Resolving Popular Faces in Curve Arrangements

Utrecht University

Phoebe de Nooijer

Supervisors: Maarten Löffler and Tamara Mtsentlintze

10 February 2022

Abstract

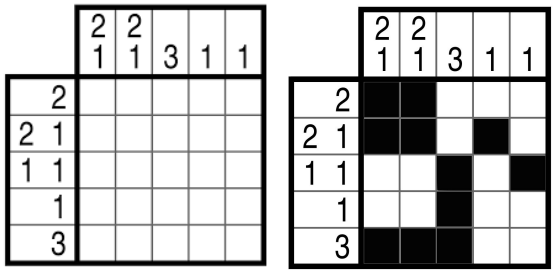
In this thesis, we propose methods for resolving popular faces in curve arrangements by adding curves to the arrangement. A popular face is a face that is bordered by two or more edges that belong to the same curve. We introduce two algorithms that aim to resolve these popular faces. The first algorithm aims to resolve each popular face individually. The second algorithm aims to resolve these faces at once using a single curve. For this second algorithm, we use a path finding method introduced by Björklund et al. for returning a cycle that passes through a provided list of specified elements.[2] This method runs in fixed parameter tractable time. Finally, we test both algorithms on a test set and perform a short comparative study on the results.

Contents

Chapter 1

Introduction

Nonograms (see figure 1.1), also known as Japanese puzzles or paint-by-number puzzles, are a type of pen-and-paper puzzle that is played on a grid. Said grid contains *hints* for each row and column of squares. The hints communicate to the puzzler how many consecutive squares on the row or column the hints are placed in should be colored black, with at least one white square separating these consecutive black squares. By combining the hints associated with the rows and columns of the puzzle, the puzzler can figure out the *solution* of a nonogram. An example of a nonogram and its corresponding solution can be seen in Figure 1.1.



(a) (b)

Figure 1.1: An example of a classic nonogram puzzle (a) with its solution (b). (Image from [11])

A curved nonogram (see Figure 1.2) is a variation on the classic nonogram involving a curve arrangement within a preset bounding space, usually a rectangle. A *curve arrangement* is the partition of the plane formed by a collection of curves. The curves will subdivide each other at intersection points, thus leaving us with *curve segments*. All curve segments will be contained within a bounding space. Instead of placing the hints in rows or columns like in a classic nonogram, a curved nonogram has hints for both sides of each curve. The hints will communicate to the puzzler how many consecutive black faces will be incident to that side of the curve. A *face* is a region within the curve arrangement that is bounded by a set of curve segments. We assume that in the curve arrangements

that form a curved nonogram, at most two curves can intersect at the same point. [4]

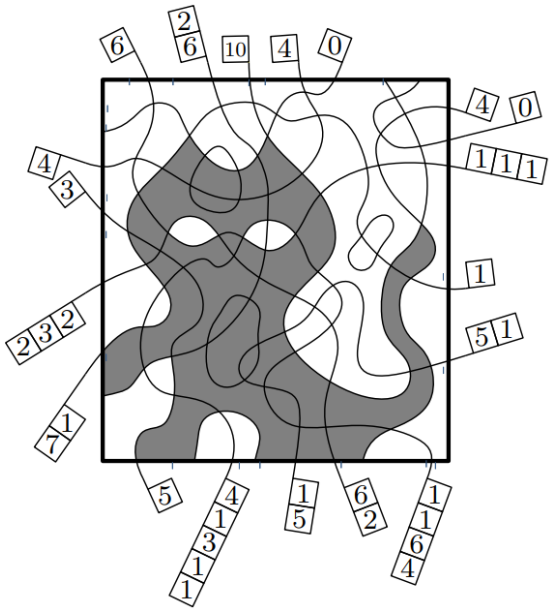
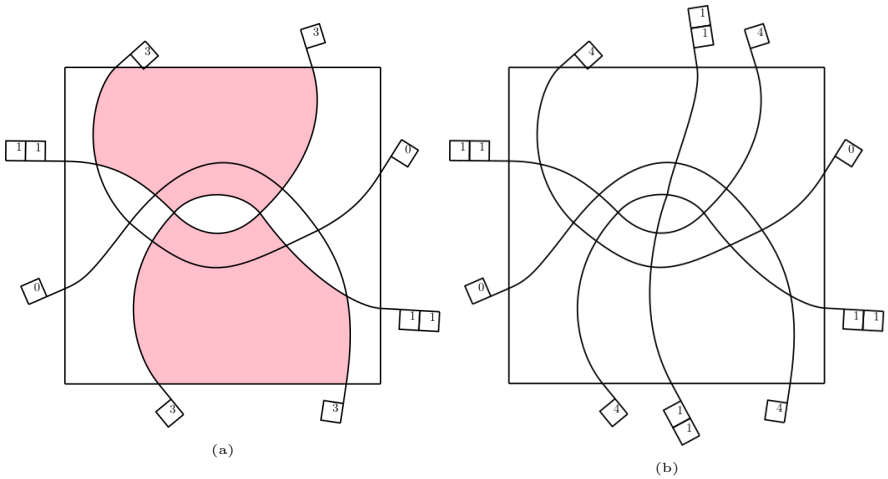


Figure 1.2: An example of a solved curved nonogram. (Image from [9])

We define *popular faces* as faces that have two or more edges that belong to the same curve. During the process of the generation of a curved nonogram, curve segments might be connected in a manner that would create a popular face in the arrangement. This is usually because such curve segments are close to one another and therefore easy for an algorithm to connect. Popular faces in a curved nonogram can pose an issue for puzzlers with little experience with curved nonograms. The labels have to contain multiple hints for the same face, which can confuse these puzzlers. We want to be able to adjust curved nonograms with popular faces such that we would be able to offer a less intimidating version of it to a puzzler with less experience. For this reason, we want to be able to *resolve* all popular faces of any curved nonogram containing any.

Let us say, for example, that we have a curve arrangement A that contains a popular face F_p that is visited twice by curve c_p. We call the two edges of F_p that belong to c_p edges e_1 and e_2. Popular face F_p could be corrected by adding a curve c_i that would split F_p, such that the two edges e_1 and e_2 are on the opposite sides of c_i. An example of a curved nonogram with popular faces and how these popular faces would be corrected by inserting a curve segment can be seen in Figure 1.3. Removing the curve that has multiple edges that belong to the same face might also resolve the popular face in some cases. Transforming certain curves might also be a solution to this problem. In this thesis, the focus lies solely on adding one or multiple curves to the original curve arrangement.

The goal of this thesis is to design an algorithm for correcting popular faces



(a) (b)

Figure 1.3: A curved nonogram with all popular faces marked in pink (a), and the same curved nonogram for which these popular faces are resolved (b).

by inserting one or more curves to the given arrangement. We introduce two different methods for correcting popular faces. The first is an iterative algorithm that tries to resolve the popular faces one by one by adding a curve that enters the arrangement from the border, goes through the popular face in a manner that resolves it, and then exits the arrangement through the border. To find its path to and from the popular face, this algorithm makes use of simple breadth first search. The second algorithm applies a novel search tactic such that the added curve enters through the border, then pass through all popular faces in a manner that resolves them all, then exit through the border.

We assess the effectiveness and efficiency of the algorithms through an experimental study. In particular, effectiveness is assessed by increase of the number of faces that are formed in the arrangement by the added curve(s). The efficiency is measured by the running time of the algorithm.

Chapter 2

Related Work

This chapter will introduce related work in fields of automated generation of nonograms and path finding.

2.1 Automated Generation Of Curved Nonograms

In order to understand how to add curves such that they fix the bad faces, it is important to understand how the original curves that make up the puzzle are created. The articles mentioned in this subsection describe the algorithms commonly used to create curved nonograms from vector drawings. It is intended that the added curves will follow a similar procedure, such that they will fit in with the established puzzle.

In his thesis, Tim de Jong [4] provides a basic algorithm to create a nonogram from a set of input curves. A description is given of the characteristics that a "good" curved nonogram should have. The algorithm is mainly focused on creating a puzzle that follows the set rules of the curved nonogram, contains unambiguous curves and obscures the solution picture well enough. The algorithm is not able to specifically produce simple, uniquely solvable curved nonograms. There are parameters that can be tweaked as the user sees fit.

Mees van de Kerkhof [11] provides an improved algorithm for the generation of curved nonograms based on the algorithm created by De Jong. The focus is again set on reducing the ambiguity of the curved and obscuring the solution image best as possible. The algorithm introduces many adjustable features that can help a user receive the desired outcome.

Löffler and Nöllenburg [9] describe solutions for adding optimal labeling to curved nonograms. The labeling should be crossing-free, balanced and compact. The authors provide several methods that would provide these characteristics to the labeling within polynomial time.

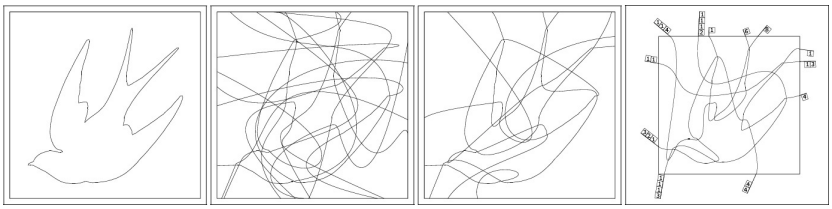


Figure 2.1: A visual representation of the process of the automated generation of a curved nonogram. The first image is a vector drawing as input. The second image shows how the curves of the vector drawing are extended to turn it into a valid curved nonogram. In the third image, this result has been de-cluttered. The final image shows the final curved nonogram, including labeling. (Image from [8])

2.2 Path Finding Algorithms

In this thesis, we want to add a curve to an already existing curve arrangement. We want this added curve to fit in with the other curves of the arrangement. Thus we want it to enter and exit through the border of the puzzle. Another requirement is that we want the added curve to pass through one or more desired faces, namely the popular faces of the arrangement. In order for this to happen, we will base the algorithm on two path finding algorithms. The first is a simple Breadth First Search. The second is an approach introduced by Björklund et al. [2]

Breadth First Search, or BFS for short, is a path finding algorithm that is widely used by programmers for multiple purposes. It is easy to implement and has a time complexity of $O(|V| + |E|)$ for a graph $G = (V, E)$. We will assume that readers are familiar with the approach, but more about BFS can be found in [1].

In their article, Björklund et al. provide an algorithm that finds the shortest closed walk in a graph through a specified set of vertices. This algorithm runs in $O(2^k \times n^2 \times l)$, for which k is the number of specified vertices, n is the total number of vertices in the graph and l is the length of the cycle. Shortcuts could be made to make it run faster, however. This algorithm runs in polynomial time for the input size, thus is fixed-parameter tractable. For this reason, we will refer to this approach as the FPT approach from hereon out.

In the case of curved nonograms, every face and the border should be represented by a vertex of a graph $G = (V, E)$. Faces that border one another will be connected by an edge in E. Eventually, the bad faces and the border should be included within the set of K vertices. We will go more in depth on this type of graph representation in chapter 4. The goal is to find the shortest cycle within graph G that would connects all vertices of set K.

Chapter 3

Definitions

In this chapter, we will discuss in depth the definitions of some of the more important terminology that will be used throughout this paper.

3.1 Nonograms

Classic Nonogram. A nonogram consists of a grid of squares with a description for each row and column of the grid. The faces of a nonogram can be colored either black or white. We refer to white faces as being empty and black faces as being filled in. Every row and column contains a sequence over these colors. A sequence can be represented as a string over the possible colors $C = \{b, w\}$ with b being black, thus filled in, and w being white, thus left empty. For example, for a sequence s of length l we say $s \in C^l$. We define S as the collection of all sequences of the nonogram. There is a sequence for every row and column, thus for a nonogram with a grid of $n \times m$, $|S| = n + m$. The description of a row or column provides a string of integers that explain the sequence of that row or column to the puzzler. Each sequence s_i corresponds to a description d_i. A description is of the form $d_i = c_1 c_2 ... c_k$ in which c_i is a positive integer. Sequence s_i adheres to description d_i when it is of the form $s_i \in w^* b^{c_1} w^+ b^{c_2} w^+ ... b^{c_k} w^*$. In this representation, w^* denotes a string of zero or more w, and b^c denotes a string of c times a b. There are k such strings in the sequence. Each string of b characters is separated by one or more w characters, represented in the sequence as w^+.

Curved Nonogram. A curved nonogram is a variation on the usual nonogram that uses curves instead of a grid structure. A curved nonogram consists of a set of curves enclosed by a border. We will call these set of curves *puzzle curves*. Puzzle curves need to follow the following three rules in order to form a valid curved nonogram: (1) lie completely within the border; (2) the endpoints of each curve lie on the border; and (3) no more than two curves intersect at the same point. A *background curve* forms a closed curve that falls completely

within the border of the puzzle. It will have no space to place descriptive labels, but will influence the faces formed by the set of curves. The curves we will add using the algorithm will be puzzle curves. We will refer to the boundary, puzzle curves and background curves combined as the full curve arrangement.

Instead of the row and column descriptions that the classic nonogram has, the curved nonogram has a description for both sides of each puzzle curve. The description gives information about the sequence of faces that are incident to that side of the curve. Similarly, each sequence had a corresponding description. The description will, similarly to the classic nonogram, give information on the consecutive strings of filled in faces within the sequence.

In a curved nonogram that contains popular faces, it is possible for the same face to be represented in the sequence multiple times.

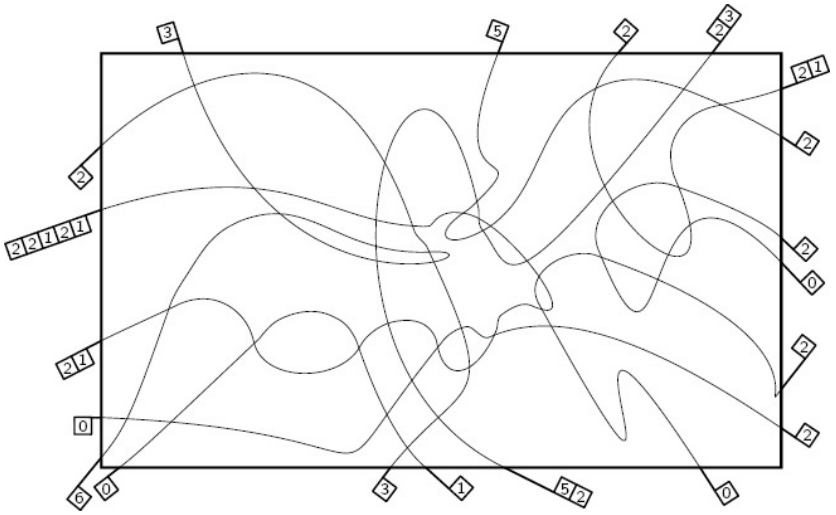


Figure 3.1: A curved nonogram with labeling. (Image from [8])

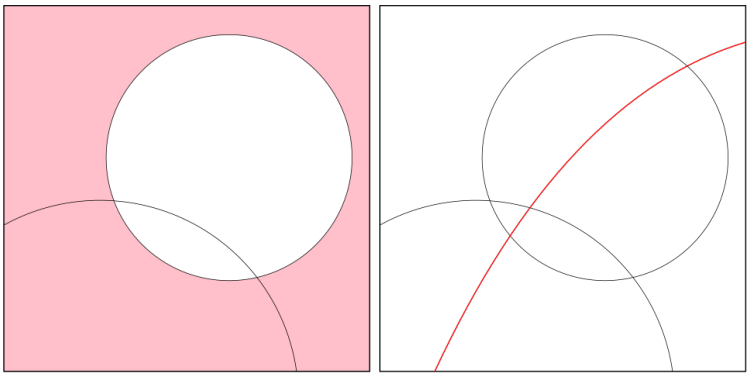
3.2 Curve Arrangement

We define a graph as $G = (V, E)$, where V is a set of vertices and $E \subseteq V \times V$ is a set of undirected edges. If vertex $v \in V$ is an endpoint of an edge $e \in E$, then we say that e is incident to v. If an edge $e = (u, v), e \in E$, then we call v and u adjacent in G. The neighborhood of $N_G(v)$ is the set of neighbors of v in G. Let E_v be the set of all edges that are incident to vertex v. Then $\delta(v) = |E_v|$ is the degree of v. A curved nonogram can be represented as a collection of curves. We will refer to the representation of a curved nonogram as a *curve arrangement*. A curve arrangement can be seen as an undirected planar graph. In such a case, the edges would be drawn using curves, and all intersections between curves would be represented by a vertex.

A drawing Γ of G is a mapping of all vertices to points on a plane and all edges to curve segments connecting the corresponding end vertices. The graph

is planar if it has a drawing where the edges do not intersect, except at vertices where they are incident. We assume that no more than two curves cross at the same point. A curve arrangement A partitions the plane into *faces*, similar to a drawing Γ with curved edges. The set F of curve arrangement A contains all faces of the curve arrangement, including the *outer face*. The outer face is the space on the plane that exists outside of the outer boundary. A curve l is a continuous mapping $l : [0, 1] \leftarrow \mathbb{R}^2$. We use $l[a :]$ to refer to a *subsegment* of the curve l that starts at a. Likewise, we use $l[: b]$ for the subsegment that ends at b, and $l[a : b]$ for the subsegment that starts at a and ends at b. In these notations, we include a and b within the subsegment. If we do not want to include these, we use $l]a : b[$ instead. Since no three curves are allowed to cross at the same point, we can say that the crossing point a of curves l and l' contains subsegments $l[a :]$, $l'[a :]$, $l[: a]$ and $l'[: a]$. The graph $G(A) = (V, E)$ is the associated graph of an arrangement A, if $v_p \in V$ for every intersection point p of two curves $l, l' \in A$, and $(v(p), v(p')) \in E$, if and only if, there exists a curve $l \in N$, such that $p, p' \in l$ and there is no $v(p'') \in V$, with $p'' \in l]p : p'[$.

Popular Face. Let δf be the set of sub segments bounding a face f. We will call this the *boundary* of f. Face f can be bounded by multiple sub segments of the same curve l. We say that l occurs k times in δf, if k is the number of different sub segments of l bounding f. If there is a curve l for which this value $k \geq 2$, we say that f is a *popular face*. We say that a set of curves L *resolves* or corrects a face in an arrangement A if $A \cup L$ does not contain any popular faces. See Figure 3.2 in the Introduction for an example of an arrangement with popular faces and a curve that would resolve these.



(a) (b)

Figure 3.2: An example of a simple curve arrangement with popular faces, marked in pink (a). An example of a curve that could be added in red to solve these particular faces (b).

Dual Graph. The *dual graph* is the representation of the drawing of a graph which also takes the form of a graph itself. Let us call the dual graph $G^* = (V^*, E^*)$. In this graph G^*, the faces of the drawing Γ are represented by vertices

V^*. If and only if a face of Γ is adjacent to another face of Γ, we add an edge $e^* \in E^*$ between the vertices $v^* \in V^*$ of these faces. An example of a curve arrangement and its dual graph can be seen in Figure 3.3. The outer face is included and thus will be represented by a vertex as well. We show this in the Figure through the blue dashed lines. Faces bordering the boundary will have its vertex connected to the vertex of the outer face by an edge.

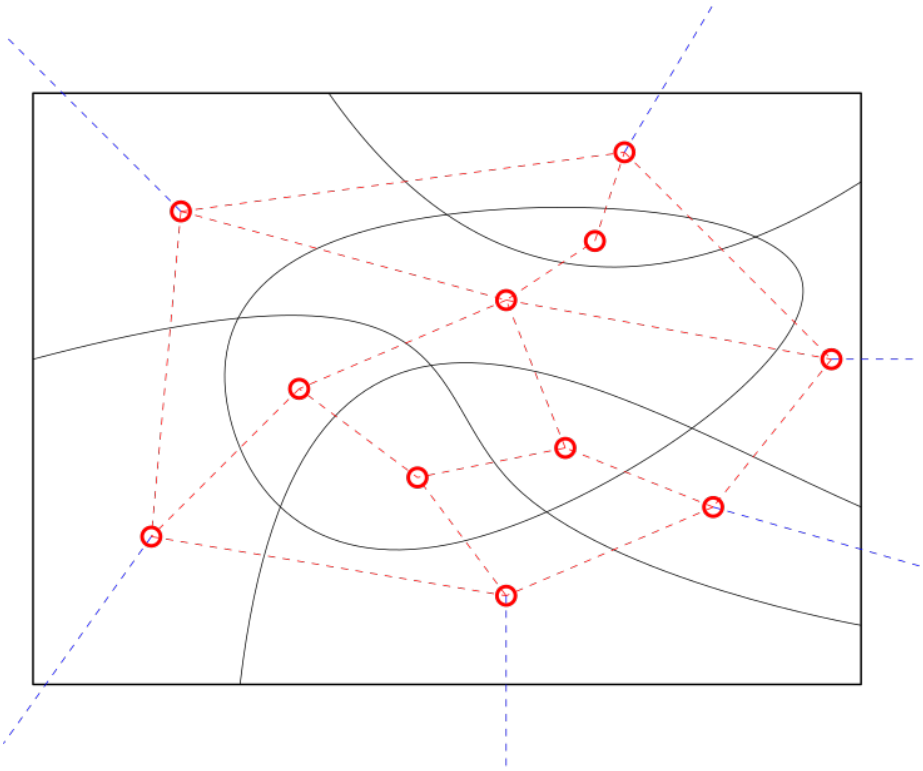


Figure 3.3: An example of a curve arrangement inside a rectangular boundary in black and its dual graph in red. The circles represent the vertices for each face, and the dashed lines connect the vertices of the faces that share an edge. The blue dashed lines connect the vertices to the outside of the bounding box.

Path. We define a *path*, sometimes also referred to as a walk, as a sequence of edges that joins a sequence of adjacent vertices. We call it a *closed* path or walk when the starting vertex and ending vertex of the path are the same. A closed path is also known as a *cycle*. A path is *simple* when no vertices appear in it more than once.

Bézier Curves. In the algorithm, (segments of) curves in the curve arrangement will be formed with *Bézier curves*. Bézier curves are a type of curves that are formed using control points. The Bézier curves we will work with will be

of either the first, second or third degree. We assume that readers are familiar with the concept of Bézier curves. More on Bézier curves can be found in [7].

3.3 Correcting a popular face by adding a curve

As described previously in the introduction, we want to correct a popular face by adding an extra curve to the curve arrangement. Let us again assume we have a curve arrangement A with popular face F_p. Curve c_p visits face F_p twice as edges e_1 and e_2 of F_p. To correct F_p, we need to add one curve c_i, such that edge e_1 is on one side of c_i, and e_2 is on the other. If there is only one edge in between e_1 and e_2 on both sides, then let us call these edges e_l and e_r for clarity. Schematically, we can represent this case by a square, as can be seen in Figure 3.4. In this Figure, edges e_1 and e_2 have the same colour, namely red. Edges with different colours do not belong to the same curve. In the Figure, e_1 and e_2 are connected only by e_l and e_r. In other cases, more than one edge could lie between e_1 and e_2 on either side. Let us define the sets of edges between e_1 and e_2 of either side as E_l and E_r. Sets E_l and E_r can thus contain one or multiple edges.

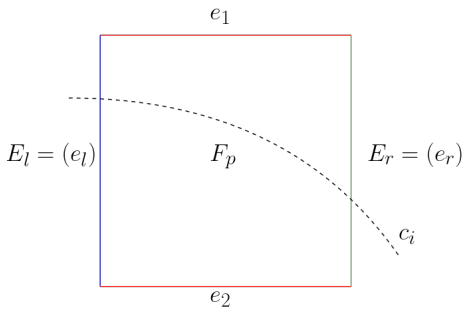


Figure 3.4: Example of popular face F_p with edges $E = (e_l, e_r, e_1, e_2)$ with the curve c_i passing through as the dashed line.

We will add terminals on edges that c_i must pass through, such that e_1 and e_2 will be on opposite sides. A *terminal* is a point on an edge of a face through which curve c_i must enter or exit the face. Topologically, it does not matter where on the edge the terminal is placed. It is important, however, that the edges on which the terminals are placed are chosen such that the popular face will be corrected. From this point onward, we will thus refer to the placement of a terminal with the edge it will be placed on. In the case seen in Figure 3.4, terminals can be added to e_l and e_r. This terminal placement can be seen in the gray dots in Figure 3.5.

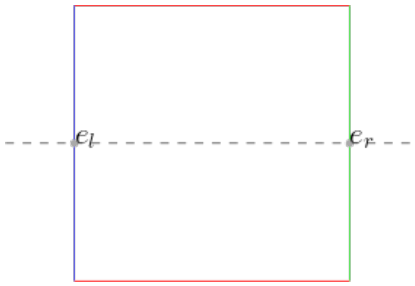


Figure 3.5: Terminals are placed on edges e_l and e_r as explained in the main text.

E_l and E_r could contain multiple edges, and this would result in multiple possible placements for the terminals. A solution would be to use one or two dummy vertices in the dual graph of the curve arrangement. From our curve arrangement we extract a dual graph $G^* = (V^*, E^*)$ with a vertex $v_p^* \in V^*$ for the popular face f_p. The faces that are incident to f_p in the curve arrangement with regards to set E_l will be referred to as set V_l^*, and the faces that are incident to f_p with regards to set E_r will be referred to as set V_r^*. Vertex v_p^* is connected to set V_l^* with edges that form a set E_l^*, and to set V_r^* with edges that form a set E_r^*. When working with dummy edges, we want to remove sets E_l^* and E_r^* from the dual graph. We then add in dummy vertices v_l^* and v_r^*, and connect each of these to v_p^* with an edge. After, we connect all vertices in set V_l^* to v_l^*, and connect all vertices of set V_r^* to v_r^*. This process is visualised in Figure 3.6.

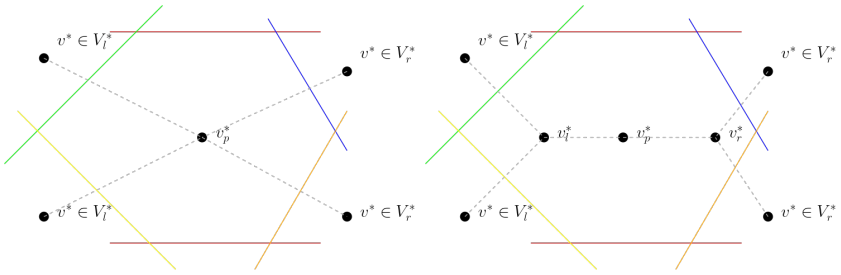


Figure 3.6: As explained, (a) shows the original section of the dual graph around vertex v_p^*, as it is connected directly to the other vertices around it. In (b) we can see the dummy vertices v_l^* and v_r^* and how they change the connections between v_p^* and its surrounding vertices.

Once a shortest path is found, we can remove the dummy edges from the sequence. The benefit to this approach is that we do not have to keep track of edge sets during the path finding, as they are contained within the dummy vertices. The drawback is that adding dummy vertices to a dual graph will increase the number of its vertices and edges, which might make traversing the graph more time expensive.

3.3.1 Finalising Added Curve

In order to finalise the arrangement, we want the curves to run through all their desired terminals, and enter and exit the bounds of the puzzle. For this goal, we will use the dual graph of the curved nonogram and a path finding algorithm.

We want the added curve to enter and exit through the boundary of the curve arrangement. To do this, we can construct a path in the dual graph from the outer face, to the popular face, and back to the outer face. It would thus form a closed path from and to the outer face. Once we have computed the path, we can let the added curve pass through all faces as described by the path. Note that we want the path to be simple in order to avoid creating new popular faces. We also want it to be as short as possible, such that we do not disturb the original arrangement to much. We want to create the smallest number of new faces. It would disrupt the original structure of the puzzle and create more faces to annotate with hints. Furthermore, more faces in an arrangement usually means that faces will be smaller, which can be confusing for the puzzler.

Chapter 4

Algorithms

The program that will be adapted in the process of this thesis is written in Haskell[5] and makes extensive use of the hgeometry library[10] for the creation of curves. The existing code is able to analyze the curve arrangements given as an input. Another function of the code is to add labels containing the hints of the curved nonogram and output an ipe file of the final result. It is also already able to detect popular faces in the given curve arrangement and mark these in a graphics window on screen (see Figure 4.1). Whenever we work with finite field elements in the code, we use the galois-field library[6].

The aim of this thesis is to create a new module for this program that will implement the heuristics as described above. The result will be an output of a curved nonogram without popular faces in it. In order to generate a curve arrangement file such that it can be read by the program, a generator has been provided. This generator takes a vector drawing as an input, and outputs a curve arrangement.

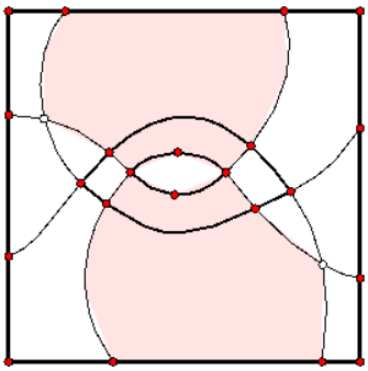


Figure 4.1: A screenshot from the impop program in which the popular faces of the curve arrangement have been marked.

4.1 Input

The input file is an Ipe page that contains a curve arrangement.[3] This arrangement consists of an outer border and curves inside of it. A curve is a chain of Bézier curves. These Bézier curves can be either of the first, second or third degree. The chain of Bézier curves must be continuous, in the sense that a Bézier curve's end point must line up with the begin point of the next one in the chain. The outer border is distinguished in the arrangement by being blue in color. Bézier curves are black or red. The difference between black and red Bézier curves is that the red ones must stay fixed in place when running certain functions within the impop program that can move the control points of the curves. Black Bézier curves are allowed to be moved by these functions. Our algorithm will not make use of such functions, thus this difference is of no concern to us and can be ignored. A chain of Bézier curves must begin and end on the border or form a closed curve. See Figure 4.2 for an example of an input arrangement.

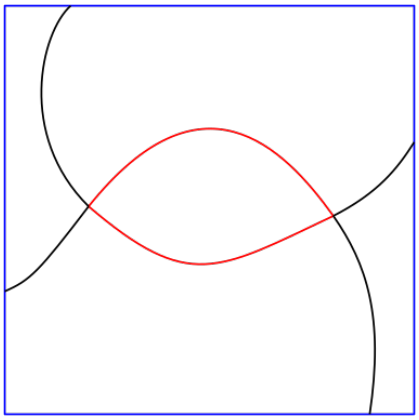


Figure 4.2: An example of an Ipe input for the impop program. The border is colored blue and the curves are black and red.

4.2 Initialising Arrangement for Usage

Once the input has been read by the program, we can extract data from the curve arrangement. One of the first steps is to create a dual graph from this data. We do this by first going over each face within the curve arrangement and search which faces are its neighbors. For each combination of a face and a neighboring face, we can create the representation of an edge. Thus, if faces f_1 and f_2 are neighbors, we can expect there to be an edge (f_1, f_2) and an edge (f_2, f_1). The dual graph will be represented in the code as a list of all edges in the dual graph. Note that the outer face, outside the boundary of the puzzle, is also represented as a face in this dual graph, and will thus also contain edges connecting to it. The outer face is represented as f_0.

We find popular faces in the curve arrangement by tracing each curve both sides. We save faces that appear twice or more in the sequence within a list of all popular faces in the arrangement.

4.3 Inserting a Curve

To add a new curve in the existing curve arrangement, we first want a path through the dual graph. The path will be represented by consecutive edges. Thus, if the faces f_1 and f_3 were to both be neighbors of face f_2, we could create a path P of edges $P = (f_1, f_2), (f_2, f_3)$. In order to comply with the rules of the puzzle curves as described in section 3.2, We want a path that starts at the outer face and ends at the outer face as well. Thus, we want a path P in the form of $P = (f_0, f_1), (f_1, f_2)...(f_{i-1}, f_i), (f_i, f0)$.

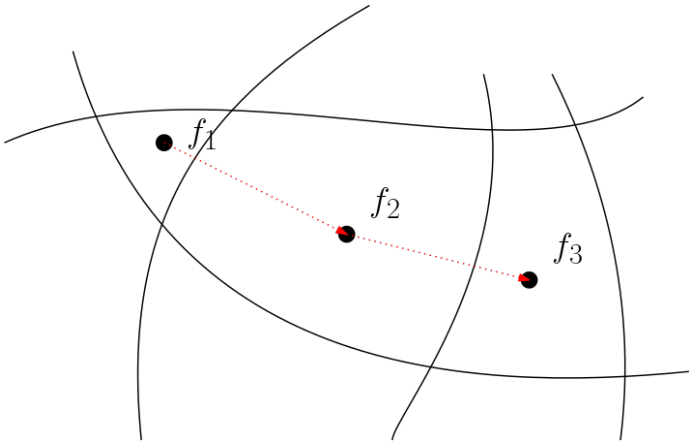


Figure 4.3: An example of the situation in which there is a path $P = (f_1, f_2), (f_2, f_3)$ available in the dual graph of the curve arrangement. The path P is symbolized by the dotted red line. The arrows show the path's direction.

To make sure the curve intersects the correct edges of the face as provided by the path, we first put vertices on the described edges. We find the edge that separates the face in the curve arrangement by looking for the common edges between both faces of an edge in the dual graph. Once we have found the correct edge, we will place a vertex randomly on either its halfway point, or one of its quarter way points. If more than one edge is common between both faces, as can be the case at the corner of the boundary, we will pick randomly between these two as well.

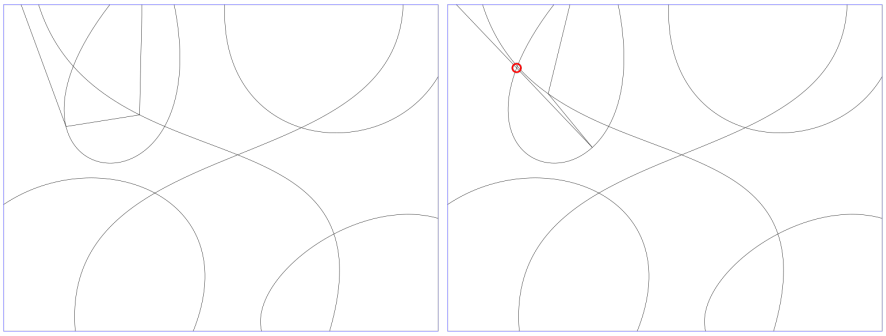
Once all vertices have been placed, we will traverse these in the order in which they have been added and add an edge between one and the next. This process can be observed in more depth in the pseudo code of algorithm 1.

It might happen that an added edge intersects another edge in the arrangement. This is something we would like to avoid. To see if we have created any unwanted intersections, we perform a check before saving our results. We do

Algorithm 1 Curve Insertion

Require: Curve arrangement $A = (V, E)$, path $(f_1, f_2).(f_2, f_3)...(f_{n-1}, f_n)$
  function CURVE INSERTION(A, path)
    $newA = (newV, newE) \leftarrow A = (V, E)$ ▷ Copy of original arrangement
    $i \leftarrow 1$
    for every $(f_x, f_y) \in path$ **do**
      Find corresponding edge $e_{xy} \in newE$ for (f_x, f_y)
      Place vertex $v_{|V|+i}$ anywhere on e_{xy}
      $i \leftarrow i + 1$
    end for
    $i \leftarrow |V| + 1$
    for every vertex $v_i \in newV$ **do**
      Place edge between v_i and v_{i+1}
      $i \leftarrow i + 1$
    end for
  return $newA$
  end function

this by looking up all intersections between every curve in the arrangement. We also look up the locations of all vertices of the arrangement. If there is any intersection point that does not correspond with the location of a vertex, we can conclude that it is an intersection created by an overlap between our added edge and another one, thus making it an unwanted intersection. Pseudo code for the intersection check can be seen in algorithm 2. If this is the case, we restart the curve insertion from the top, meaning that we take back the original curve arrangement and place new vertices according to the given path. Because of the random placement of the vertices, a new iteration might provide a result without any unwanted intersections. The pseudo code for this function can be found in algorithm 3.



(a) (b)

Figure 4.4: figure (a) shows a curve arrangement with an added curve that does not give us any unwanted intersections between edges. It only crosses other edges where there is a vertex. In Figure (b), we see that the added curve does intersect with another curve on a point where there is no vertex. The unwanted intersection in (b) is marked with a red circle.

Algorithm 2 Intersection Check

Require: Curve arrangement $A = (V, E)$
  function INTERSECTION CHECK(A)
    for every $v \in V$ **do**
      $vertexLocations \leftarrow locationofv$
    end for
    for every intersection of every $e \in E$ **do**
      $intersectionLocations \leftarrow$ location of intersecton
    end for
    if any $intersectionLocations \notin vertexLocations$ **then**
  return True
    else
  return False
    end if
  end function

Algorithm 3 Create New Curve Arrangement

Require: Curve arrangement $A = (V, E)$, path $(f_1, f_2).(f_2, f_3)...(f_{n-1}, f_n)$
  function CREATE NEW CURVE ARRANGEMENT(A, path)
    $newA = (newV, newE) \leftarrow CurveInsertion(A, path)$
    if $IntersectionCheck(newA)$ **then**
      $CreateNewCurveArrangement(A, path)$
    else
  return $newA$
    end if
  end function

4.4 Iterative Algorithm

In our first approach, we resolve each popular face in the arrangement one by one. This means that given a list of popular faces in the arrangement, we take the first face of this list and try to resolve it by adding a curve to correct it. Once this curve is added, we will move on to the next face in the list and resolve this one in a similar manner. It is possible that a curve added earlier will accidentally resolve another popular face further down the list. Because of this, we update the list of popular faces after each added curve until it is empty.

For this approach, we focus on finding a path through the dual graph based on a Breadth First Search (BFS) algorithm. What we want to accomplish with our path is as follows: (1) a shortest possible path; (2) it must pass through a popular face in such a manner that it resolves it; (3) it must begin and end on the boundary of the curved nonogram. We accomplish these characteristics in the following manner.

To make sure we pass through the correct edges in order to correct the popular face, we want to make a list of all neighbors of the popular face in either clockwise or counter-clockwise order. We will call the popular face f_p and all its neighbors belong to the set $N(f_p)$. We then look at all edges bordering the popular face and find which ones belong to the same curve. We will refer to these edges as the *bad edges*. We want to find out which faces in $N(f_p)$ share the bad edges with f_p. We will call these faces the *forbidden faces*. This algorithm will only account for cases in which there are two forbidden faces. We will refer to these forbidden faces as f_l and f_r. Out of our set $N(f_p)$, we want to split the faces into two sets regarding their position to the forbidden faces. Every face counter-clockwise between f_l and f_r in the list $N(f_p)$ will be contained in a set of faces F_l. Every face counter-clockwise between f_r and f_l in the list $N(f_p)$ will be contained in set F_r. To visualise this process, see Figure 4.5. We want the path created by our algorithm to contain an edge between one face of F_l and f_p, and one edge between one face of F_r and f_p.

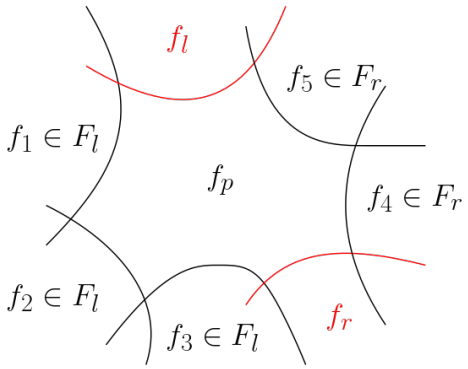


Figure 4.5: In this example, we see a popular face f_p and its neighboring faces. The forbidden faces f_l and f_r have been colored red. Neighboring faces $N(f_p) = f_l, f_1, f_2, f_3, f_r, f_4, f_5$ in counter-clockwise order. We can split $N(f_p)$ into two sets F_l and F_r by cutting it at f_l and f_r. This would result in $F_r = f_4, f_5$ and $F_l = f_1, f_2, f_3$.

To find the shortest path from the border to a face in F_l, we run BFS with the outer face as the starting point. We compute paths from the outer face to all faces in the set F_l and return the shortest out of these. We then add an edge between the last face of this path and the popular face f_p. We will call this path P_1. Next, we run BFS to compute shortest path from a face in F_r to the border. We compute the shortest paths from each face within set F_r to the outer face and pick the shortest path out of these. We make sure we do not cross any faces we have already visited in path P_1 by deleting the edges used in this path from the dual graph over which we run the algorithm this time. See Figure 4.6 for an example of how this process would work.

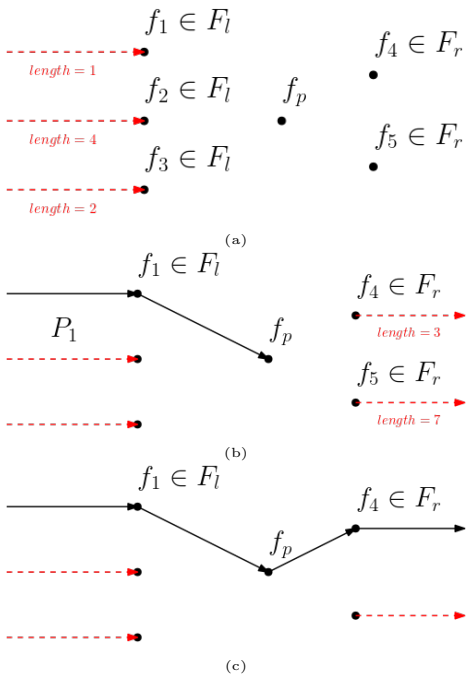


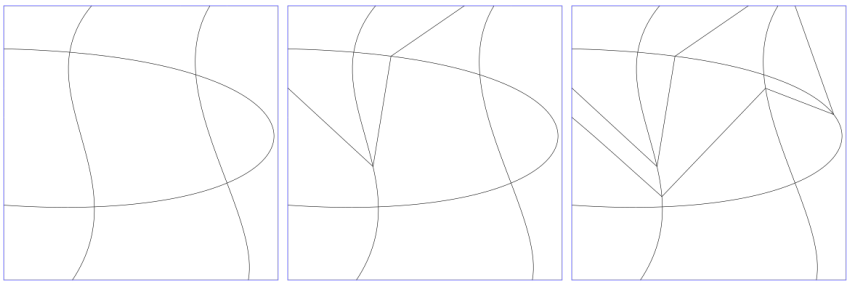
Figure 4.6: In (a), we have run BFS on all faces in F_l. Paths going to these faces are shown in dashed red lines. The arrows indicate the direction of the path. The lengths of the resulting paths are shown in red. In (b), we pick path P_1 to be the shortest path to a face in F_l. In this specific example, f_1 has the shortest path. Path P_1 is shown in black. We now run BFS on the faces in F_r. Again, these paths are marked with a red dashed line and their lengths are shown in red. In (c), we connect the established path to shortest path from a face from F_r. Face f_4 has the shortest path in this example. We continue the path by extending the black line.

4.5 FPT Algorithm

In the FPT approach we make use of dynamic programming and finite fields. We will refer to the finite field as Φ, and to each element from field Φ as ϕ. We also us the dummy face method described in the previous chapter. Through the FPT approach, we will be able to find the shortest cycle through a set of specified edges if one such path exists. We will call the set of edges we want our path to pass through set K. We want set K to be defined such that the final path passes through all popular faces of the curve arrangement in a manner that would resolve them all.

4.5.1 Initialisation

We find the dual graph of the given curve arrangement. We save its popular faces too. We adapt the dual graph such that it includes two dummy faces per popular face. If popular face f_p is connected to two dummy faces f_l and f_r, we make sure that the dual graph includes edges (f_p, f_l) and (f_p, f_r). Similarly, we want the faces f_l and f_r to be connected to the correct neighbors of f_p.



(a) (b) (c)

Figure 4.7: An example of the process of the iterative approach. We start from the original input figure (a), and add curves one by one. figure (c) is the final result.

We initialise a table of edges in the dual graph and a random element from the finite field for each of them. We will refer to the finite field element assigned to an edge as $\phi(e)$, with e being its respective edge. The finite field value of a path is the product of the finite field values of its edges. This results in the finite field value of a path w being

$$\phi(w) = \prod_{e \in w} \phi(e) \tag{4.1}$$

Through the characteristics of a finite field, the finite field value of a path is $\phi(w) = 0$ if an edge appears in the path more than once. The path is a simple path if its finite field value is not 0.

We decide a set K of edges we want our final cycle to pass through. These edges consist of the dummy faces connected to their respective popular faces. We define length k as $|K|$. The starting point is the outer face f_0. We set the length of our desired cycle $l = k + 2$.

We now run the dynamic algorithm. A valid solution is one for which (1) all edges in set K have been visited; (2) the finite field value is not 0; and (3) its final edge leads to the starting point f_0, thus making it a cycle. If no valid solution is found with for length l, we increment it by $+1$ until $l = |E|$ of the dual graph. If there is no solution for $l = |E|$, there sadly is no simple cycle through all edges of K and the starting point f_0.

4.5.2 Dynamic programming

The dynamic algorithm makes use of table T, which contains the following data per entry: (1) a subset of set K, which we will refer to as set S, containing edges form K which the path has passed through; (2) the current length r of the path; (3) edge e_1, which is an edge from our starting point f_0 to any of its neighbors in $N(f_0)$; (4) edge $e_r \in E$. We set the resulting value of the entry of the table as

$$T(S, r, e_1, e_r) = \sum_{w \in W_r} \phi(w) \tag{4.2}$$

in which the sum is taken over all paths $p = e_1, e_2...e_r$ for which $e_1 = (f_0, f_i)$ where $f_i \in N(f_0)$.

We initialise the table by setting it up as:

$$T(\{\}, 2, (f_0, f_i), (f_i, f_0)) = \phi(f_0, f_i)\phi(f_i, f_0) \tag{4.3}$$

for which f_i is any face in $N(f_0)$. We then set $r = 3$. We keep incrementing r by 1 until $r = l$. We update table $T(S, r, e1, er)$ for every e_1 and e_r in E and every $S \subseteq K$ as explained below. If $e_r \in K$ and $e_r \in S$, then:

$$T(S, r, e_1, e_r) = \phi(e_r) \sum_{e_{r-1} \in E} T(S - \{e_r\}, r - 1, e_1, e_{r-1}) \tag{4.4}$$

If $e_r \notin K$, then:

$$T(S, r, e_1, e_r) = \phi(e_r) \sum_{e_{r-1} \in E} T(S, r - 1, e_1, e_{r-1}) \tag{4.5}$$

For any e_r which is the reverse of e_{r-1}, we cut out its table entry as a form of optimization. These table entries would result in a finite field entry of 0, which we do not desire. These types of entries are very easy to find and cutting them out will avoid unnecessary branching in the table, thus speeding up the algorithm.

Once we reach $r = l$, we scan the table for valid solutions. If a valid solution is found, we return its table entry, thus providing us with the length of the shortest cycle r and the first and final edge of the cycle, e_1 and e_r respectively. Using this information, we can backtrack our desired path.

4.5.3 Backtracking

To backtrack through the result of the dynamic algorithm, we use an oracle which tells us for graph G, an edge in G, a set of edges R and a length l, whether there exists a solution. We use this oracle to construct an explicit path as explained below.

1. We pick any first edge e that is incident to vertex a from G.
2. We check if e is in set K. If so, we check whether $e \in R$. If $e \in R$, we update R to $R - \{e\}$. If $e \notin R$, then we skip edge e and pick a new starting edge.
3. We delete e from G.
4. Now we run the oracle on the modified graph G. If the oracle says there still exists a path of length $l - 1$ from edge e, we picked a good edge. We recurse from step 1 with our modified graph, from the second vertex of e with length $l - 1$ and a modified set R if needed. If there exists no such path according to the oracle, we guessed wrong and revert the changes made.

The pseudo code for the backtracking method can be found below in algorithm 4.

Algorithm 4 FPT Backtracking

Require: Dual Graph G, length l, set K, first edge e_1 and final edge e_r
  $R \leftarrow K$
  for Edges $e_i \in N(e_1)$ **do**
    if $e_i \in K$ **then**
      if $e_i \in R$ **then**
        $R \leftarrow R - \{e_i\}$
      else Skip edge e_i
      end if
    end if
    Run oracle for length $l - 1$
    if Solution is found **then**
      $G \leftarrow G - e_i$
      $l \leftarrow l - 1$
      Recurse for neighbors in $N(e_i)$
    else Skip edge e_i
    end if
  end forreturn Path w

Once the path has been returned, we apply the curve insertion to the curve arrangement.

Chapter 5

Experiment and Results

5.1 Experimental Setup

In this section, we explain the setup of the experiment. We first discuss the test set, followed by the score components, and finally the performance measurement. We aim to compare the Iterative approach and the FPT approach by running both algorithms on a set of curve arrangements containing popular faces.

Test Set. The test set consists of small curve arrangements with varying numbers of faces and popular faces. The test set can be found in appendix A, Figure A.1. We define each input image by its initial number of faces and number of popular faces. We define its *complexity* as the percentage of how many of its faces are popular. Characteristics of the complete test set can be found in Table 5.1 below. For each test in the test set, a shortest path that resolves all popular faces exists.

5.1.1 Score Components

We run the Iterative algorithm and the FPT algorithm on all tests in the test set. Once all popular faces have been resolved, we save the resulting curve arrangement. For both results, we calculate the new number of faces. We define the *face surplus* as the increase in faces compared to the original curve arrangement. We differentiate between a *nominal face surplus*, which is represented as the number of faces by which the curve arrangement has increased, and a *relative face surplus*, which is represented as the increase in number of faces as a percentage of the original number of faces. A small face surplus is desirable.

5.1.2 Performance

We measure the performance of both algorithms by its running time. The running time includes the insertion of new curves into the curve arrangement. This excludes, however, the time saving and loading curve arrangements into

Test Characteristics			
	original faces	popular faces	complexity
Test 1	8	2	25%
Test 2	10	1	10%
Test 3	9	3	33.33%
Test 4	8	2	25%
Test 5	15	5	33.33%
Test 6	14	2	14.29%
Test 7	6	2	33.33%
Test 8	10	3	30%
Test 9	13	4	30.77%
Test 10	13	2	15.38%
Test 11	15	4	26.67%
Test 12	13	2	15.38%
Test 13	17	5	29.41%
Test 14	15	4	26.67%
Test 15	14	3	21.43%
Test 16	17	6	35.29%

Table 5.1: Characteristics of the test cases.

ipe files and into the program. We keep track of the time the FPT algorithm took to compute a path separately as well. All tests are run on an Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz 2.80 GHz.

5.2 Results

In this section, we discuss the test results. We compare the test scores and test performance of the Iterative and FPT algorithms. The resulting curve arrangements after running the algorithms can be found in appendix A, Figures A.2 and A.3. Tables and graphs are provided. Extra graphs can be found in appendix B for visualization if needed. These are referenced when applicable. We further discuss any results that are notable.

In graphs, we stay consistent with the coloring. Blue refers to results of the Iterative algorithm and red refers to results of the FPT algorithm.

5.2.1 Test Scores

The test scores can be seen in Table 5.2 below. In appendix B, we have included graph representations of these results for easier comparison. Figure B.1 shows the nominal face surplus and Figure B.2 shows the relative face surplus after running both algorithms. As expected, for most cases the Iterative algorithm gives a similar or larger increase in number of faces compared to the FPT algorithm. Test cases with a small number of popular faces have an increased chance of being resolved by adding a single curve using the Iterative approach, especially if these popular faces are places close together. Naturally, for cases

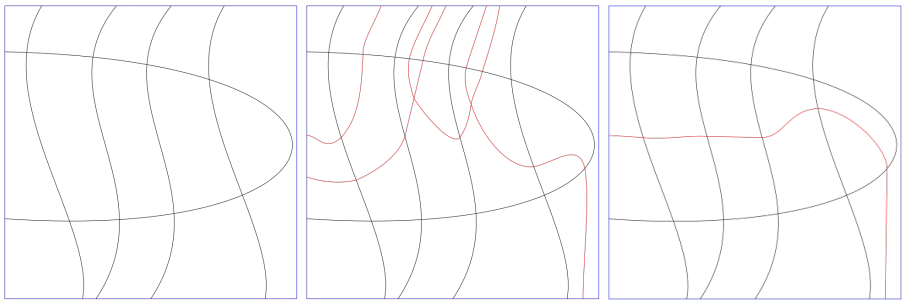
with only 1 popular face the Iterative and FPT approach yields the same face surplus. We can see that for curve arrangements with only 2 popular faces, the face surplus for the Iterative and FPT approaches are also usually the same.

	Iterative Results		FPT Results	
	nominal face increase	relative face increase	nominal face increase	relative face increase
Test 1	3	37.5%	3	37.5%
Test 2	3	30%	3	30%
Test 3	7	77.78%	4	44.44%
Test 4	3	37.5%	3	37.5%
Test 5	17	113.33%	6	40%
Test 6	3	21.43%	3	21.43%
Test 7	3	50%	3	50%
Test 8	6	60%	5	50%
Test 9	6	46.15%	6	46.15%
Test 10	7	53.85%	4	30.77%
Test 11	12	80%	5	33.33%
Test 12	4	30.77%	4	30.77%
Test 13	17	100%	6	35.29%
Test 14	9	60%	5	33.33%
Test 15	6	42.86%	7	50%
Test16	26	152.94%	7	41.18%

Table 5.2: Nominal and relative face surplus for each test case after running both the iterative and FPT algorithm on them.

Similarly, a curve arrangement with many popular faces yields a higher face surplus after applying the Iterative algorithm compared to the FPT algorithm. As the test cases with the most popular faces, Test 5, 13 and 16 yield a much higher face surplus for the Iterative approach. The difference in the final total number o faces is apparent when comparing the results. In Figure 5.1, we can see Test 5 and its results. The Iterative result contains more added curves, thus adding to the face surplus. The FPT approach resolves the popular faces with a single curve, thus disrupting the original arrangement a lot less. The relations between the initial number of popular faces and the difference in face surplus can be seen in Figure B.3 in appendix B. Figure B.4 in appendix B shows the relation between complexity and the difference in relative face surplus.

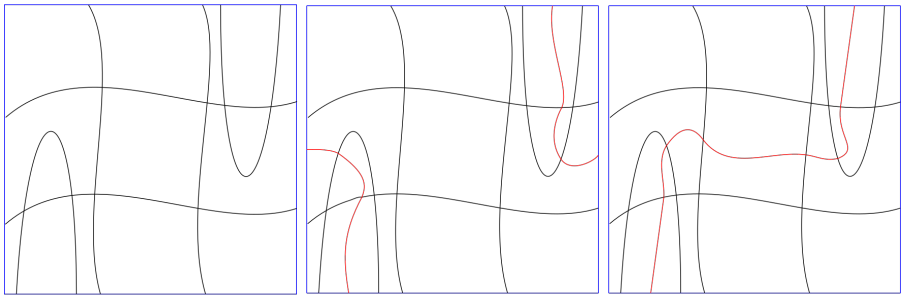
An unexpected result can be seen for Test 15. This is the only case in the test set for which the face surplus of the Iterative approach is lower than that of the FPT approach. The reason for this is the distance between the popular faces in the arrangement. The popular faces on the left and right side of the curve arrangement are resolved separately from one another using the Iterative algorithm. We would enter the arrangement from the boundary, resolve the popular faces of one side, and exit through the boundary again. The same goes for the popular faces on the other side. When using the FPT algorithm, we would need to make a connecting between the paths that would resolve both sides. This would resolve in an added curve that would be longer than the



(a) (b) (c)

Figure 5.1: An example of a test case with a big difference in resulting face surplus. The test case is referred to as Test 5 in the test set. Added curves are marked red. figure (a) shows the input arrangement, figure (b) shows the result of the Iterative approach, and figure (c) shows the result of the FPT approach.

length of the two separate curves together. This example can be seen in Figure 5.2. This is an effect that can also be seen in the results of Test 9.



(a) (b) (c)

Figure 5.2: An example of a test case for which the FPT face surplus is higher than the Iterative face surplus. The test case is referred to as Test 15 in the test set. Added curves are marked red. Figure (a) shows the input arrangement, Figure (b) shows the result of the Iterative approach, and Figure (c) shows the result of the FPT approach.

5.2.2 Test Performance

The running times for both algorithms can be found in Table 5.3 below. This running time includes the insertion of curves into the curve arrangement. Please keep in mind that the curve insertion has a certain dependence on randomness when placing curves. This could result in extra time taken to check for unwanted intersections and recalculating vertex placement after an unwanted intersection is discovered. It does not include time taken loading and saving ipe files into the program.

For easier comparison, Figure 5.3 below shows the running time of both algorithms in milliseconds. From this graph, we can easily conclude that the difference in running time between the test cases is bigger for the Iterative approach. The running times of the FPT approach do not differ as greatly from

	Iterative time	FPT time
Test 1	00:02:26:465	00:00:48:137
Test 2	00:01:00:038	00:01:30:690
Test 3	00:04:18:065	00:01:41:151
Test 4	00:00:50:578	00:02:03:945
Test 5	00:40:17:953	00:07:14:249
Test 6	00:11:56:740	00:05:31:890
Test 7	00:00:28:436	00:00:39:836
Test 8	00:09:05:161	00:02:56:052
Test 9	00:04:58:189	00:02:32:600
Test 10	00:09:29:898	00:14:21:082
Test 11	00:46:02:113	00:05:41:047
Test 12	00:02:47:459	00:02:50:743
Test 13	02:49:09:620	00:15:09:756
Test 14	00:37:30:123	00:23:13:883
Test 15	00:07:29:824	00:16:09:722
Test 16	03:03:05:087	00:04:50:509

Table 5.3: Running time for each test case using both Iterative and FPT approach. Time is shown in hours, minutes, seconds and milliseconds.

one another. We will explain why we believe this is happening.

It seems that especially for cases in which the face surplus is substantially higher for the Iterative approach compared to the FPT approach, the running time of the Iterative approach is also longer. Test cases such as Test 5, 11, 13 and 16 show such results. From Figure 5.4 below, we can make out the relation between a large face surplus and longer running time for the Iterative approach. This is something that can be seen for both the nominal and relative face surplus. Figure B.5 in appendix B shows the relation between running time and relative face surplus. Both Figure 5.4 and B.5 show a similar pattern. The reason for this behavior is that a large face surplus for an Iterative approach is usually due to the insertion of multiple curves. After the insertion of each curve, we need to rerun the process of initializing the resulting curve arrangement, finding a path, and inserting a curve. This process is dependent on the random placement of vertices. A wrong vertex placement would result in a rerun of the curve insertion method. Thus, we might conclude that the use of random vertex placement slows the Iterative approach down for cases with a high face surplus.

For cases with small differences in face surplus, we also see only a small difference in running times. Examples of such test cases are Test 1, 2, 4 and 7. For these test cases, both the Iterative and FPT algorithms only add a single curve to the arrangement. Thus, there is no need for the Iterative approach to rerun its initialisation process, which saves it some time.

In graph 5.3 we can see that the FPT running times do not differ much from one another. As can be seen in graphs 5.4 and B.5, the test cases do not differ much in face surplus either. This could indicate a relation between a small face surplus and short running time for the FPT results.

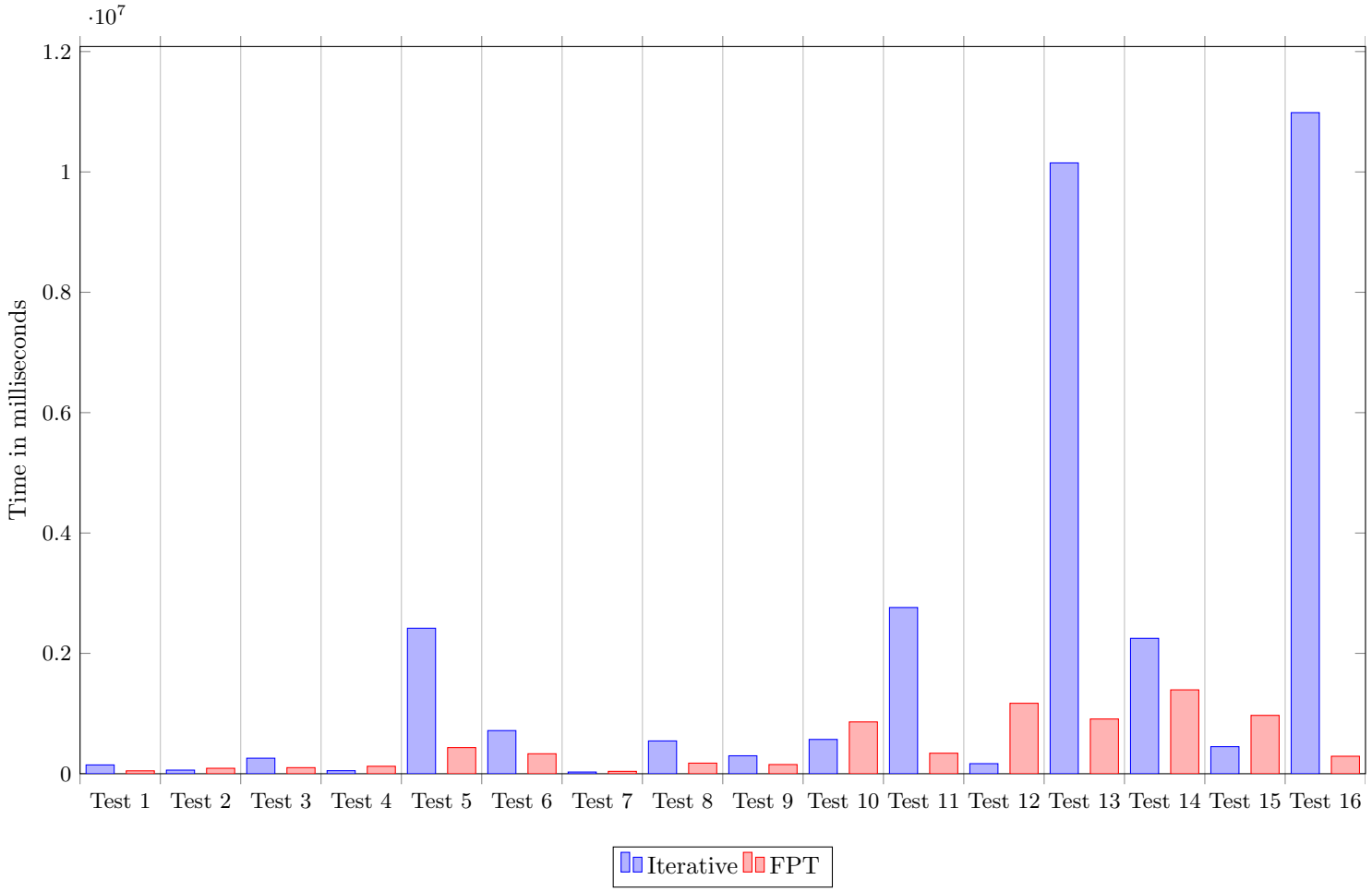


Figure 5.3: Running time for all test cases in milliseconds.

In the section below, we look at the time it took the FPT approach only to compute a path with regards to other characteristics of the test cases and their results.

FPT path time

Table 5.4 below shows the time it took for the FPT approach to only find a path that would resolve all popular faces in the provided test cases.

According to the theory, the FPT algorithm has a running time of $O = 2^k \times n^2 \times l$, for which k is the number of specified elements we want the cycle to run through, n is the total number of vertices in the dual graph, and l is the length of the resulting cycle. We can see the dependence on the size of the set of specified elements from graph 5.5 below. The number of specified elements in

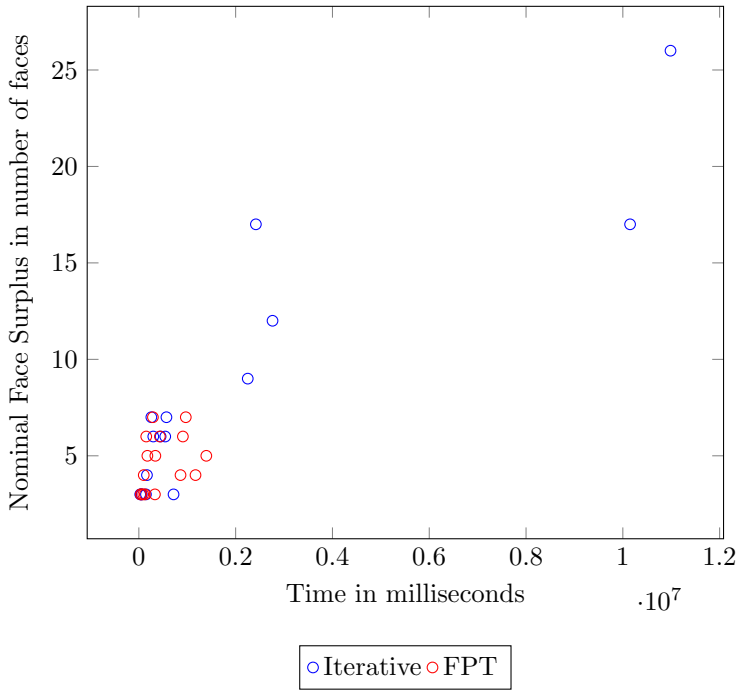


Figure 5.4: Running time in milliseconds and nominal face surplus in number of faces for Iterative and FPT approaches.

our cases depend on the number of popular faces in the original arrangement. We can see that a bigger number of popular faces results in longer running time.

The dependence on number of vertices in the dual graph can be seen in graph B.6 in appendix B, as the number of faces in the original curve arrangement coincides with the number of vertices. Dependence on the length l can be seen in graph B.7 in appendix B. The increase in number of faces after inserting a curve corresponds to the length of the path the FPT algorithm has found. From both graphs, we can find a positive correlation between the values n and l and the running time of the FPT path finding algorithm. We have included graphs comparing the FPT path finding's running time to relative face surplus in Figure B.9 and to complexity of the original arrangement in Figure B.8 in appendix B.

	FPT path time
Test 1	00:057
Test 2	00:045
Test 3	00:101
Test 4	00:063
Test 5	02:029
Test 6	00:424
Test 7	00:012
Test 8	00:241
Test 9	01:357
Test 10	00:303
Test 11	00:958
Test 12	01:738
Test 13	02:934
Test 14	00:899
Test 15	04:640
Test 16	18:626

Table 5.4: Running time for only computing a path using the FPT approach on all test cases in seconds and milliseconds.

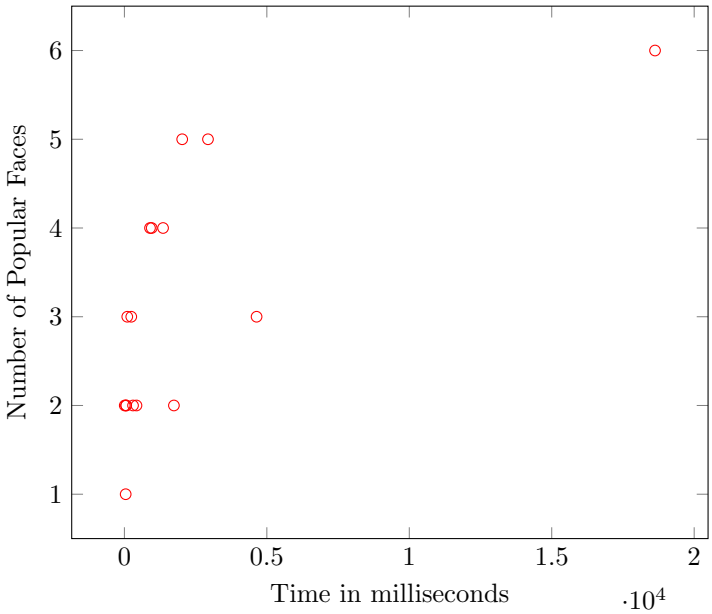


Figure 5.5: FPT path time in milliseconds vs number of popular faces in the original curve arrangement.

Chapter 6

Conclusion

In this thesis, we have introduced two algorithms for resolving popular faces in curve arrangements. We have introduced methods for ensuring that a curve splits a popular face correctly in order to resolve it. We have provided an algorithm that resolves popular faces iteratively, and one that aims to resolve all popular faces in the arrangement at once. We have successfully implemented the FPT search algorithm introduced by Björklund et al. and adapted it in order to search for paths that resolve multiple popular faces with one curve.

We have provided a comparative study on the Iterative and FPT algorithms. We have provided test cases and scored the results and measured performance after applying both approaches. From this study, we were able to conclude that for most cases the FPT algorithm yields the same or a better score than the Iterative algorithm. There were exceptions, however, which we were able to explain. We were able to conclude that the Iterative approach has a longer running time when it needs more curves to resolve the curve arrangement.

6.1 Future Work

Dependence on randomness. The correct insertion of curves is currently very dependent on the random placement of vertices. This increases the running time of the Iterative algorithm especially. Performance measures can be skewed due to the randomness as well. We would like to introduce a manner of vertex placement that does not depend on randomness in order to avoid unwanted intersections.

More complex popular faces. The introduced algorithms are able to resolve popular faces that have two edges that belong to the same curve. There are, however, more complex types of popular faces. These include popular faces that have two edges that belong to one curve, and two edges that belong to another one. Popular faces that have three or more edges that belong to the same curve exist as well. The FPT algorithm is not able to resolve these types of popular

faces, and the Iterative approach is also not able to do so most of the time. An algorithm that would be able to resolve these cases could be developed in the future.

FPT for multiple curves. The FPT algorithm we have introduces is only able to resolve a curve arrangement if only a single curve is sufficient to resolve all popular faces. The FPT algorithm could be altered to run iteratively for cases in which more curves would be needed. The FPT algorithm would need to compute whether a path exists that would resolve all popular faces, then run for subsets of popular faces if this is not the case. Once a subset of popular faces that can be resolved by a single curve can be found, we can rerun the FPT algorithm for the remaining popular faces.

Automatic curvature. Both algorithms add curve sections that do not curve correctly. We would like to be able to automatically curve the curve segments in a manner that is consistent with the rest of the curve arrangement. Thus, we would like the added curves to be continuous.

Bibliography

[1] John B Ahlquist and Jeannie Novak. *Game Artificial Intelligence*. Thomson, 2008, pp. 171–175.

[2] Andreas Björklund, Thore Husfeldt, and Nina Taslaman. "Shortest Cycle Through Specified Elements". In: *Proceedings of the 2012 Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 1747–1753. DOI: 10.1137/1.9781611973099.139. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611973099.139. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611973099.139.

[3] Otfried Cheong. *Ipe extensible drawing editor*. Dec. 2020. URL: http://ipe.otfried.org.

[4] Tim de Jong. "The concept and automatic generation of the Curved Nonogram puzzle". Master Thesis. Utrecht University, July 2016.

[5] *Haskell Language*. URL: https://www.haskell.org/.

[6] Adjoint Inc. *galois-field*. Apr. 2020. URL: https://github.com/adjoint-io/galois-field#readme.

[7] Mike Kamermans. *A Primer on Bézier Curves*. 2020. URL: https://pomax.github.io/bezierinfo/.

[8] Mees van de Kerkhof et al. "Design and Automated Generation of Japanese Picture Puzzles". In: *Computer Graphics Forum* 38.2 (2019), pp. 343–353. DOI: https://doi.org/10.1111/cgf.13642. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13642. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13642.

[9] Maarten Löffler and Martin Nöllenburg. "Labeling Nonograms". In: Mar. 2020.

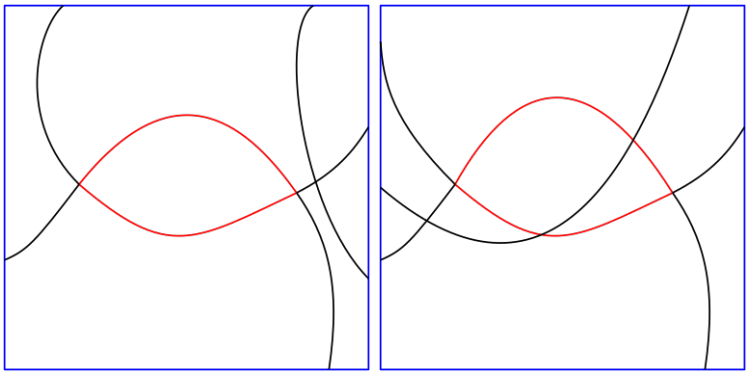
[10] Frank Staals. *hgeometry*. Apr. 2021. URL: https://fstaals.net/software/hgeometry/.

[11] Mees van de Kerkhof. "Improved Automatic Generation of Curved Nonograms". Master Thesis. Utrecht University, Nov. 2017.

Appendix A

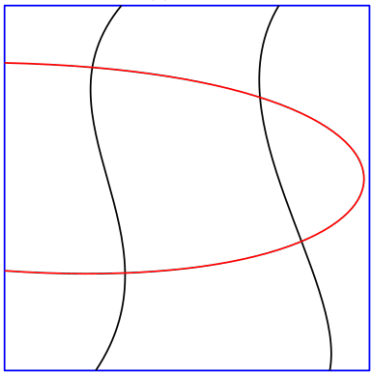
Test Set and Results

A.1 Test Set

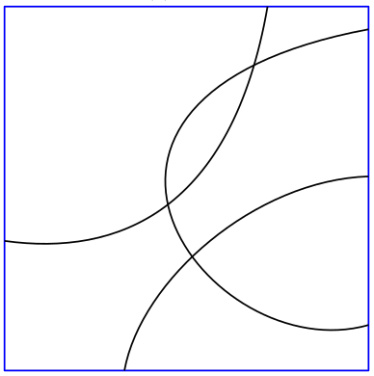


(a) Test 1

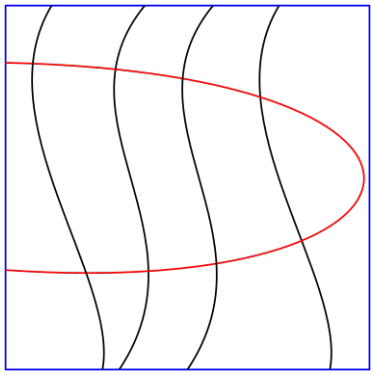
(b) Test 2



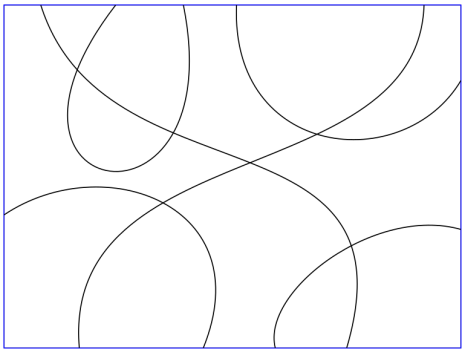
(c) Test 3



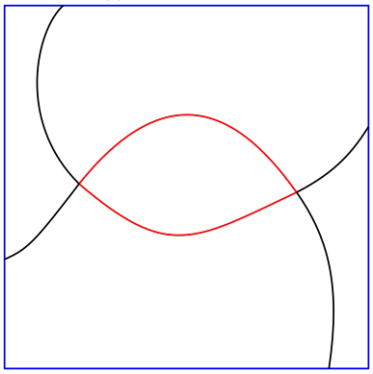
(d) Test 4



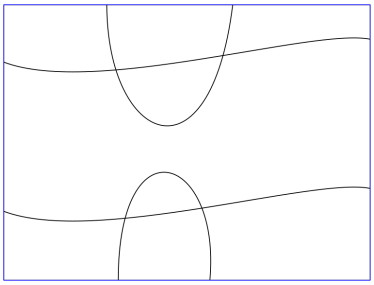
(e) Test 5



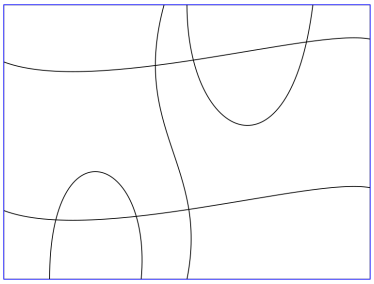
(f) Test 6



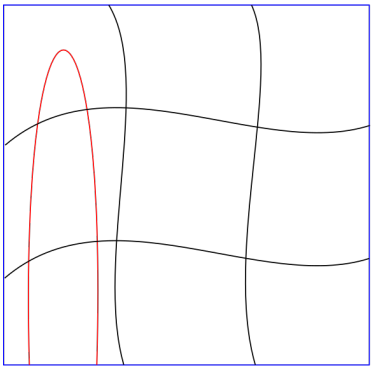
(g) Test 7



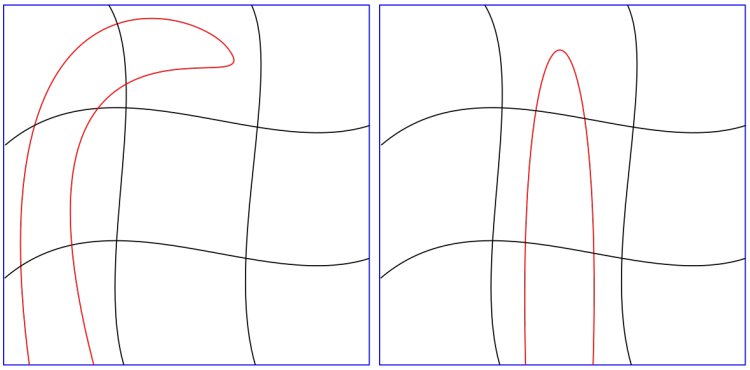
(h) Test 8



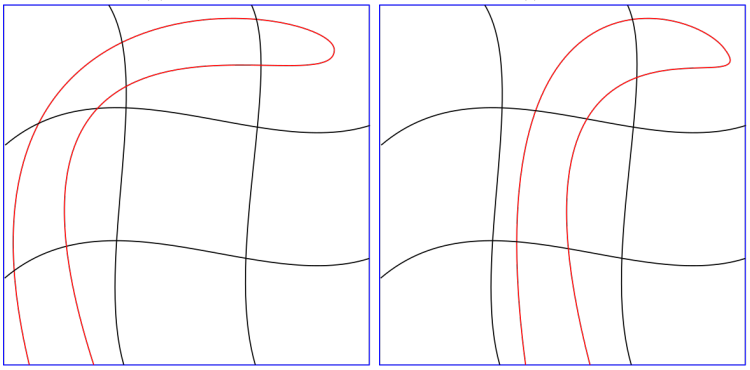
(i) Test 9



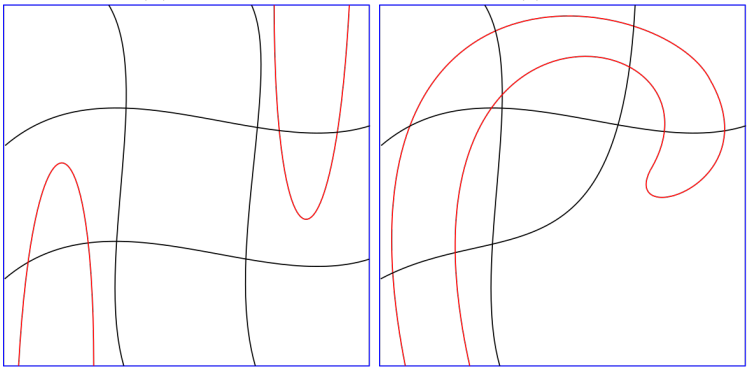
(j) Test 10



(k) Test 11 (l) Test 12



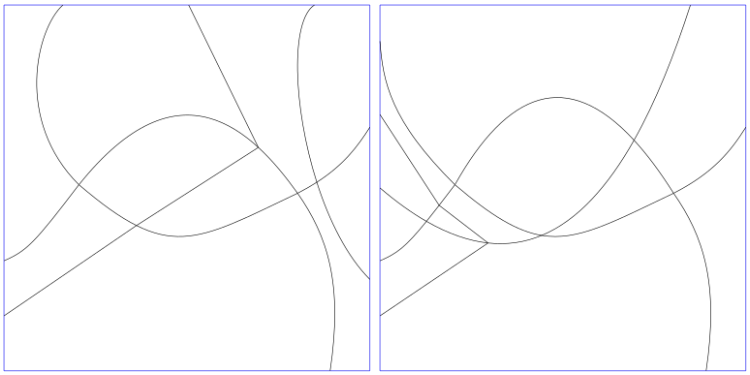
(m) Test 13 (n) Test 14



(o) Test 15 (p) Test 16

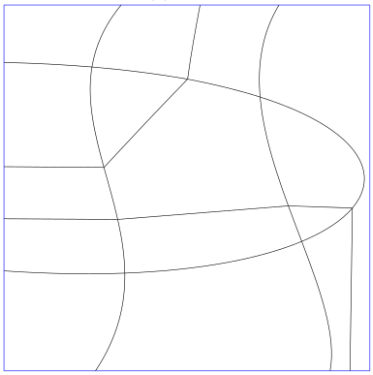
Figure A.1: Test Cases

A.2 Iterative Results

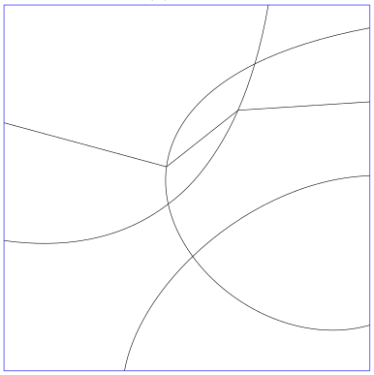


(a) Test 1

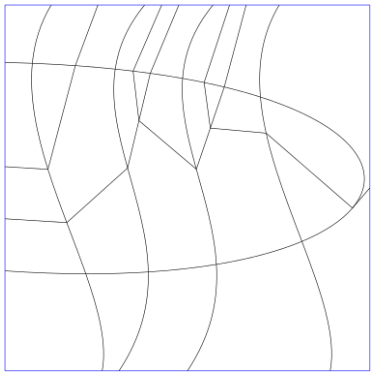
(b) Test 2



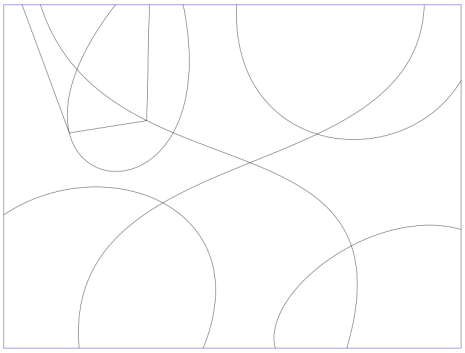
(c) Test 3



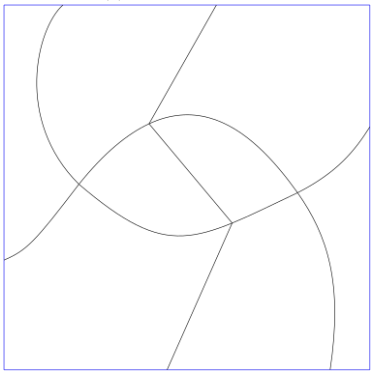
(d) Test 4



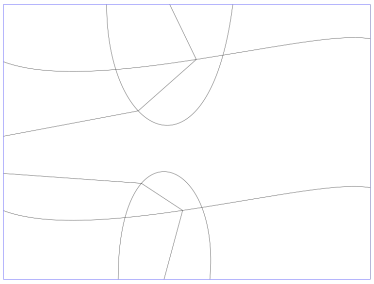
(e) Test 5



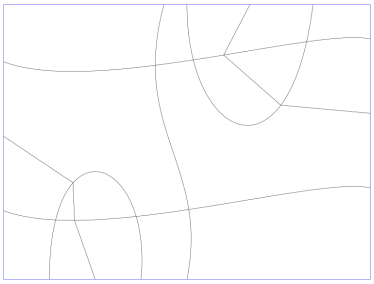
(f) Test 6



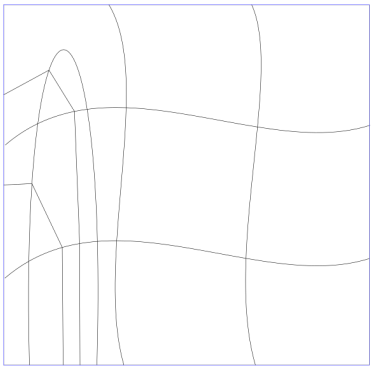
(g) Test 7



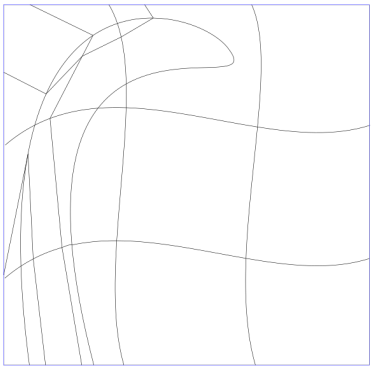
(h) Test 8



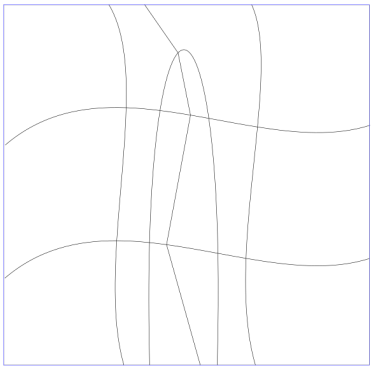
(i) Test 9



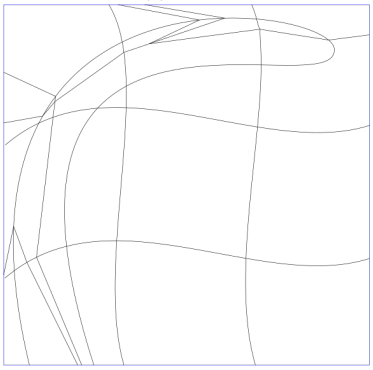
(j) Test 10



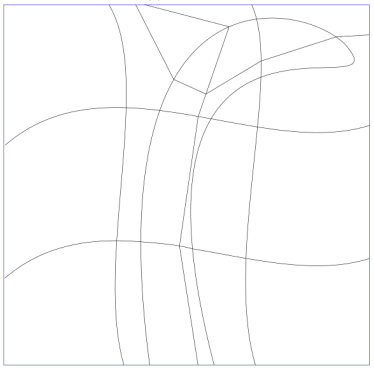
(k) Test 11



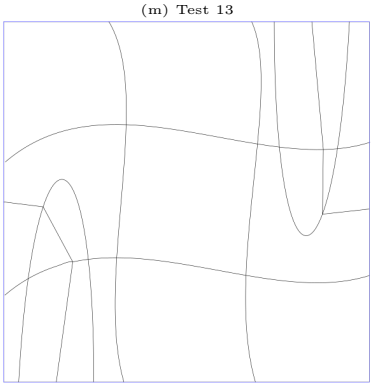
(l) Test 12



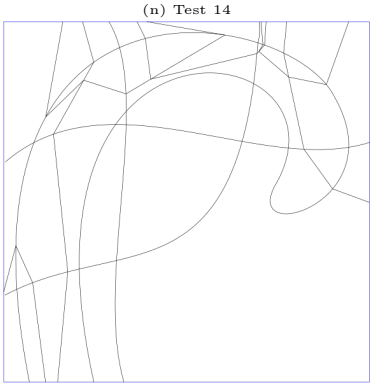
(m) Test 13



(n) Test 14



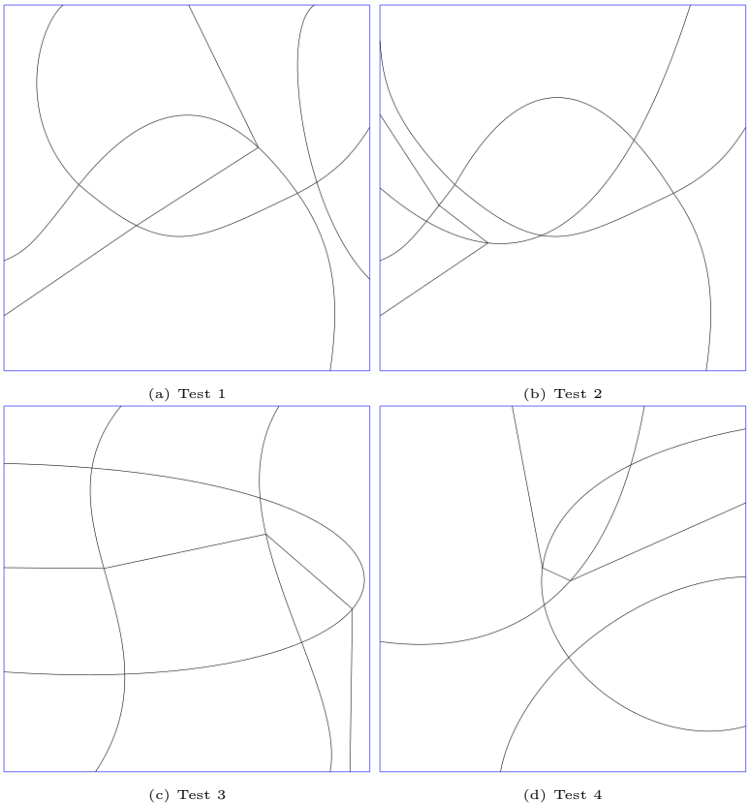
(o) Test 15



(p) Test 16

Figure A.2: Test cases after running the Iterative algorithm

A.3 FPT Results

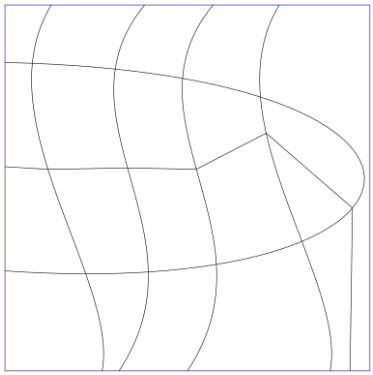


(a) Test 1

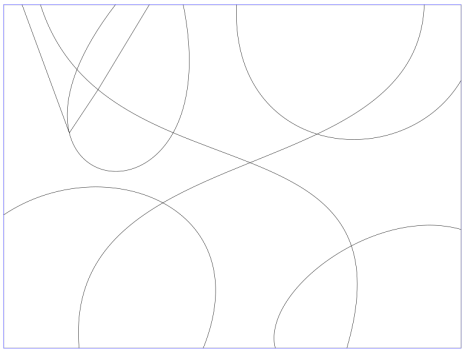
(b) Test 2

(c) Test 3

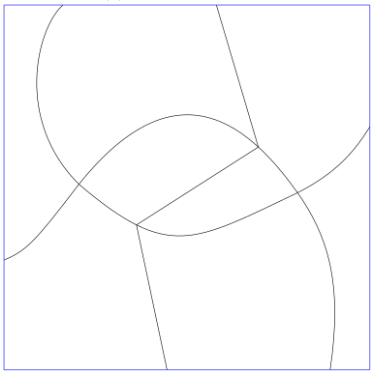
(d) Test 4



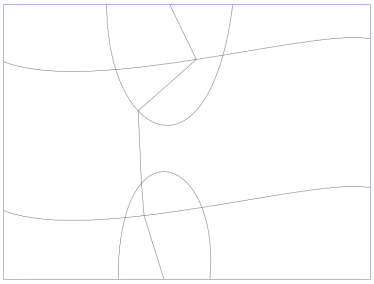
(e) Test 5



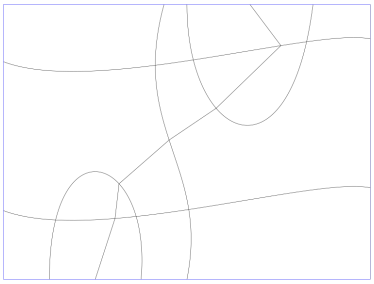
(f) Test 6



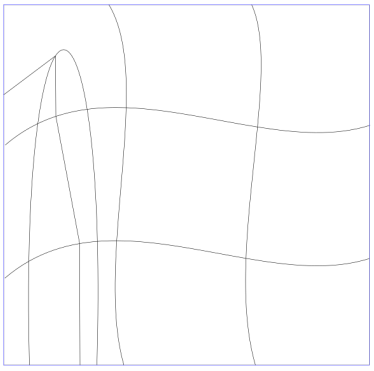
(g) Test 7



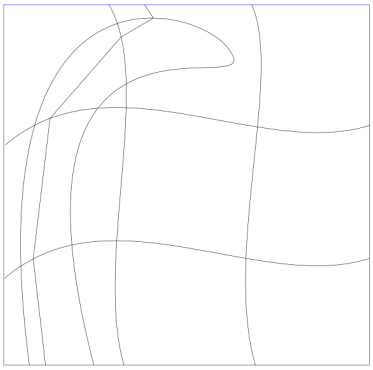
(h) Test 8



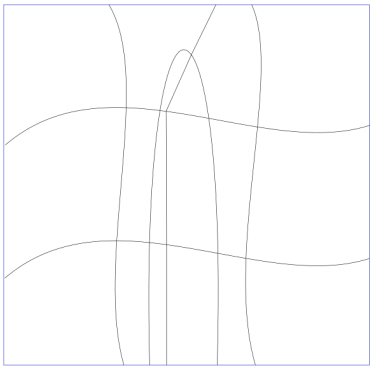
(i) Test 9



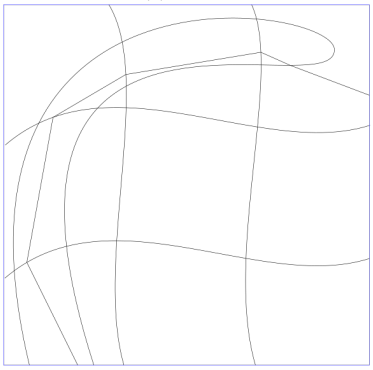
(j) Test 10



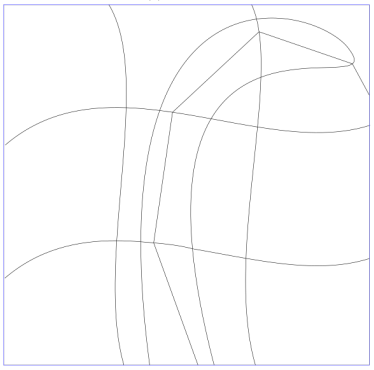
(k) Test 11



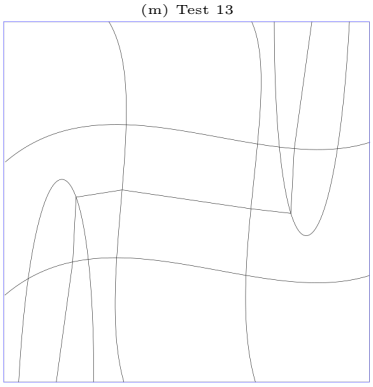
(l) Test 12



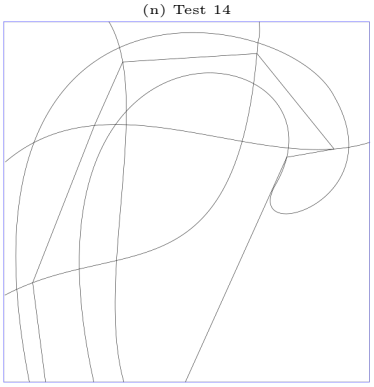
(m) Test 13



(n) Test 14



(o) Test 15



(p) Test 16

Figure A.3: Test cases after running the FPT algorithm

Appendix B

Graphs

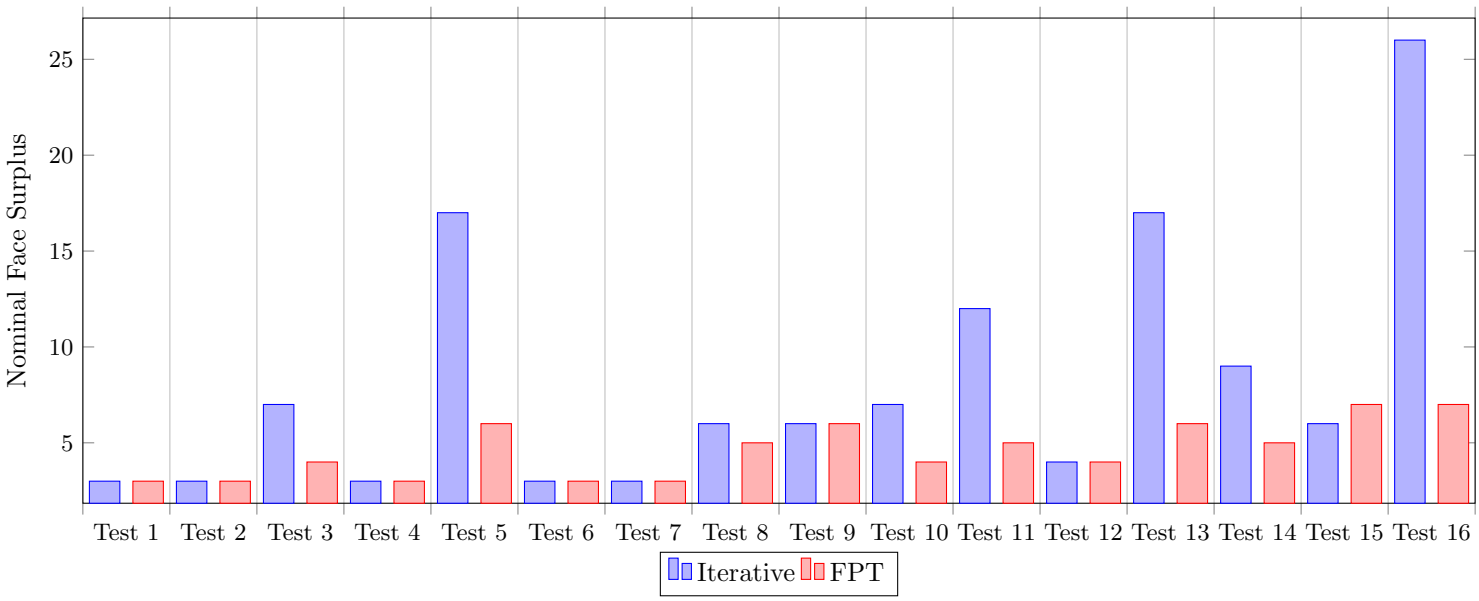


Figure B.1: The nominal face surplus in number of faces for each test case after running both the iterative and FPT algorithm on them.

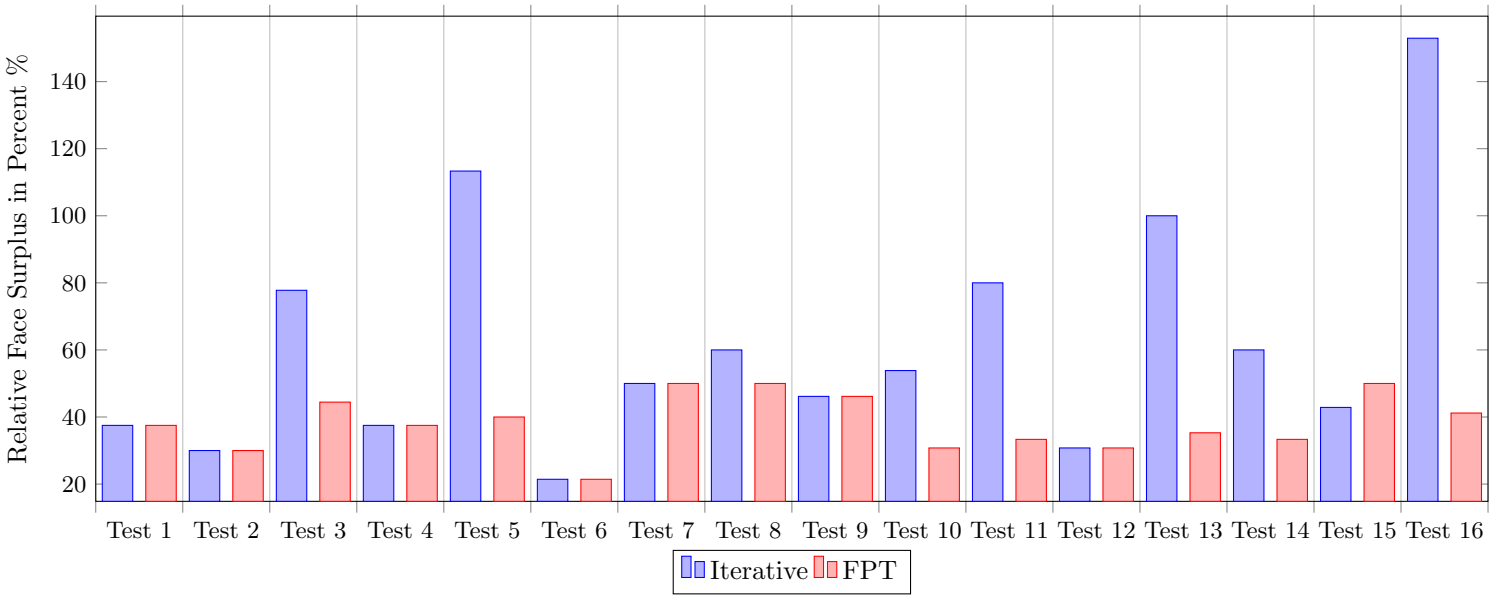


Figure B.2: The relative face surplus in percentage for each test case after running both the iterative and FPT algorithm on them.

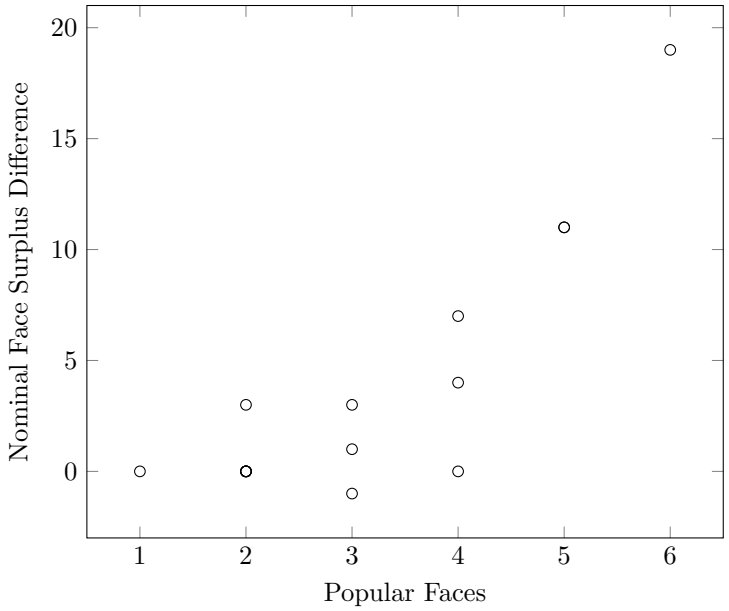


Figure B.3: The number of popular faces in the original curve arrangement vs the difference in nominal face surplus. The difference in face surplus is calculated as the nominal face surplus of the iterative approach minus the nominal face surplus of the FPT approach.

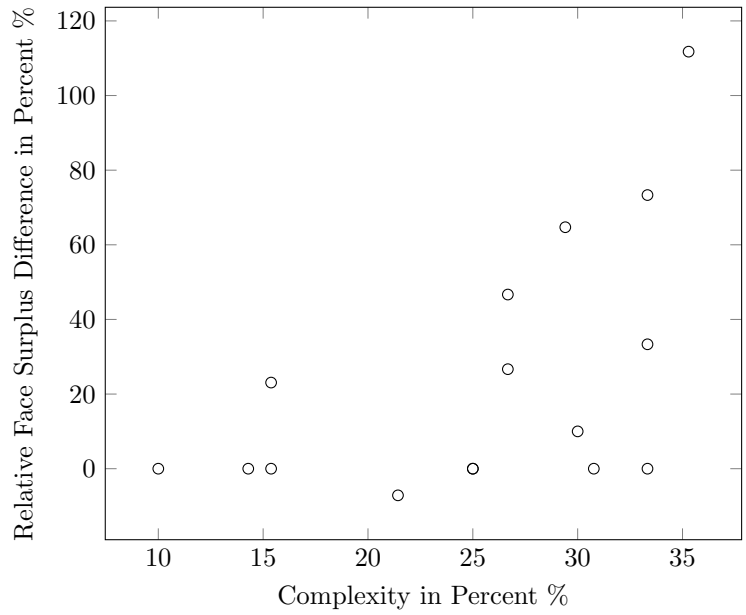


Figure B.4: The complexity of the original curve arrangement vs the difference in relative face surplus. The difference in face surplus is calculated as the relative face surplus of the iterative approach minus the relative face surplus of the FPT approach.

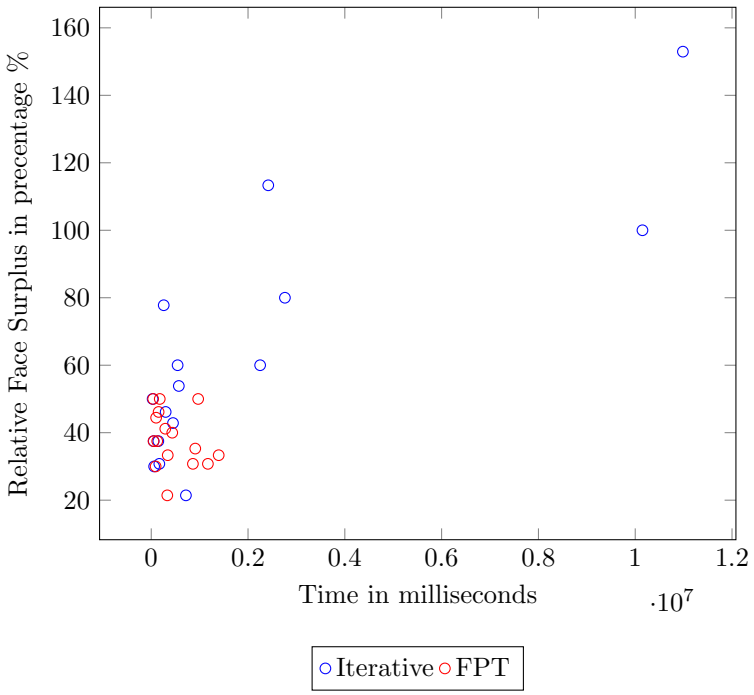


Figure B.5: Running time in milliseconds and relative face surplus in percentage for Iterative and FPT approaches.

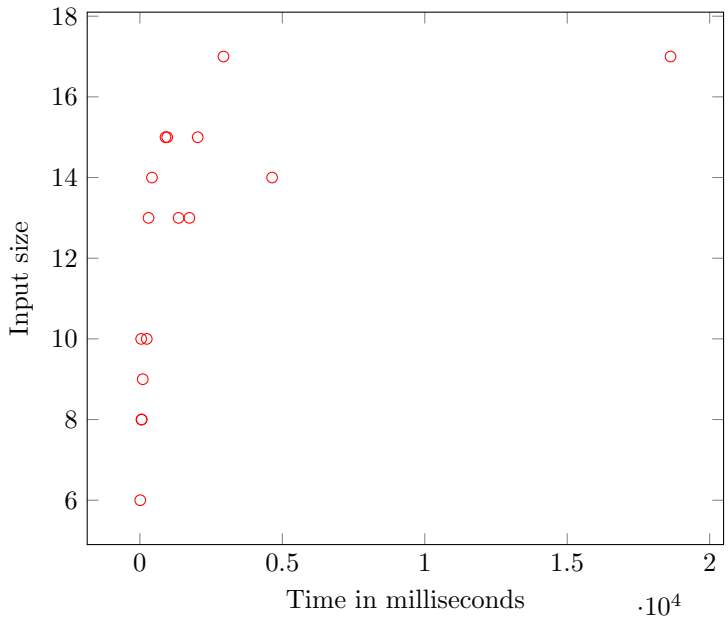


Figure B.6: FPT path time in milliseconds vs input size in total number of faces.

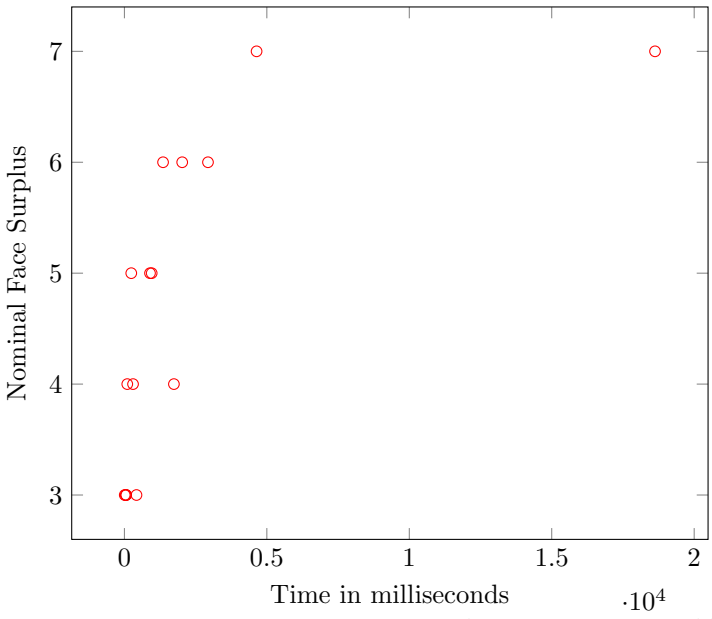


Figure B.7: FPT path time in milliseconds vs nominal face surplus in number of faces.

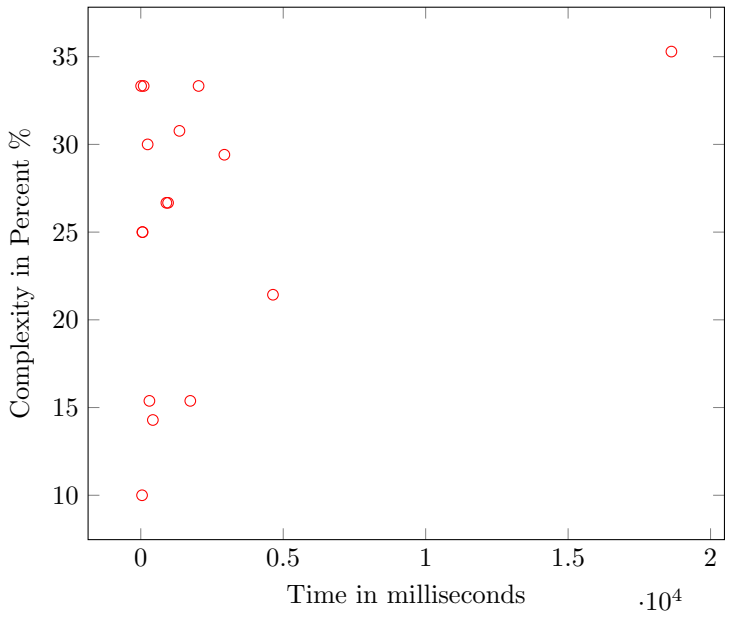


Figure B.8: FPT path time in milliseconds vs complexity of the original curve arrangement in percentage.

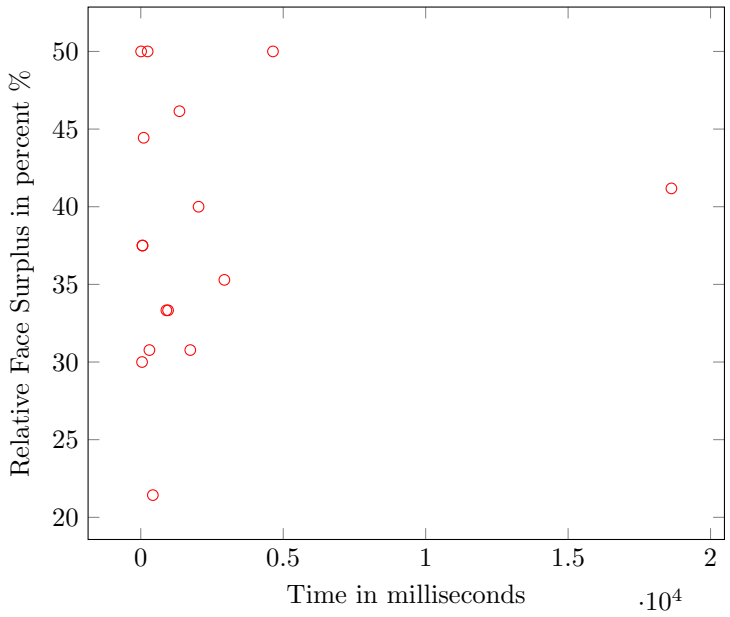


Figure B.9: FPT path time in milliseconds vs relative face surplus in percentage.