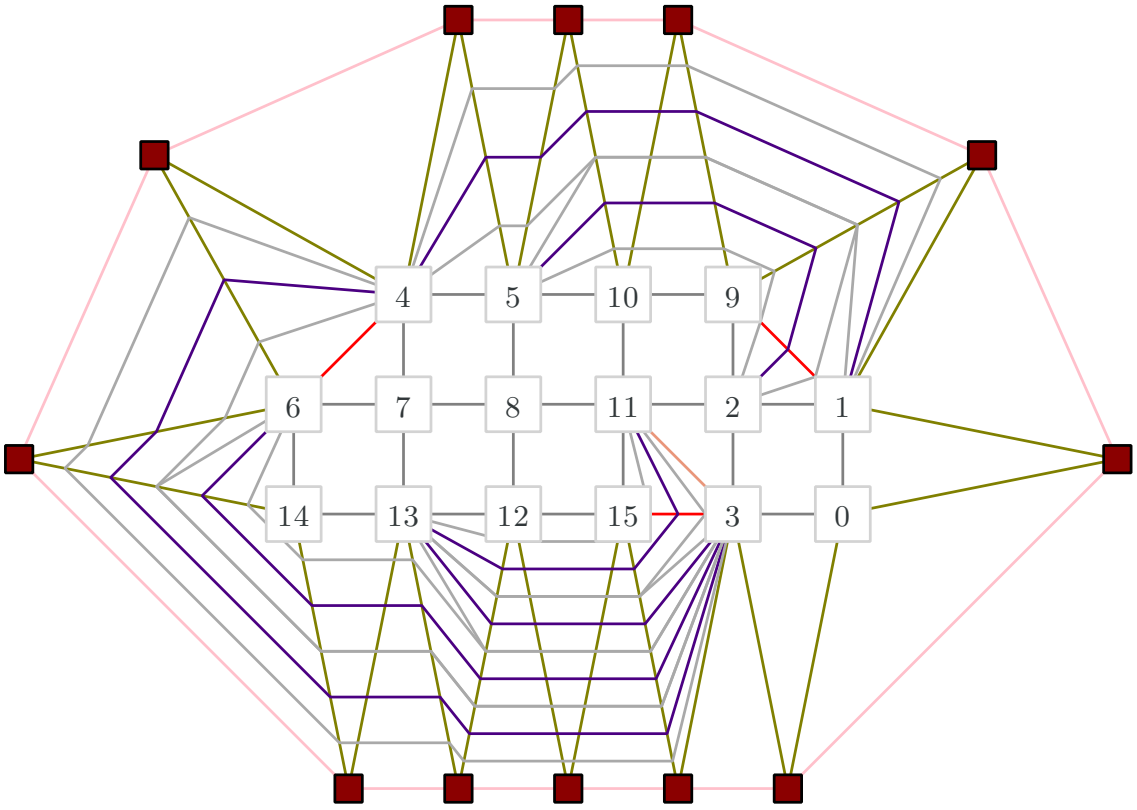


Figure 5.7.: The graph G_e after we inserted the deleted edges as polylines.

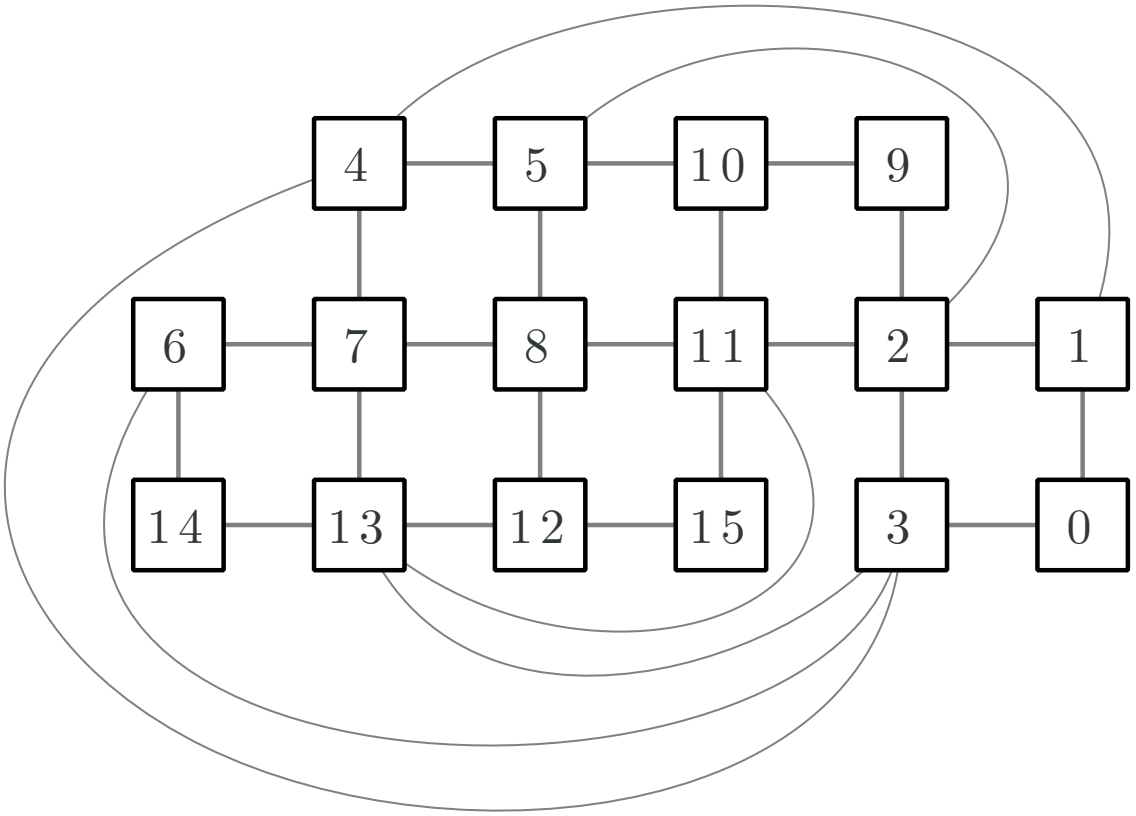


Figure 5.8.: The example graph drawn with edges as curves. This would be the goal of the method, outlined in this chapter.

the outer face clockwise and l_e contains all the nodes if we go around it counterclockwise Without loss of generality we consider only the clockwise interval. For two edges $e \neq f \in E_d$ we say their intervals overlap if we find a set of nodes containing the start node of e and the end node of f or the other way around, such that this set is a real subset of r_e and r_f. We say an interval r_e contains another interval r_f if $r_f \subseteq r_e$. Now for any two edges they can only cross if their intervals r_e and r_f or l_e and l_f overlap. Finding an order of the edges, such that as few as possible edges cross can then be done by computing which edges interval contains which and use this information to distribute the edges on the helper edges, which are crossed.
For every edge in E_H we know how many edges cross it and in which order they appear. This allows us to compute the bounding boxes. The easiest way is to simply take the distance between the start and end node of an edge in E_H, compute the distance between them and then give every edge the same space. Using this method we get the bounding boxes, which you can see in light grey in Figure 5.7. This method can of course be changed. An idea would be to give the node on the outer face of the original graph G an attracting force and the node on the outer face of G_e a repulsing one. The effect would be, that the bounding boxes would be closer to the original graph and hence the resulting edges would be closer to the graph as well. Here lies definitely some potential future work.
In the end we can route the edges freely in the computed bounding boxes. Figure 5.7 shows them in purple. Here we use simple polylines, but if one could implement the method discussed in 4, this would be a perfect application. The resulting drawing then might look like the one in Figure 5.8.

6. Conclusion

In this thesis we introduced the k-curve path problem, a generalization of the k-link path problem to polynomial curves. For the case of $k = 1$ we investigated two methods of computing the control points of a Bézier curve. A direct approach to compute them only works in the case of a quadratic curve, where we are able to describe an area containing all allowed control points. Curves of higher degree proved to be more difficult. The idea we followed is to sample the control points. Sampling in a naive way needs a big set of samples. Using ε-nets we can guarantee that small sets of sample points are enough, if the allowed area has a bounded VC-Dimension.
Finally, we presented an application. Adding edges to a graph as curves can be described as routing the edges through polygons. Given a set of disjoint polygons we can guarantee no two edges intersect if no, curve used to draw an edge, intersects the boundary of its bounding polygon.

Future Work

The next step to extend the presented results is to prove Conjecture 4.6. The conversion between bounding the VC-Dimension of the allowed area and a colouring problem seems promising. If the VC-Dimension of the allowed area proves to be bounded, one can develop a sampling algorithm by following the constructive proof of the ε-net theorem.
An advantage when using ε-nets is the ability to add more restrictions to the set of forbidden points, as long as we can describe them as sets of bounded VC-Dimension. For example if we utilize the sampling, we might be interested in excluding all curves containing a loop. Describing the control points leading to a loop as a set with bounded VC-Dimension, we can use Lemma 4.3 to get a bound on the VC-Dimension of the union between the old set and the restriction we are adding. Afterwards one can again use ε-nets to guarantee a small sample.
Solving the general k-curve path problem for $k > 1$ posts new problems. On the one hand we get restrictions at the control points, because we want the complete curve to be smooth. But on the other hand start and end point are no longer fixed for every curve segment. Restricting the possible end points of one segment by a linear equation might be a good start.

Bibliography

[BB87] John C Beatty and Brian A Barsky. *An introduction to splines for use in computer graphics and geometric modeling*. Morgan Kaufmann, 1987.

[Bis14] Simon Bischof. Induzierte Bäume in Planaren Graphen. Karlsruher Institute of Technology (KIT), Bachelorthesis, 2014.

[CFG+14] Timothy M Chan, Fabrizio Frati, Carsten Gutwenger, Anna Lubiw, Petra Mutzel, and Marcus Schaefer. Drawing partially embedded and simultaneously planar graphs. In *Graph Drawing*, pages 25–39. Springer, 2014.

[CGJ+11] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W Klau, Karsten Klein, and Petra Mutzel. The open graph drawing framework (ogdf). *Handbook of Graph Drawing and Visualization*, pages 543–569, 2011.

[DETT94] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Comput. Geom.*, 4:235–282, 1994.

[DGKN97] David P. Dobkin, Emden R. Gansner, Eleftherios Koutsofios, and Stephen C. North. Implementing a general-purpose edge router. In *Graph Drawing, 5th International Symposium, GD '97, Rome, Italy, September 18-20, 1997, Proceedings*, pages 262–271, 1997.

[EET93] Hossam A. ElGindy, Hazel Everett, and Godfried T. Toussaint. Slicing an ear using prune-and-search. *Pattern Recognition Letters*, 14(9):719–722, 1993.

[GJ83] Michael R Garey and David S Johnson. Crossing number is np-complete. *SIAM Journal on Algebraic Discrete Methods*, 4(3):312–316, 1983.

[Kob12] Stephen G. Kobourov. Spring embedders and force directed graph drawing algorithms. *CoRR*, abs/1201.3011, 2012.

[LP84] Der-Tsai Lee and Franco P Preparata. Euclidean shortest paths in the presence of rectilinear barriers. *Networks*, 14(3):393–410, 1984.

[Mat02] Jiří Matoušek. *Lectures on discrete geometry*, volume 108. 2002.

[Mel87] Avraham A. Melkman. On-line construction of the convex hull of a simple polyline. *Inf. Process. Lett.*, 25(1):11–12, 1987.

[Mit91] Joseph SB Mitchell. A new algorithm for shortest paths among obstacles in the plane. *Annals of Mathematics and Artificial Intelligence*, 3(1):83–105, 1991.

[MRW92] Joseph SB Mitchell, Günter Rote, and Gerhard Woeginger. Minimum-link paths among obstacles in the plane. *Algorithmica*, 8(1-6):431–459, 1992.

[SAG84] Thomas W Sederberg, David C Anderson, and Ronald N Goldman. Implicit representation of parametric curves and surfaces. *Computer Vision, Graphics, and Image Processing*, 28(1):72–84, 1984.

[Sch90] Philip J Schneider. An algorithm for automatically fitting digitized curves. In *Graphics gems*, pages 612–626. Academic Press Professional, Inc., 1990.

[SG86] Thomas W Sederberg and Ronald N Goldman. Algebraic geometry for computer-aided geometric design. *Computer Graphics and Applications, IEEE*, 6(6):52–59, 1986.

[SS86] Micha Sharir and Amir Schorr. On shortest paths in polyhedral spaces. *SIAM Journal on Computing*, 15(1):193–215, 1986.

[Sur86] Subhash Suri. A linear time algorithm for minimum link paths inside a simple polygon. *Computer Vision, Graphics, and Image Processing*, 35(1):99–110, 1986.

[Wol14] Wolfram Research, Inc., Mathematica, Version 10.0, Champaign, IL, 2014.

[yWo15] yWorks GmbH, yEd, Version 3.14, Tübingen, Germany, 2015.

Appendix

A. Mathematica Code

This section shows some of the used Mathematica code. Besides the code to generate an interactive demo of the forbidden area for the degree 2 and the degree 3 case, we show the code for the figures generated in Mathematica. The code was written and tested with the Mathematica 10.0.1.0 Linux 64-bit version.

Quadratic Bézier Curve

```
(*Define standard 2d curve , line segment and genral points*)

B[t_, ps_List, pc_List, pe_List] := (1 - t)^2*ps + 2*(1 - t)*t*pc + t^2*pe

L[s_, b_List, e_List] := b + s*e

b := {bx, by}

e := {ex, ey}

ps := {psx, psy}

pc := {pcx, pcy}

pe := {pex, pey}

(*Calculate all forbidden points*)

forbiddenResult := Simplify[Solve[
   B[t, ps, pc, pe] == L[s, b, e]
    && 0 <= s <= 1
    && 0 <= t <= 1, pc, Reals]]

forbiddenPoints := Simplify[{
   pcx /. Extract[Normal[forbiddenResult], {1}],
   pcy /. Extract[Normal[forbiddenResult], {1}]
  }]

f[t_, s_, ps_List, pe_List, b_List, e_List] :=
 -(((1 - t)^2*ps + t^2*pe - b - s*(e - b))/(2*(1 - t)*t))

(*Get derivative of forbidden points second component and set it to zero*)

DF := Simplify[D[f[t, s, ps, pe, b, e], t]][[2]]

Simplify[Reduce[DF == 0
   && 0 < t < 1
   && s == 0
   && by == 0
   && ey == 0, t, Reals]]

(*Define ti(P)*)

t[psy_, pey_] := Which[
  pey == psy, 1/2,
  Abs[psy] < Abs[pey],
      -(psy/(pey - psy)) + Sqrt[(pey*psy)/(pey - psy)^2],
  Abs[psy] > Abs[pey],
      -(psy/(pey - psy)) - Sqrt[(pey*psy)/(pey - psy)^2]]

(*Formulas for case differentiation over x-coordinate*)

Simplify[Reduce[
  f[t0, 0, ps, pe, {bx, 0}, {ex, 0}] == f[t1, 1, ps, pe, {bx, 0}, {ex, 0}]
   && 0 < t0 < t1 < 1
   && ((pey > 0 && psy > 0) || (pey < 0 && psy < 0))
   && ex > bx > 0, t0, Reals]]

(*Interactive demo*)
```

```
With[
 {b={-3,0},
  e={3,0}},Manipulate[Show[
    ParametricPlot[f[t, s, ps, pe, b, e],
        {t, 0.01, 0.99}, {s, 0, 1}, PlotStyle -> LightGray],
    ParametricPlot[f[t[ps[[2]], pe[[2]]], s, ps, pe, b, e],
        {s, 0, 1}, PlotStyle -> Orange],
    ParametricPlot[f[t, 0, ps, pe, b, e],
        {t, 0.01, 0.99}, PlotStyle -> {Dashing[0], Blue}],
   ParametricPlot[f[t, 1, ps, pe, b, e],
        {t, 0.01, 0.99}, PlotStyle -> Blue],
    ParametricPlot[L[s, b, e - b], {s, 0., 1.}, PlotStyle -> Red],
   ParametricPlot[B[t, ps, pc, pe], {t, 0, 1}, PlotStyle -> Green],
    Graphics[{Gray, PointSize[0.02], Point[ps], Point[pc],
         Point[pe], Point[b], Point[e]}],
   PlotRange -> {{-10,10}, {-10, 10}}, AxesStyle -> Gray, Frame -> False,
    Ticks -> None, Method -> {"AxesInfront" -> False}, ImageSize -> Large],
  {{ps,{-3,1.65}},Locator},
  {{pc,{-1,-3.8}},Locator},
  {{pe,{3,-1.25}},Locator}
 ]]

(*Figures*)

With[{
   ps = {2, -1},
   pc = {4.5, 0.75},
   pe = {7, -2},
   b = {3, 0},
   e = {6, 0}},
  Show[
   ParametricPlot[f[t, s, ps, pe, b, e],
       {t, 0.01, 0.99}, {s, 0, 1}, PlotStyle -> LightGray],
   ParametricPlot[f[t[ps[[2]], pe[[2]]], s, ps, pe, b, e],
       {s, 0, 1}, PlotStyle -> Orange],
   ParametricPlot[f[t, 0, ps, pe, b, e],
       {t, 0.01, 0.99}, PlotStyle -> {Dashing[0], Blue}],
   ParametricPlot[f[t, 1, ps, pe, b, e],
       {t, 0.01, 0.99}, PlotStyle -> Blue],
   ParametricPlot[L[s, b, e - b], {s, 0., 1.}, PlotStyle -> Red],
   ParametricPlot[B[t, ps, pc, pe], {t, 0, 1}, PlotStyle -> Green],
   Graphics[{Gray, PointSize[0.02], Point[ps], Point[pc],
        Point[pe], Point[b], Point[e]}],
   PlotRange -> {{0, 9}, {-2, 5}}, AxesStyle -> Gray, Frame -> False,
   Ticks -> None, Method -> {"AxesInfront" -> False}, ImageSize -> Large
 ]
]
```

```
With[{
   ps = {3.5, -0.5},
   pc = {4.5, 0.75},
   pe = {6, -2},
   b = {3, 0},
   e = {6, 0}},
  Show[
   ParametricPlot[f[t, s, ps, pe, b, e],
       {t, 0.01, 0.99}, {s, 0, 1}, PlotStyle -> LightGray],
   ParametricPlot[f[t, 0, ps, pe, b, e],
       {t, 0.01, 0.99}, PlotStyle -> {Dashing[0], Blue}],
   ParametricPlot[f[t, 1, ps, pe, b, e],
       {t, 0.01, 0.99}, PlotStyle -> Blue],
   ParametricPlot[L[s, b, e - b], {s, 0., 1.}, PlotStyle -> Red],
   ParametricPlot[B[t, ps, pc, pe], {t, 0, 1}, PlotStyle -> Green],
   Graphics[{Gray, PointSize[0.02], Point[ps], Point[pc],
        Point[pe], Point[b], Point[e]}],
   ParametricPlot[
       {{(pe[[2]]*(ps[[1]] - b[[1]]))/ps[[2]] + b[[1]] + (e[[1]] - b[[1]]), t}},
       {t, -10, 10}, PlotStyle -> {{Dashed, Orange}, {Dashed, Orange}}
  ],
   PlotRange -> {{0, 9}, {-2, 5}}, AxesStyle -> Gray, Frame -> False,
   Ticks -> None, Method -> {"AxesInfront" -> False}, ImageSize -> Large
 ]
]

With[{
   ps = {2, -1},
   pc = {4.5, 0.75},
   pe = {7, -2},
   b = {3, 0},
   e = {6, 0}},
  Show[
   ParametricPlot[f[t, s, ps, pe, b, e],
       {t, 0.01, 0.99}, {s, 0, 1}, PlotStyle -> LightGray],
   ParametricPlot[f[t, 0, ps, pe, b, e],
       {t, 0.01, 0.99}, PlotStyle -> {Dashing[0], Blue}],
   ParametricPlot[f[t, 1, ps, pe, b, e],
       {t, 0.01, 0.99}, PlotStyle -> Blue],
   ParametricPlot[L[s, b, e - b], {s, 0., 1.}, PlotStyle -> Red],
   ParametricPlot[B[t, ps, pc, pe], {t, 0, 1}, PlotStyle -> Green],
   Graphics[{Gray, PointSize[0.02], Point[ps], Point[pc],
        Point[pe], Point[b], Point[e]}],
   PlotRange -> {{0, 9}, {-2, 5}}, AxesStyle -> Gray, Frame -> False,
   Ticks -> None, Method -> {"AxesInfront" -> False}, ImageSize -> Large
 ]
]
```

```
With[{
   ps = {2, -1.5},
   pc = {2.2, 1.8},
   pe = {7, 2},
   b = {3, 0},
   e = {6, 0}},
  Show[
   ParametricPlot[f[t, s, ps, pe, b, e],
       {t, 0.01, 0.99}, {s, 0, 1}, PlotStyle -> LightGray],
   ParametricPlot[f[t, 0, ps, pe, b, e],
       {t, 0.01, 0.99}, PlotStyle -> {Dashing[0], Blue}],
   ParametricPlot[f[t, 1, ps, pe, b, e],
       {t, 0.01, 0.99}, PlotStyle -> Blue],
   ParametricPlot[L[s, b, e - b], {s, 0., 1.}, PlotStyle -> Red],
   ParametricPlot[B[t, ps, pc, pe], {t, 0, 1}, PlotStyle -> Green],
   Graphics[{Gray, PointSize[0.02], Point[ps], Point[pc],
        Point[pe], Point[b], Point[e]}],
   PlotRange -> {{0, 9}, {-2, 5}}, AxesStyle -> Gray, Frame -> False,
   Ticks -> None, Method -> {"AxesInfront" -> False}, ImageSize -> Large
 ]
]
```

```
With[{
   ps = {2, -0.7},
   pc = {4.5, 0.75},
   pe = {7, -2},
   b = {3, 0},
   e = {6, 0}},
  Show[
   ParametricPlot[f[t, s, ps, pe, b, e],
       {t, 0.01, 0.99}, {s, 0, 1}, PlotStyle -> LightGray],
   ParametricPlot[f[t, 0, ps, pe, b, e],
       {t, 0.01, 0.99}, PlotStyle -> {Dashing[0], Blue}],
   ParametricPlot[f[t, 1, ps, pe, b, e],
       {t, 0.01, 0.99}, PlotStyle -> Blue],
   ParametricPlot[L[s, b, e - b], {s, 0., 1.}, PlotStyle -> Red],
   ParametricPlot[B[t, ps, pc, pe], {t, 0, 1}, PlotStyle -> Green],
   Graphics[{Gray, PointSize[0.02], Point[ps], Point[pc],
        Point[pe], Point[b], Point[e]}],
   ParametricPlot[
       {{(pe[[2]]*(ps[[1]] - b[[1]]))/ps[[2]] + b[[1]] + (e[[1]] - b[[1]]), t}},
       {t, -10, 10}, PlotStyle -> {{Dashed, Orange}, {Dashed, Orange}}
  ],
   PlotRange -> {{0, 9}, {-2, 5}}, AxesStyle -> Gray, Frame -> False,
   Ticks -> None, Method -> {"AxesInfront" -> False}, ImageSize -> Large
 ]
]
```

Cubic Bézier Curve

```
(*Define standard 3d curve , line segment and genral points*)

B[t_, ps_List, pc1_List, pc2_List, pe_List] :=
              (1 - t) ^ 3 ps + 3 (1 - t) ^ 2 t pc1 + 3 (1 - t) t ^ 2 pc2 + t ^ 3 pe

L[s_, b_List, e_List] := b + s * e

b := {bx, by}

e := {ex, ey}

ps := {psx, psy}

pc1 := {pc1x, pc1y}

pc2 := {pc2x, pc2y}

pe := {pex, pey}

(*Calculate all forbidden points*)

forbiddenResult := Simplify[Solve[B[t, ps, pc1, pc2, pe] == L[s, b, e]
    && 0 <= s <= 1
    && 0 <= t <= 1, pc2, Reals]]

forbiddenPoints := Simplify[{
   pcx /. Extract[Normal[forbiddenResult], {1}],
   pcy /. Extract[Normal[forbiddenResult], {1}]
  }]

F2[t_, s_, ps_, pe_, pc_, u_, v_] :=
         - (((1 - t) ^ 3 ps + t ^ 3 pe + 3 (1 - t) ^ 2 t * pc - u - s * (v - u)) / (3 (1 - t) t ^ 2))

(*Get derivative of forbidden points second component and set it to zero*)

DF := Simplify[D[F2[t, s, ps, pc1, pe, b, e], t]][[2]]

Simplify[Reduce[DF == 0
   && 0 < t < 1
   && s == 0 && by == 0
   && ey == 0, t, Reals]]

(*Define ti(P)*)

t0[psy_, pc1y_, pey_] := Which[
  psy == pey, Which[
   psy < 0, Which[
    pc1y > psy, {troot[psy, pc1y, pey, 1]},
    pc1y < psy, {troot[psy, pc1y, pey, 2]},
    pc1y == psy, 2 psy / (-3 pc1y + 6 psy)],
   psy > 0, Which[
    pc1y > psy, {troot[psy, pc1y, pey, 2]},
    pc1y < psy, {troot[psy, pc1y, pey, 1]},
    pc1y == psy, 2 psy / (-3 pc1y + 6 psy)]
  ],
  psy < pey, Which[
```

```
 pey < 0, Which[
  pc1y ≥ pey, troot[psy, pc1y, pey, 1],
  pc1y ≤ psy, troot[psy, pc1y, pey, 2],
  psy < pc1y < pey, Which[
   2 / 3 psy ≤ pc1y, troot[psy, pc1y, pey, 1],
   2 / 3 psy > pc1y, Which[
    pc1y ≥ (pey + 2 psy) / 3, {troot[psy, pc1y, pey, 1]},
    pc1y < (pey + 2 psy) / 3, {troot[psy, pc1y, pey, 2]}]]],
 psy > 0, Which[
  pc1y ≥ pey, {troot[psy, pc1y, pey, 2]},
  pc1y ≤ psy, {troot[psy, pc1y, pey, 1]},
  psy < pc1y < pey, Which[
   pc1y^3 < pey * psy^2, {troot[psy, pc1y, pey, 1]},
   3 pc1y < pey + 2 psy && pc1y^3 ≥ pey * psy^2, {troot[psy, pc1y, pey, 3]},
   3 pc1y ≥ pey + 2 psy, {troot[psy, pc1y, pey, 2]}]],
 psy < 0 && pey > 0, Which[
  pc1y > pey > 0, Which[
   pc1y ≥ -psy, {troot[psy, pc1y, pey, 1], troot[psy, pc1y, pey, 2]},
   pc1y < -psy, Which[
    pc1y^3 == pey psy^2, {troot[psy, pc1y, pey, 1]},
    pc1y^3 > pey psy^2, {troot[psy, pc1y, pey, 1], troot[psy, pc1y, pey, 2]},
    True, {}],
   True, {}],
  psy < pc1y < pey, Which[
   -psy < pc1y ≤ -2 psy, Which[
    pc1y^3 == pey psy^2, {troot[psy, pc1y, pey, 1]},
    psy^2 * pey < pc1y^3, {troot[psy, pc1y, pey, 1], troot[psy, pc1y, pey, 2]},
    True, {}],
   pc1y > -2 psy, Which[
    3 pc1y ≥ pey + 2 psy,
        {troot[psy, pc1y, pey, 1], troot[psy, pc1y, pey, 2]},
    3 pc1y < pey + 2 psy && pey * psy^2 < pc1y^3,
        {troot[psy, pc1y, pey, 2], troot[psy, pc1y, pey, 3]},
    pc1y^3 == pey psy^2, {troot[psy, pc1y, pey, 2]},
    True, {}],
   True, {}],
  True, {}]
],
psy > pey, Which[
 psy < 0, Which[
  pc1y ≥ psy, {troot[psy, pc1y, pey, 2]},
  pc1y ≤ pey, {troot[psy, pc1y, pey, 1]},
  psy > pc1y > pey, Which[
   pc1y^3 > pey * psy^2, {troot[psy, pc1y, pey, 1]},
   3 pc1y > pey + 2 psy && pc1y^3 ≤ pey * psy^2, {troot[psy, pc1y, pey, 3]},
   3 pc1y ≤ pey + 2 psy, {troot[psy, pc1y, pey, 2]}]],
```

```
   pey > 0, Which[
    pc1y ≥ psy, troot[psy, pc1y, pey, 1],
    pc1y ≤ pey, troot[psy, pc1y, pey, 2],
    psy > pc1y > pey, Which[
     2 / 3 psy ≥ pc1y, troot[psy, pc1y, pey, 1],
     2 / 3 psy < pc1y, Which[
      pc1y ≤ (pey + 2 psy) / 3, {troot[psy, pc1y, pey, 1]},
      pc1y > (pey + 2 psy) / 3, {troot[psy, pc1y, pey, 2]}]]],
   psy > 0 && pey < 0, Which[
    pc1y < pey < 0, Which[
     pc1y ≤ -psy, {troot[psy, pc1y, pey, 1], troot[psy, pc1y, pey, 2]},
     pc1y > -psy, Which[
      pc1y^3 == pey psy^2, {troot[psy, pc1y, pey, 1]},
      pc1y^3 < pey psy^2, {troot[psy, pc1y, pey, 1], troot[psy, pc1y, pey, 2]},
      True, {}],
     True, {}],
    psy > pc1y > pey, Which[
     -2 psy ≤ pc1y < -psy, Which[
      pc1y^3 == pey psy^2,
          {troot[psy, pc1y, pey, 1]},
      psy^2 * pey > pc1y^3,
          {troot[psy, pc1y, pey, 1], troot[psy, pc1y, pey, 2]},
      True, {}],
     pc1y < -2 psy, Which[
      3 pc1y ≤ pey + 2 psy,
          {troot[psy, pc1y, pey, 1], troot[psy, pc1y, pey, 2]},
      3 pc1y > pey + 2 psy && pey * psy^2 > pc1y^3,
          {troot[psy, pc1y, pey, 2], troot[psy, pc1y, pey, 3]},
      pc1y^3 == pey psy^2, {troot[psy, pc1y, pey, 2]},
      True, {}],
     True, {}],
    True, {}]]
 ]

(*Interactive demo*)
```

```
With[
 {b = {-3, 0},
  e = {3, 0}}, Manipulate[Show[
   ParametricPlot[F2[t, s, ps, pe, pc1, b, e],
    {t, 0.01, 0.99}, {s, 0, 1}, PlotStyle → Red],
   With[{tt = t0[ps[[2]], pc1[[2]], pe[[2]]]},
    Table[ParametricPlot[F2[t, s, ps, pe, pc1, b, e], {s, 0, 1}], {t, tt}]],
   ParametricPlot[B[t, ps, pc1, pc2, pe], {t, 0.01, 0.99}],
   ParametricPlot[L[s, b, (e - b)], {s, 0, 1}, PlotStyle → Green],
   Graphics[{Gray, PointSize[0.02], Point[b], Point[e]}],
   PlotRange → {{-10, 10}, {-10, 10}}],
  {{ps, {-3, 1.65}}, Locator},
  {{pc1, {-3.37, -3.8}}, Locator},
  {{pc2, {6.5, 4.2}}, Locator},
  {{pe, {3, -1.25}}, Locator}
 ]]

(*Figures*)

With[{ps = {2.7, 1.65},
  pc1 = {3.37, -3.8},
  pc2 = {7.5, 4.2},
  pe = {9, -1.25},
  b = {3, 0},
  e = {9, 0}
 }, Show[ParametricPlot[F2[t, s, ps, pe, pc1, b, e],
   {t, 0.01, 0.99}, {s, 0, 1}, PlotStyle → LightGray],
  With[{tt = t0[ps[[2]], pc1[[2]], pe[[2]]]},
       Table[ParametricPlot[
     F2[t0, s, ps, pe, pc1, b, e], {s, 0, 1}, PlotStyle → Orange], {t0, tt}]],
  ParametricPlot[F2[t, 0, ps, pe, pc1, b, e], {t, 0.01, 0.99}, PlotStyle → Blue],
  ParametricPlot[F2[t, 1, ps, pe, pc1, b, e], {t, 0.01, 0.99}, PlotStyle → Blue],
  ParametricPlot[L[s, b, e - b], {s, 0.00, 1.00}, PlotStyle → Red],
  ParametricPlot[B[t, ps, pc1, pc2, pe], {t, 0, 1}, PlotStyle → Green],
  Graphics[{Gray, PointSize[0.02], Point[ps], Point[pc1], Point[pc2],
        Point[pe], Point[b], Point[e]}],
  PlotRange → {{-2, 18}, {-6.5, 10}}, AxesStyle → Gray, Frame → False,
      Ticks → None, Method → {"AxesInFront" → False}, ImageSize → Medium]]
```