Removing popular faces with additional curves

by finding cycles in graphs

vorgelegt von
Daniel Yu

vom Fachbereich Mathematik und Informatik
der Freien Universität Berlin
zur Erlangung des akademischen Grades

Bachelor of Science

Prüfungsausschuss:
:
Betreuer: Prof. Dr Günther Rote
Erstgutachter: Prof. Dr. Günther Rote
Zweitgutachter: Prof. Dr. Wolfgang Mulzer
Tag der Einreichung: 14.03.2025

Berlin 2025

Abstract

In this thesis we study the problem of resolving popular faces on curved nonograms by adding curves to the arrangement. We implement the proposed algorithm from *Nooijer et al.* [1] and test it on automatic generated nonogram puzzles and synthesized graphs. Further we added the possibility of using more than one curve and give some emprical studies on the runtime and error probability.

Eidesstattliche Erklärung

Ich versichere Ihnen hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht. Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keiner anderen Universität als Prüfungsleistung eingereicht.

Daniel Yu, 14. März 2025

Acknowledgements

First and foremost, I would like to thank my supervisor Günther Rote, his research assistant Mahmoud and the research group of theoretical computer science for introducing me to data structures, discrete mathematics and algorithmic techniques. Throughout my entire degree, they provided me with puzzles, sources and wonderful lectures that shaped my academic journey. I am also grateful to my colleagues, especially Arman, Valentin, Frederik, Ilja and Nils, for the conversations and lunches. A special thanks of course goes to my friends, who shaped me beyond university life and with whom I enjoyed competing in games - in particular to my closest friends, Leonard, Ioannis and Yannick. I would also like to thank my entire family, who fed me throughout the thesis. Their care was out of the world. I am very happy for the help and support of my partner Grace. She not only read the thesis and gave me good advices, but also encouraged me when my motivation was lacking and provided me with lots of emotional support the whole time. Thank you very much!

This thesis would not have been possible in this quality without any of the mentioned persons.

Table of Contents

Title Page i

Abstract iii

Eidesstattliche Erklärung v

1 Motivation 1

1.1 Nonograms . . . 1
1.2 Curved nonograms . . . 2
1.3 How to make nonograms easier? . . . 2

2 Graph Problem 5

2.1 Primal Graph . . . 5
2.2 Dual graph . . . 6
2.3 Resolving popular faces via a graph . . . 7
2.4 Graph Problem . . . 8

3 Preprocessing 9

3.1 Extracting data . . . 9
3.2 Detecting Curves . . . 11
3.3 Detecting faces . . . 12
3.4 Computing the constraint sets . . . 13
3.5 Convert the dual to SNESC . . . 14
3.5.1 Construction of the solution curve . . . 14

4 Finding Cycle in the Graph 17

4.1 Introduction to polynomial identity testing (PIT) . . . 17
4.2 Randomized FPT method . . . 17
4.2.1 Decide with DP . . . 18
4.2.1.1 Formulation and ideas . . . 18
4.2.1.2 Recursive formulation . . . 18
4.2.1.3 Runtime and probability of success . . . 19
4.2.1.4 Implementation details . . . 19
4.2.2 Recovering the solution . . . 21
4.2.3 Interesting cases . . . 22
4.3 Exact method . . . 26

5 Experiments and Results **29**
5.1 Experimental Setup 29
5.2 Decision Results 30
5.2.1 Element distribution and error probability 31
5.3 Search Results 33
5.3.1 Overall error Probability 34

6 Conclusion **37**
6.1 Open work 37

Appendix A Nonogram test set **41**

Appendix B ILP Performance **45**

1
Motivation

1.1 Nonograms

Nonograms also known as paint-by-number puzzles are a type of pen-paper puzzle that is played on a grid. It begins with an uncolored grid and descriptions for every row and column. Each description contains numbers of how many consecutive colored squares should lie in order to the row or column. The player can then color faces of the grid and the challenge of the puzzle is to find a coloring of squares, that matches the description. Using the description and logic make the game fun and worth trying. A solution of a nonogram puzzle is given in Figure 1.1

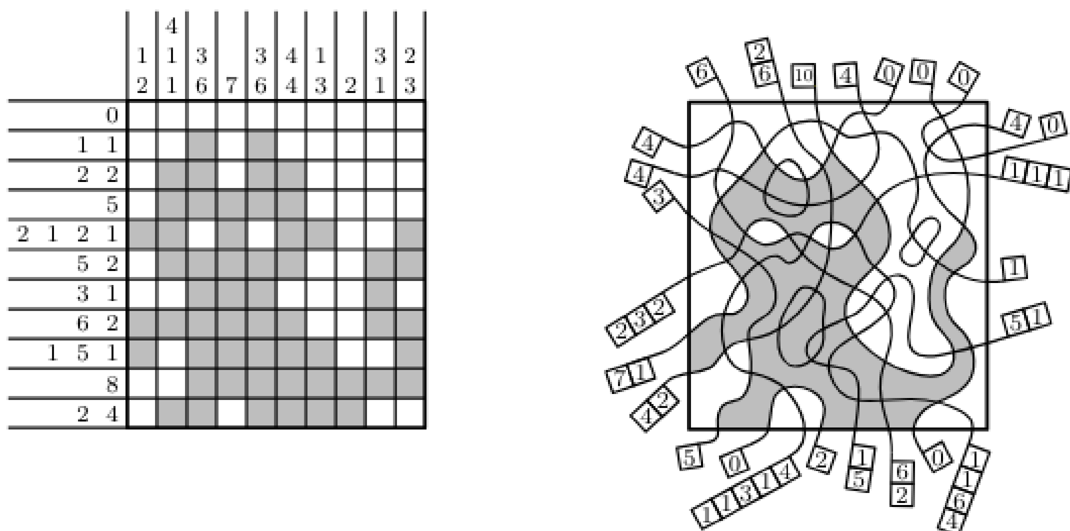

Figure 1.1: An example for a classic and curved nonogram (Image from [1])

Obviously one can verify a solution-coloring very easily by checking whether all descriptions are satisfied, which places nonograms in NP. Ueda and Nagao have shown that solving nonograms or finding another solution given a solution is NP-complete[2].

1.2 Curved nonograms

Curved nonograms, introduced by de Jong [3], are a variation on the classic nonogram. The used image can be described by a set of curves A within a preset bounding frame. The curves and the bounding frame themselves can be subdivided by their intersection points into curve segments. A face is a region in the nonogram that is bounded by a set of curve segments. A face is called popular if it is hit multiple times by a curve.

Curved nonograms offer new challenges to puzzlers, even if they are familiar with the rules. The main difference is that the information about whether one face is colored isn't strictly captured by exactly one column and row description, but it can be referred by many descriptions and each arbitrarily often. This is because the corresponding curve can hit a face multiple times and a face can be hit by arbitrarily many curves. Therefore a player has to carefully trace the sequence of faces of a curve in order to take the new dependencies in the description into account[3]. Furthermore, Van de Kerkhof, de Jong, Parment, Löffler, Vaxman and van Kreveld [4] suggested complexity levels to *curved* nonograms dividing them by how hard it is to understand the rules. They state it would be of interest to generate puzzles of a specific complexity level; currently, they produce it by trial and error.

The three complexity levels are defined as follows:

- *Basic* nonograms are puzzles in which each description corresponds to distinct faces as in the classic nonograms.
- *Advanced* nonograms have popular faces. So the face appears multiple times in some description.
- *Expert* nonograms may have descriptions in which a single face is incident to the same curve on *both* sides. These are exactly the curves that cross themselves [1].

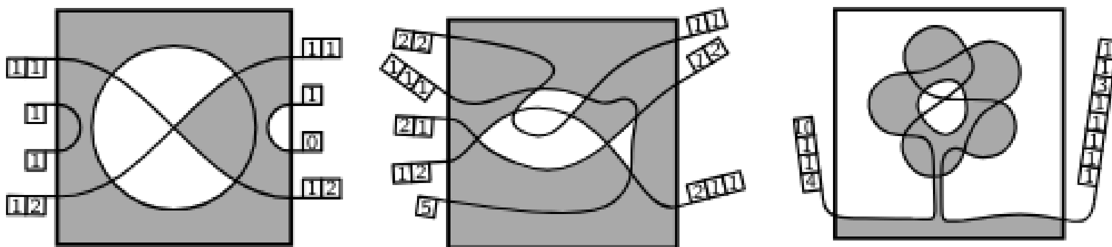

Figure 1.2: An example for the proposed levels (Puzzles from van de Kerkhof et al. [4])

1.3 How to make nonograms easier?

Making nonograms harder is a trivial task, because you can always add a curve that produces popular faces or a curve that intersects itself to get an advanced or expert nonogram. Much more difficult is the task of lowering the complexity of a curve arrangement. A natural approach to attack this problem is to reroute the curves or to add curves to the curve arrangement, to get an easier complexity class. Note also that we can't change the property of self-intersecting

curves by adding new curves, so expert nonograms have to be solved by rerouting curves. De Noojier, Terziadis, Weinberger, Masárová, Mchedlidze, Löffler and Rote [1] showed it is NP-hard to remove all popular faces from a curve arrangement by using one additional curve, but they provided an approach to solve it by reducing the original problem to a graph problem and solving it algorithmically in FPT-time in the number of popular faces. This means it can be computed for some bounded number of popular faces in polynomial time. An example of a solution curve can be found in Figure 1.3. Their work builds our starting point of our implementation and we provide a pipeline, which automatically resolves all popular faces from a given advanced nonogram to a basic nonogram by inserting one curve. Throughout the chapters, we go over the stages of the pipeline and mention key ideas in the implementation [1], which is fully written in python. We also provide variation in the number of curves without changing too much of the approach and are motivated by its failure cases and real error probability.

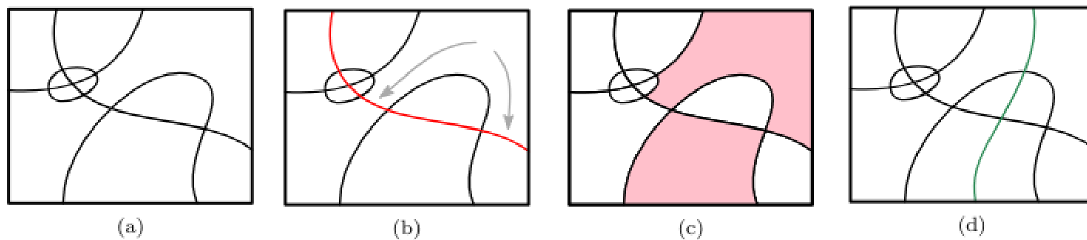

Figure 1.3: (a) curve arrangement in a box. (b) The red curve hits the top right face two times, which makes it popular. (c) All popular faces in the box. (d) All popular faces resolved by the green curve. (Picture taken from [1])

[1]https://git.imp.fu-berlin.de/dy6554fu/easier-nonogram

2

Graph Problem

2.1 Primal Graph

We define the picture of a curved nonogram as a set of curves $\mathcal{A}$ which lie inside a bounding frame. We consider only those arrangements as introduced by [1], where no three curves meet at a point and only finitely many intersections exist.

Definition 1 *A* multigraph *is a graph $G = (V, E)$ with Vas the vertex set and $E \subseteq V \times V$ as the constraint set, which may contain multiple edges (parallel edges) between the same pair of vertices and may include loops. We also often denote an edge between two vertices u, v as uv.*

A plane multigraph *is a graph, that only has a fixed embedding in the plane without edge crossings.*

A plane multigraph is different from a planar multigraph, because it refers to a specific fixed drawing rather than just the property of being drawable without crossings.

Intersection and corner points of the picture can be seen as vertices of V. We can then partition each curve and the bounding curve into curve segments by the vertices. For each curve segment, we define an edge as the tuple of both of its endpoints. This plane multigraph $G = (V, E)$ will be our starting point. We also call this the primal graph.

An embedded graph introduces cyclic orders of edges incident to the same vertex. The set of all these cyclic orders is called a rotation system. Embeddings with the same rotation system are considered to be equivalent, and the corresponding equivalence class of embeddings is called a combinatorial embedding.

The typical data structure to represent an embedding is the doubly connected edge list (DCEL), also known as the half-edge data structure. Each vertex as an object contains its coordinates and also stores a pointer to an arbitrary edge that has the vertex as its origin. Each edge is represented as two halfedges, also called twins, which are typically oriented in opposite directions. Each halfedge is represented as an object and has a pointer to one endpoint, its target vertex, its twin, its right and left neighbor in the embedding from its origin vertex.

Definition 2 (Path, Walk and Cycle) *Let $G = (V, E)$ be a graph, where V is the set of vertices and E is the set of edges. A* walk *in G is a finite sequence of vertices*

$$W = (v_0, v_1, v_2, \dots, v_k)$$

such that for every i, $(v_i, v_{i+1}) \in E$. A path *is a walk in which all vertices are distinct, i.e., $v_i \neq v_j$ for all $i \neq j$. If $v_0 = v_k$, the walk is called a **closed walk**, and if the path satisfies the condition, it is called a **simple cycle**.*

The advantage of the data structure is that it introduces an orientation of each halfedge and we can preserve the orientation while traversing the graph. For a halfedge e for example the next-face-half-edge can be formulated as $next(e) = e.twin.right$ and it preserves the orientation from the original half-edge. In that way a face can be traversed clockwise or counterclockwise, see Figure 2.1.

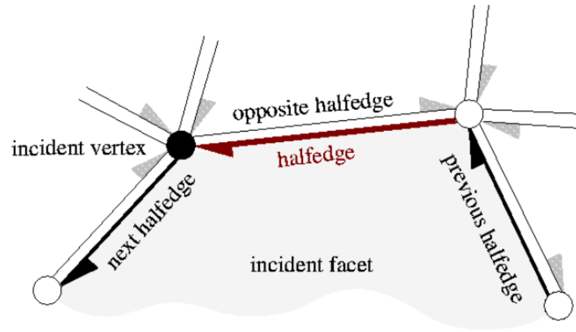

Figure 2.1: Shows the edge relation from a given halfedge e from [5]

With this data structure we can easily identify faces by face walks, iterating through all next-face-edges of a half-edge e until we are at the starting halfedge. The face walks form face sets, which partition the set of half-edges. The face is uniquely determined by the face walk. We will also store for each halfedge the face, which it covers in the face walk.

2.2 Dual graph

Definition 3 *A* dual graph *from a given a planar graph $G = (V, E)$ is the graph $G^d = (V^d, E^d)$, where each face of G corresponds to a vertex in G^d and an edge exists between two vertices in G^d if and only if their corresponding faces are adjacent.*

An example is given in 2.2.

For ease of notation we will denote a vertex in the dual graph as f, because it belongs to a face f in the primal. Note that the dual of the dual graph is the primal graph, so we have a symmetric relation. Further each edge in the primal graph corresponds to an edge in the dual, because two faces are adjacent if and only if there is an edge separating them. So we can easily compute the dual from the primal by identifying the faces for the vertices and for each twin of halfedges we define its dual edge, which connects the two faces that are incident to the twin. The dual graph then can also be a multigraph.

Now the important observation is that every additional curve l in the arrangement $\mathcal{A}$, that visits a sequence of faces $f_1, \dots, f_k$, can be described as a corresponding walk in the dual

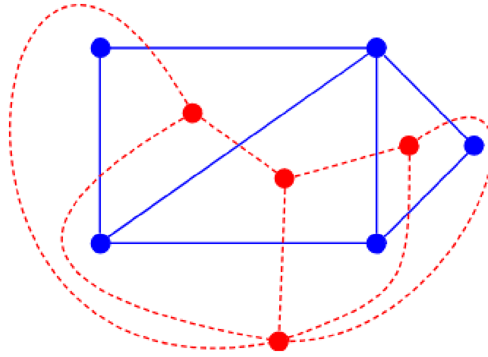

Figure 2.2: The red graph is the dual graph of the blue graph (Image taken from [6])

graph. So the route of the curve can be decomposed as a sequence of edges and we describe the curve as the combinatorial traversal of faces rather than the exact geometric shape of the curve. Note also that we can simulate the entrance and exit of the bounding frame with a corresponding edge to the outer face.

2.3 Resolving popular faces via a graph

Recall that the difference between basic and advanced puzzles lies in the presence of *popular faces* in the arrangement, therefore we want to remove popular faces. We say one curve l resolves or corrects $\mathcal{A}$, if $\mathcal{A} \cup l$ has no popular faces. We call this problem of finding that one curve l as N1R.

Now if a curve enters and exits a face f, we say it visits or crosses f. Note that, if a curve visits a face twice, it automatically produces a popular face, so every face can be only visited and cut into two faces once. This also implies that we can only resolve popular faces with exactly two incident popular curve segments. Of course, there also exist popular faces with only two incident popular curve segments, which are not resolvable by one curve. If we want to do better, we need more curves, but we stay in the scenario of adding just one curve. Also the curve l must not cross an intersection point, if we want to preserve a simple arrangement $\mathcal{A} \cup l$.

With the rules in mind we want to focus on resolving popular faces locally with the dual graph. Let f_p be a popular face and let c, c' be two curve segments contained in the facewalk of f_p and belonging to the same curve a. The additional solution-curve l has to cut the face f_p into two new faces, such that c and c' are separated. Each local cut decision can be represented as a pair of edges between two different adjacent face neighbors n_1, n_2 of f_p. Because we always have to traverse f_p in the middle, we can get rid of f_p and instead add a shortcut edge (n_1, n_2) as our solution edge, i.e. Figure 2.3. In general if we have more than one pair of popular edges, we have to cut all popular edge pairs into two different face walks. Therefore a solution edge can be recognized by the property, that both new face walks have to hit exactly one popular curve segment for each popular curve from f_p. This idea will later be used to gather all solution edges from a popular face.

There are some side effects we have to take care of; keep in mind that with the shortcut solution edges, we do not want to allow traversing popular faces, because we do it implicitly in the solution edge. Eliminating vertices has to be done carefully, because we could eliminate

vertices from a solution edge, i.e. a solution edge (f_1, f_2) of f_p where f_1 is also popular. Only in this case of adjacent popular faces, we have to introduce bridge vertices between the faces, which represent the possible traversal of both faces.

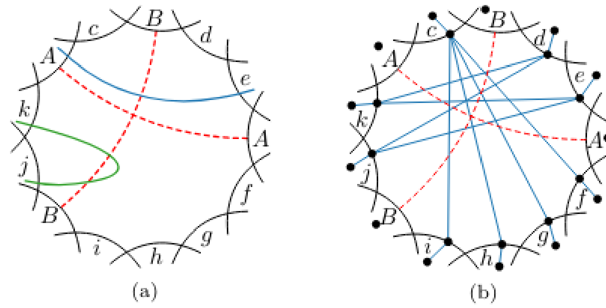

Figure 2.3: (a) shows a face, bounded by its curve segments. Curve segments from the same curve are connected with a red curve. (b) shows all solution curves to resolve the popular face. Each left and right path of a solution edge contain exactly one B and A. The cut can be also seen as cut of edges between the popular curve segments (Image taken from [1])

With all that in mind, a proper solution-curve for a curve arrangement A is a walk in the dual, that uses for every popular face one of its solution edges. This walk also has to visit the outer face once, so we can route the curve in and out of the bounding frame.

2.4 Graph Problem

Consider a map with location points and routes between them. At each route, one can go over many attractions like a supermarket, a house, or a school. The problem is to find the shortest tour of visiting points, such that some given attraction is visited exactly once.

Behind this riddle lies a clean combinatorial problem:

Problem 1 (SNESC) *For a given graph and a number $G = (V, E)$, a number l and constraint sets $\{S_1, \ldots, S_k\}$ with $S_i \subseteq E$. Does there exist a cycle of length l that contains exactly one edge from each constraint set S_i?*

So we can represent the attraction tour problem as a SNESC-instance, but more importantly we can also represent our N1R-instance as a SNESC-instance by using a modified dual graph and the set of solution edges for each popular face as a constraint set.

The modifications include the vertex elimination of all popular faces and their edges, the inclusion of the bridge vertices between two adjacent popular faces, and the addition of the solution edges, where popular faces are replaced by their bridge vertices. To enforce the traversal of the outer face, we can simulate the outer face as a popular face and include all of its transitive edges in a new constraint set S_{k+1}.

A resolving curve has to traverse from face to face and the possible ways to resolve popular edges are captured by the constraint sets S_i. So if we find a simple cycle without repeated vertices using each edge in the constraint set once, it corresponds to a curve that resolves each face once and uses the outerface. The solution is even a one-to-one correspondence apart from the geometric shape of the curve. We will discuss the part of solving and retrieving the solution from the SNESC-problem in chapter 4.

3 Preprocessing

3.1 Extracting data

The images from the automatic nonogram generator from [7] generate ipe-files. Ipe itself is a drawing editor and stores its files in an XML-based file format, ending with .ipe. The curves in one drawing are all encoded in a metatag called <path>. They use control points and character curve flags, which determine uniquely the shape of the curve. For the nonogram we only need two types of curves:

1. character l for line, which takes two controlpoints - its endpoints
2. character c for cubic Bezier curve, which takes four controlpoints. It is also a parameterized curve, more details can be found from van de Kerkhof's master thesis [7].

Matplotlib supports drawing both curves, so one could replicate the image by drawing them with an external drawer.

We are only interested in the provided snapped files, in which the curves are already cut into curve segments. Cutting curves into curve segments is out of the scope of this thesis.

The first step we do before extracting the curves is to normalize the input files in src/preprocess::normalize_file_path. We want to ensure that all meta tags in an ipe-file are on separate lines, improving consistency and fulfilling the assumptions of the scanner. In

```
1 <?xml version="1.0">
2 <metadata>
3 </metadata>
4 <page>
5     <point></point>
6 </page>
7
```

(a)

```
1 <path metadata>
2   100 100 m
3   200 200 l
4 </path>
5
```

(b)

```
1 <path metadata>
2   100 100 m
3   150 100
4   150 150
5   200 200 c
6 </path>
7
```

(c)

Figure 3.1: (a) ipe-file structure. (b) line tag. (c) cubic curve tag.

the current implementation the normalizer is always adding a newline character in front and behind an important tag, even if they are unnecessary. A cleaner normalizer is still in work.

Then our program calls our scanner function src/preprocess::ipe_and_extract_primal_graph, which iterates through each line and scans the type and the control points from the curve. The scanner assumes that our ipe file is well-structured as in the Figure 3.1. Each relevant path tag with its control points is stored in a curve object. To ensure correctness, we can draw all the curve objects with matplotlib and validate whether the drawing is the original image.

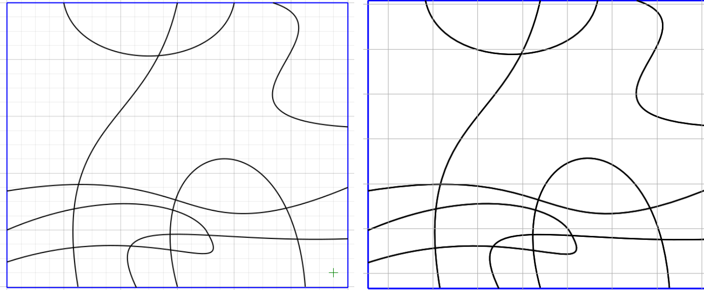

Figure 3.2: Left is the image parsed with ipe, right is the own created image with matplotlib

At the same time, when running into a curve, we dynamically grow our primal graph by adding two half edges and sorting them correctly in the linked list of their endpoints and adding vertices when new endpoints are seen. Correct means that for every vertex its linked list in the DCEL has to match the order of the unique plane graph we want to get. We achieve this by storing for each edge the clockwise degree between the $(1,0)^\top$ vector and the tangent of the curve at the vertex coordinates and sorting the half edges in each linked list accordingly. We assume that no two curves on a vertex have the same degree; if so, we raise an exception. Because we are working with Bezier curves and lines, the tangent vector $\vec{s}$ to the curve at a point (t_1, t_2) is $(t_1 - c_1, t_2 - c_2)^T$, where c_1, c_2 is the next control point. We can then directly compute the degree by using the smaller degree between the two vectors and some case distinction. The smaller degree can be computed by:

$$deg(\mathbf{a}, \mathbf{b}) = \cos^{-1}\left(\frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}\right)$$

For the case distinction, one should have the following image in front.

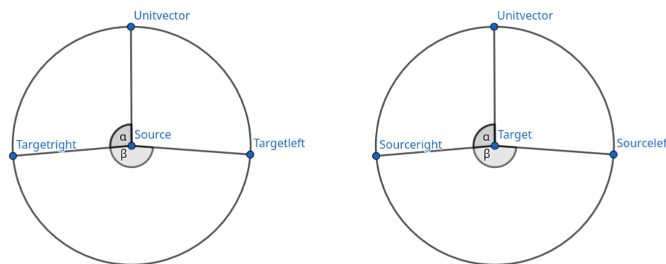

Figure 3.3: On the left picture the centerpoint source represents the start point of a curve. Outgoing targets represents the next possible controlpoints. The Unitvector is fixed. The right picture is very similar, but here the centerpoint represents the endpoint of the curve. α is by the formula given above

To compute the degree of the start point to the next control point, the relative position from target to start is crucial, whether we have to replace α by $\beta = 360 - \alpha$, i.e. 3.3. The case from a previous control point to the endpoint is very similar, but you have to keep in mind that you need the degree of the target point. So the computation of the degree is constant and because we only consider bounded many intersections on a vertex by a constant. So we have bounded many half edges in a linked list, which makes the sorting by insertion constant. The normalizer and scanner work like an automaton, and because the operations in one state are always constant, they need linear time in the file length to create the primal graph.

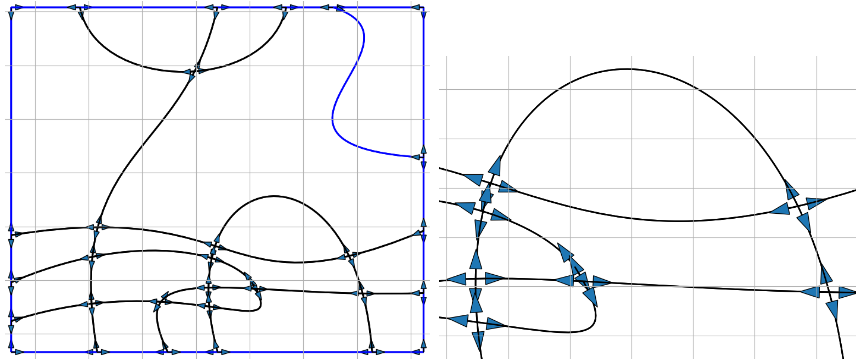

Figure 3.4: Left image shows curve segments with the internal stored outdegree in the vertex. Right image is a zoomed version of the image

3.2 Detecting Curves

To detect popular edges of popular faces, we have to compute which edges belong to the same curve. Our first step to find an entire curve globally over the image is to first process locally connecting parts on a vertex. We introduce a new pointer for every halfedge e, which points to the opposite curve edge from the same origin vertex if it exists. Assuming that two curve segments c_1, c_2 from the same curve split at a point p. Their ingoing degree d_1, d_2 at p is about on opposite sides, meaning $d_1 + 180 \mod 360 \approx d_2$ and the two terms do not deviate more than 10 degrees. We also have to take into account that near zero the difference can deviate about 360 degrees. By this property we can assign each halfedge from a given origin vertex a best suitable halfedge in the same neighborhood. In practice we search for a suitable curve neighbor once for every halfedge when adding the halfedge to the primal graph. If there exists a local curve pair, one edge has to be inserted later and sets both curve pointers.

With the local relation given, we can easily traverse each curve from an arbitrary point of it. We know that each edge has to be assigned to one curve, which means we can start with an arbitrary edge. We call the subroutine src/graph.find_and_set_all_joint_curves, which essentially assigns each edge exactly one curve id. Edges with the same curve id belong to the same curve. Because we visit each edge only once and the pointer operations are constant, this procedure needs linear time in the number of halfedges.

Algorithm 1 Set Curve id in one direction

Require: A stack with a subset of all half edges E, startedge e, id id

Ensure: Every edge on the same curve direction as e has the same id

1: cur_e $\leftarrow e$
2: cur_e.curve_id $\leftarrow$ id
3: E.remove(cur_e)
4: **while** cur_e.twin.lneighbor $\neq$ None **do**
5: cur_e $\leftarrow$ cur_e.twin.lneighbor
6: cur_e.curve_id $\leftarrow$ id
7: cur_e.twin.curve_id $\leftarrow$ id
8: E.remove(cur_e)
9: E.remove(cur_e.twin)
10: **end while**

Algorithm 2 Set all Cruve ids

Require: A stack of all half edges E

Ensure: Every edge has a curve id

1: curve_id $\leftarrow$ 0
2: **while** $E \neq \emptyset$ **do**
3: cur_e $\leftarrow E.pop()$
4: SetCurveId(cur_e, curve_id, E)
5: SetCurveId(cur_e.twin, curve_id, E)
6: Curve_id $\leftarrow$ curve_id + 1
7: **end while**

For Validation, we also support drawing and labeling edges by their curve IDs.

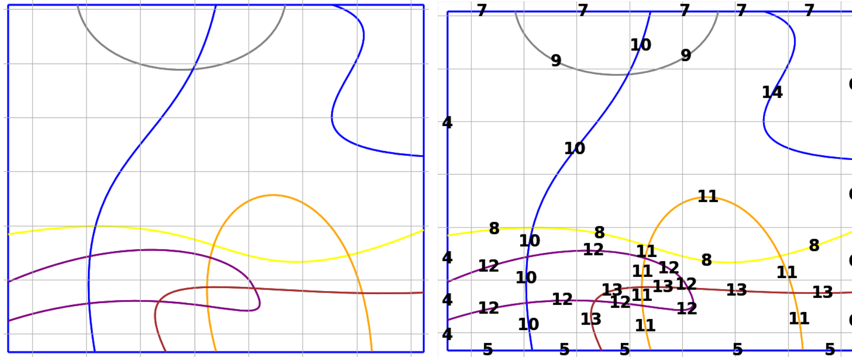

Figure 3.5: Left image shows curve segment colored by their id. Right image shows the curve segments labeled with their id in a zoomed version

3.3 Detecting faces

The standard algorithm to compute the dual graph uses the idea that every halfedge is used exactly once in a face walk, and each face is characterized by its halfedges. It is essentially very similar to our curve traversal, but we do face walks.

This is essentially implemented in src/graph.compute_dual_graph_and_find_popular_faces

Simultaneously to the face walk, we keep track of the order of seen curve IDs and store the beginning halfedge of the curve. If we finish with a face walk, we can then easily compute whether we have popular edges and store pairs of edges in a pointer. If we have more than two popular edges from the same curve on a face, we cannot solve the instance directly with just one additional curve and mark it as so.

Algorithm 3 compute all faces of a graph

Require: A stack of all half edges E
Ensure: Create set of faces F

1: $edge_{start} \leftarrow E.pop()$
2: $edge_{cur} \leftarrow edge_{start}$
3: **while** $E \neq \emptyset$ **do**
4: $\quad edge_{cur} \leftarrow edge_{cur}.next_face_edge()$
5: $\quad E.remove(edge_{cur})$
6: $\quad$ **if** $edge_{cur} = edge_{start}$ **then:**
7: $\qquad$ Create new Face f, store facewalk in f
8: $\qquad edge_{start} \leftarrow E.pop()$ $\quad \triangleright$ We can safely do this, because E is not empty
9: $\qquad edge_{cur} \leftarrow edge_{start}$
10: $\quad$ **end if**
11: **end while**

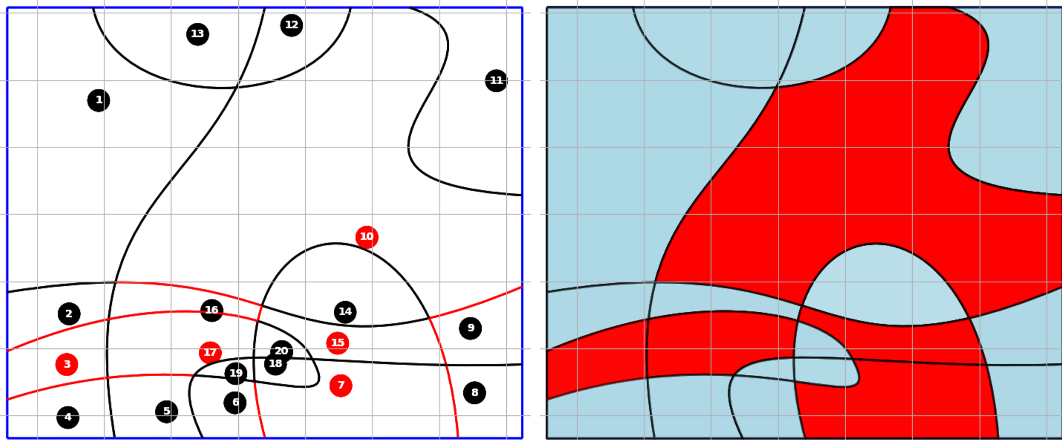

Figure 3.6: Left image shows the computed faces with popular faces. Right image is a colored version of the faces

3.4 Computing the constraint sets

Assume that a given popular face f_p only has popular edges, which come in pairs from the same curve. If not, it should be recognized by the dual procedure before. We use a sliding window, memorizing the front edge of the window, the end edge of the window and the seen popular curve segments in between of the window. Recall the characterization of solution edges is that one popular curve segment from a popular curve of the popular face has to be seen exactly once. If the condition of a solution edge is not met for the window, meaning we didn't hit every popular curve from the popular face once, we have to grow the window by incrementing the front edge. If we have a pair of popular curve segments in between, we have to grow the end edge until we only have one of the two popular curve segments. Now to keep the seen popular curve segments in between correct, we just have to update them constantly when changing the front and end edge. Now if the condition for the solution edge is met, we can take the set of edges E_1, which are all edges from the front edge to the next popular curve segments, and the set of edges E_2, which are all edges from the end edge to the next popular curve segments, and $E_1 \times E_2$ form all solution edges of the popular face, because any endpoint from E_1 with an endpoint from E_2 fulfills the solution condition. We do the

sliding window until the front edge visits the starting point of the end edge, because every possible solution edge with the starting point of the end edge should be already covered by the end edge. Because the sliding window depends on the growing of the front edge and end edge and their change can be bounded by the number of halfedges in the facewalk, we only do a linear number of steps. The bottleneck of the function is the construction of the solution edges, as their number can grow quadratically with respect to the half-edges in the face walk. The implementation can be found under src/graph.set_solution_edges and does it for all popular faces.

3.5 Convert the dual to SNESC

Our construction of the SNESC-graph is split into diverse functions. First of all, we have to construct the dual graph from the edges and face of the primal. We use the face id for the For the vertex elimination and integration of solution edges from the popular faces, we call preprocess.integrate_popular_info_to_dual_graph. It also integrates bridge vertices, if we have adjacent popular faces; see 2.3, and provides a backsubstitution table for bridge vertices to reconstruct the dual path from a SNESC-solution. We also provide in separate functions the cloning of vertices, connecting edges, and adding edges to edge constraint. With these features, we can clone the outer face from the dual, such that a dual face can be traversed multiple times. This allows us to represent multiple curves in the curve arrangement.

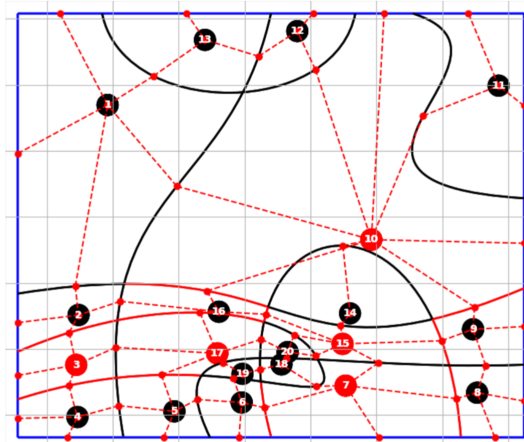

Figure 3.7: Dual graph in the primal, red vertices are popular faces

We also have to mention that in the implementation we modeled the dual graph as a simple graph rather than a multigraph and defined the edge identity as the pair of its endpoints rather than giving it its own identity. An extension to multigraphs could be interesting, because if we want to apply data reduction rules, multi-edges can be constructed with different characters due to the constraint sets.

3.5.1 Construction of the solution curve

Given an optimal solution $C = f_0 \dots f_l f_0$ for the SNESC-instance. For each edge $f_i f_{i+1}$ in the cycle, there exists a set of dual edges $S_i = \{e_1, \dots e_w\}$ to $f_i f_{i+1}$, that are all possible to intersect. Each dual edge e_1 stores its corresponding curve segment c_1. To construct the

solution drawing, we have to choose exactly one dual edge e_j from $S_i \quad \forall i \in [l]$. Choosing one dual edge e_j for the solution curve means intersecting its curve segment c_j. Then we can draw the solution curve local on each face $f_i \in C$, by connecting the midpoints of c_p, c_q from $e_p \in S_{i-1}, e_q \in S_i$. The midpoint of a straight-line can be computed by the arithmetic mean of its endpoints. If the curve c is a Bézier curve, all points can be computed by the formula

$$x = \sum_{i=0}^{n} \binom{n}{i} t^i (1-t)^{n-i} b_{ix}$$

where $b_i x$ is the x-coordinate of the i-control point and $t \in [0, 1]$. Setting $t = 0.5$ yields the midpoint m_c. With the coordinates of the midpoints we can insert the solution with new path tags to the starting ipe-file.

Currently we select arbitrarily dual edge without any aesthetic decision making. Additionally we only support drawing straight-lines between midpoints (m_c, m'_c). See Appendix A A for examples.

When there exist an aesthetic measurement, we can formulate our problem as finding a combination of dual edges such that the aesthetic measurement is maximized or minimized.

To improve the visual quality, we propose some ideas: when drawing the solution curve l into the face f_i. Let w.l.o.g. be e the dual edge from S_{i-1} from $f_{i-1}f_i$. To lift l up from c at the intersection point of l and curve segment c the solution curve should be perpendicular to the slope-vector of the curve segment. We call the perpendicular slope-vector $\vec{p}$, which is oriented into the face f_1.

A scaled vector of $\vec{p}$ can be used to compute the middle control point for a Bézier curve, which gives us a smoother transition between curve segments.

$\vec{p}$ can also be used to measure how suitable the next border edge $e' \in S_i$ is, i.e. measure the similarity between the slope vector and the vector between midpoints $\vec{m_c} - \vec{m'_c}$ by the dot product.

4

Finding Cycle in the Graph

4.1 Introduction to polynomial identity testing (PIT)

Problem 2 PIT *Given a polynomial* $p(x_1, \dots, x_n)$ *in some finite field* $\mathbb{F}[x_1, \dots, x_n]$*. Is* p *not the zero polynomial, which always returns zero.*

It is an open question whether PIT lies in P, but there exist practical algorithms by just choosing and evaluating random points. This idea works mainly because a non-zero polynomial cannot have "too many" roots over a field. This algorithm, due independently to Schwartz [8], Zippel [9] and DeMillo and Lipton [10], is a Monte Carlo algorithm with bounded probability of false positives.

The following lemma guarantees us that we can bound the error probability of false positives:

Lemma 1 (Schwartz-Zippel-DeMillo-Lipton Lemma) *Let* $P \in F[x_1, x_2, \dots, x_n]$ *be a non-zero polynomial of degree* $d \geq 0$ *over a field* F*. Let* S *be a finite subset of* F*. Then,*

$$\Pr_{r_1, \dots, r_n \in S}[P(r_1, \dots, r_n) = 0] \leq \frac{d}{|S|}.$$

It is a well-known result and there is some work into derandomization. In fact, there are implications that it is very much unlikely that PIT $\in$ P [11].

4.2 Randomized FPT method

In this method we present the dynamic programming (DP) approach using finite fields, as described by de Noojier, Terziadis, Weinberger, Masárov, Löffler and Rote [1], extending an algorithm of Björklund, Husfeld and Taslaman [12].

4.2.1 Decide with DP

4.2.1.1 Formulation and ideas

We aim to determine whether a feasible cycle of a given length exists with the Lemma 4.1. Let the assignment of a finite element to an edge $uv \in E$ be $\phi(uv)$. Furthermore, we represent any walk W as a monomial, defined as the product of the finite field values of its edges. This results to

$$\phi(W) = \prod_{uv \in W} \phi(uv)$$

If a monomial exists, the lemma guarantees a positive answer with high probability. However, for the original problem, there can exist exponentially many walks and so exponentially many monomials we may have to check. The crucial result is that we can compute in FPT-time the sum of monomials of some restricted walks that satisfy the edge constraints and decide for all polynomials altogether. The power lies behind the encoding of walks as monomials and the allowed operations over the polynomial ring. We can easily talk about aggregated walks by addition of monomials and adding the usage of an edge on multiple such walks by the multiplication with an aggregated sum.

To efficiently compute the sum of monomials for walks satisfying edge constraints, we use a DP formulation. First we focus on only walks root with an ending point f. Because we know a valid cycle has to pass through one of the edges of a constraint set S_i, we branch for the ending point into every edge of the smallest of the constraint sets. Let without loss of generality S_1 be the smallest constraint set among all $S_i \subseteq E$. We will use one vertex s from the edge $sf \in S_1$ as the start point and the other as the end point f, to detect a closed walk. Note, that by this method we indirectly orient the walk of the solution cycle uniquely. Let also k be the number of constraint sets and $R \subseteq [k]$ be a subset of index set of constraints sets.

4.2.1.2 Recursive formulation

Let $T_f(l, R, v)$ be the polynomial of all walks W, that

- ends in f and starts in v
- have their last edge in S_1
- consists of l edges
- use exactly one edge from each set S_i with $i \in R$
- contains no edge from the set S_i with $i \notin R$

with $T_f(l, S, v) = \sum_{w \in W_{l,S,v}} \phi(w)$. For the optimal solution, we search for the $T_f(l, [k], f)$ with the smallest l, that is non-zero for any $sf \in S_1$. With the tables, we can keep track of the existence of walks with specified vertex v and the set R, that are already visited.

Our base case of the program will be $T_f(1, I(sf), s) = \phi(sf)$.

The idea of the recurrence is that we can reuse already computed walks of length $l-1$ for the walks of length l. By this we get the following recurrence

$$T_f(l, R, v) = \sum_{\substack{uv \in E \\ I(uv) \subseteq R}} \phi(uv) T_f(l-1, R \setminus I(uv), u)$$

where $I(uv) := \{i \mid uv \in S_i\}$ of sets S_i denotes the index set of constraint sets containing uv.

4.2.1.3 Runtime and probability of success

The runtime depends on the number of entries from the DP table we compute and the cost per entry. In the worst case, if no solution exists, we have to compute all T_f values for every $l \leq n$, every start point $v \in V$, and every subset $R \in [k]$ on every branch $sf \in S_1$. Each table needs $O(m)$ lookups and constant operation for each edge. This results in a runtime complexity of $O(2^k mn^2 |S_1|)$.

The probability of successfully detecting the optimal solution we can bound by the lemma 4.1 and the fact that we evaluate a polynomial of degree $l \leq n$ with l as the solution length. This implies a probability of at least $1 - \frac{n}{q}$. Here q is the size of the finite field. By choosing q to be very large, around $2^{64} \approx 1.8 \cdot 10^{19}$, it is sufficient for most practical applications. We also have to restrict the field to powers of two for correctness. If no solution exists, the algorithm always returns False.

The proof of correctness can be found in [1], which explains how all valid non-simple cycles cancel out in the polynomial.

4.2.1.4 Implementation details

Each vertex is encoded as an integer, and an edge as a frozenset of vertices. Every subset R will be represented as a bit vector, meaning each index corresponds to one bit of the vector. The entire SNESC—instance will be then encoded as an object.

Static information, such as the function I as a map, the edge list E and the encoding of the power set of $[k]$ as bit vectors is precomputed once and stored in the object. Set operations are simulated by bitwise operations in constant time for $k \leq 64$, which is sufficient for all practical use cases. For finite field calculation there are a variety of powerful libraries; we will use Hostetter's Galois library, which efficiently integrates NumPy arrays [13]. This allows us to store and use a single 3-dimensional NumPy array for the DP tables.

The dynamic program is implemented under src/snesc::has_simple_cycle. The assignment of weights is built in a separate function in src/snesc::set_random_weights and before the DP. We also want to mention that we integrated the feature to search for a path with endpoints a, b, that fulfills the constraints set. The only adaptation we made is that we have to branch on each incident edge from a start point b and have to check for $T_a(l, [k], b)$ for every l.

Algorithm 4 Dynamic Programming for SNESC

```
1: Input: S_1, I, E, bitmap_P([k]), φ
2: for each v_end u ∈ S_1 do
3:     memo ← []
4:     memo[1, I(v_end u)], u] ← φ(v_end u)
5:     for l from 2 to n do
6:         for R ∈ bitmap_P([k]) do
7:             for uv ∈ E do
8:                 if I(uv) ⊆ R then
9:                     memo[l, R, v] += φ(uv) · memo[l − 1, R \ I(uv), u]
10:                    memo[l, R, u] += φ(uv) · memo[l − 1, R \ I(uv), v]
11:                end if
12:            end for
13:        end for
14:        if memo[l][{1, ..., k}][v_end] ≠ 0 then
15:            return True
16:        end if
17:    end for
18:    return False
19: end for
```

4.2.2 Recovering the solution

To recover the solution, we can deduce solution edges by systematically eliminating edges and observing the change of the output of the modified graph. We use the method above as an oracle. Start with one vertex from the first edge that we used to orient the cycle. The other vertex acts as the endpoint, ensuring the termination of recovery and guaranteeing the formation of a cycle. So if we have not reached the endpoint, to get the next vertex from the current vertex, we do a binary search on the unused incident edges of the current vertex and by partitioning the incident edges into two subsets. If the output returns zero after removing the subset of edges and the other subset returns a non-zero value, we can safely remove the subset that didn't influence the returned value. If both return a value, we have to take the subset yielding the shortest solution length. At last, if both removing parts return zero, we have to do the oracle again with completely new weights, because we got no information on which subset is essential for the optimal solution.

Algorithm 5 Binary Search on Incident Edges for SNESC

1: **Input:** $\mathrm{v}_{cur}, \mathrm{v}_{end}, \mathrm{e}_{first}$
2: $e_{cur} \leftarrow e_{first}$
3: $\mathrm{snesc}_{sol} \leftarrow []$
4: **while** $\mathrm{v}_{cur} \neq \mathrm{v}_{end}$ **do**
5:     incident_edges $\leftarrow \{(u, \mathrm{v}_{cur}) \mid u \in N(\mathrm{v}_{cur}) \setminus e_{cur}\}$
6:     num_edges $\leftarrow$ |incident_edges|
7:     **while** num_edges > 1 **do**
8:         left_half $\leftarrow$ incident_edges[: num_edges / 2]
9:         right_half $\leftarrow$ incident_edges[num_edges / 2 :]
10:         answer_{right}, $\mathrm{sol_len}_r \leftarrow$ decide(left_half, edge_{first})
11:         answer_{left}, $\mathrm{sol_len}_l \leftarrow$ decide(right_half, edge_{first})
12:         **if** $\mathrm{answer}_{right} = True \wedge \mathrm{answer}_{left} = True$ **then**
13:             $\mathrm{answer}_{left} \leftarrow \mathrm{sol_len}_l \leq \mathrm{sol_len}_r$
14:             $\mathrm{answer}_{right} \leftarrow \mathrm{sol_len}_r < \mathrm{sol_len}_l$ ▷ Have to take solution with smaller length
15:         **end if**
16:         **if** $\mathrm{answer}_{left} = True$ **then**
17:             incident_edges $\leftarrow$ left_half
18:             **continue**
19:         **end if**
20:         **if** $\mathrm{answer}_{right} = True$ **then**
21:             incident_edges $\leftarrow$ right_half
22:             **continue**
23:         **end if**
24:         rest, $\mathrm{special}_{removal} \leftarrow \mathrm{decide}_{special}$(left_half, right_half, e_{first})
25:         incident_edges $\leftarrow$ rest
26:     **end while**
27:     snesc_{sol}.add(incident_edges)
28:     $u, v \leftarrow$ incident_edges[0]
29:     $e_{cur} \leftarrow$ incident_edges[0]
30:     $\mathrm{v}_{cur} \leftarrow v$ **if** $v \neq \mathrm{v}_{cur}$ **else** u
31: **end while**

In figure 5 provides a sketch of the algorithm. The function decide(E_r, e_{start}) is the DP described in 4, where in the DP the set of edges E is modified to $E \setminus E_r$. Instead of branching over all vertices of the first constraint set S_1, we run a single DP for e_{first}. In the case where removing both subsets returns zero, a special function is called decide$_{special}$. This function performs the same operations as the body of the while loop but additionally reassigns the edge weights of the object. Aside from the edge list E we also maintain an adjacency list of the graph for quick lookup of incident edges.

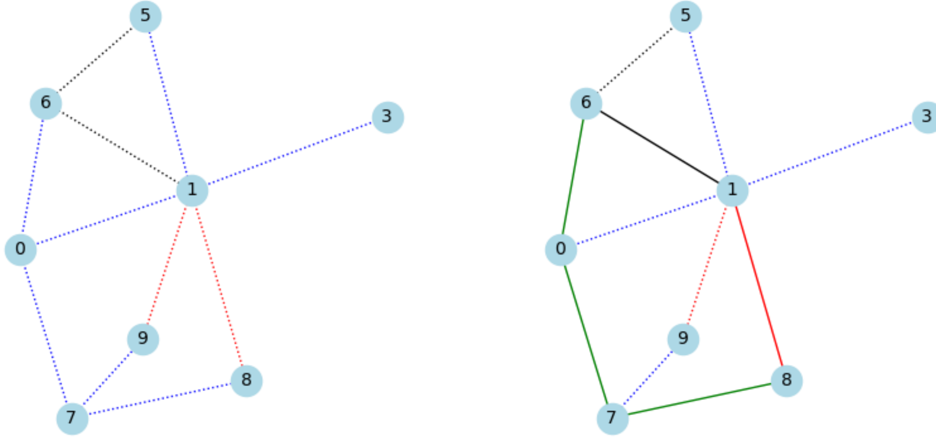

Figure 4.1: Visualization of a snesc graph. Black and red colored edges are contained in separate constraint sets. Left picture shows the graph, right picture shows the graph with a possible solution cycle.

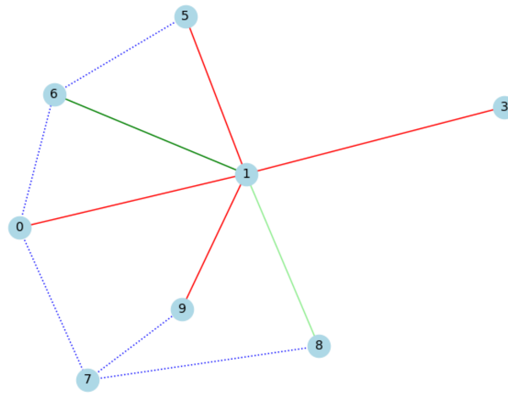

Figure 4.2: Visualization of the search. Red edges indicate edges, that are irrelevant for the solution. Light green edges mark already found solution edges. Dark green edge is the newest found solution edge. The first edge is (8,1) oriented towards 1.

To test our entire implementation, we got a small set of nonogram images from an automatic nonogram generator provided by M. Löffler. The test set can be found in Appendix A A.

4.2.3 Interesting cases

In this section, we discuss interesting cases where the algorithm fails to find a solution or even finds a suboptimal one, as well as cases where it eventually identifies the optimal solution despite initially detecting a longer walk. If an optimal solution with length l couldn't be

detected in the l round, then either a non-zero monomial eliminates itself, because one of its variables got the zero element of the finite field, or multiple monomials of length l may cancel each other out if they produce finite elements twice, a consequence of the characteristic-2 property of the field. See i.e 4.3 and let all edge weights be of value α. In the following examples our constraint set only consist of $S_1 = \{(A, B)\}$, marked in blue.

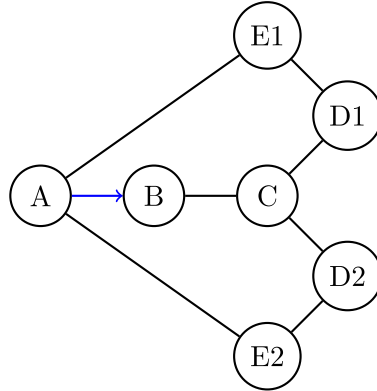

Figure 4.3: Case 1: Graph with two identical cycles

The algorithm will return zero for $T_A(l, [1], A) \quad \forall l \leq n$. In the relevant iteration $l = 5$ we have two cycles $ABCD_2E_2A$ and $ABCD_1E_1A$, which cancel each other because they construct the same value.

But for larger graphs, there is hope to find a solution. Assume the optimal cycle C has length l and it could not be detected, because some other optimal cycles C' , ... , C'' with C cancel each other out. The valid cycle can still be used for a valid walk. See Figure 4.4. We apply the same rules as stated above. The optimal cycle cannot be detected as in case 1 for $l = 5$. But for length $l = 7$ there exists one extra walk $ABCD_2E_2FE_2A$, which produces a non-zero value α^7 for $T_A(7, [1], A)$. We refer to such walks as valid palindrome cycles because their edges can be decomposed into a feasible cycle along with additional edges that form a palindrome, such as E_2FE_2.

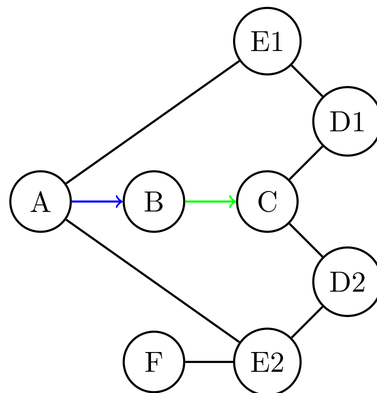

Figure 4.4: Case 2: The green edge marks edges, that are found to the solution. C is the vertex, where the two optimal cycles diverge. Removing one of the edges results detectability of an optimal cycle.

We call this a rebound effect, because we get the chance to detect snippets of a valid walk. Indeed we can even retrieve the optimal walk without the palindrome. Suppose a valid palindrome cycle of length $l + 2n$ with an optimal cycle C with length l contributes a non-zero value and let v be the first vertex that C and the palindrome shares. Also C cancels out with a set of optimal cycles $\mathcal{C} = C', \ldots, C''$ with $\phi(C) = \sum_{C' \in \mathcal{C}} \phi(C')$. Let v' be the first vertex, where after that C and $C' \in \mathcal{C}$ diverge. Then if v comes after v', we get the chance to branch between the optimal cycles C and C'. By eliminating one branch, we can break the equation $\phi(C) = \sum_{C' \in \mathcal{C}} \phi(C')$ and produce a non-zero value for a smaller l. An example graph is shown in 4.4. See 4.5 for further interesting examples, where we can find the optimal solution.

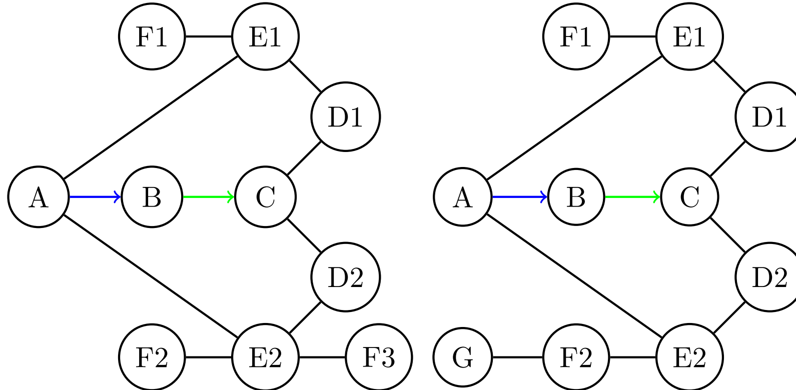

Figure 4.5: The DP for the graph on the left picture returns a non-zero value, because of an odd number of valid palindrome cycles with $l = 7$. In the right picture the DP returns $l = 7$ zero for the graph, but a non-zero value for $l = 9$. Removing one of the edges in C again results detectability of an optimal cycle.

One could also consider the palindrome as a distinct, detectable path (see, e.g., 4.6). The reasoning remains the same as above: the longer path must lead to a vertex where branching between the optimal cycles becomes possible. When this occurs, the optimal cycle can be identified, allowing the search to adapt accordingly.

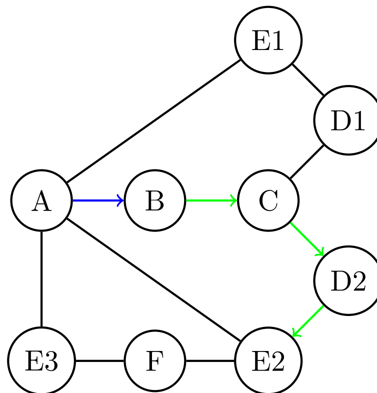

Figure 4.6: Case 3: The green edge marks edges, that are found to the solution. C is the vertex, where the two optimal cycles diverge. Removing one of the edges results detectability of an optimal cycle. Here it will always find the optimal valid cycle

However, there exists also path placements where a suboptimal solution may be found. This occurs when, during the binary search, no branching happens between optimal solution cycles that cancel each other out. An example of this is shown in 4.7.

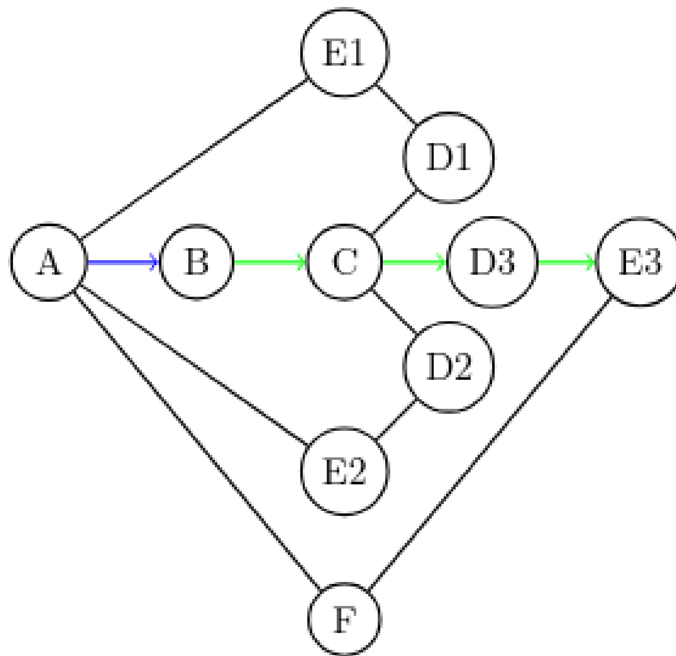

Figure 4.7: Case 4: Three cycles. The branching on C is between [CD1, CD2] and [CD3]. It will take CD3, because the removal of CD3 leads to case 1 and the removal of [CD1, CD2] let the suboptimal cycle remain.

4.3 Exact method

Integer linear programming ILP is a mathematical optimization technique, in which all variables are integers. It consists of a linear objective function and linear constraints. To verify the optimality of the approach described above, we also provide an exact method for solving SNESC using ILP. Let $((V, E), S_1, \dots, S_k)$ be a SNESC-instance given with a total order $\preceq$ on the vertices. Additionally, we define a set V' of vertices v, each corresponds to an edge uv in S_1, where v is the smaller in the ordering to u, $v \preceq u$. Let f be the function, that returns the smaller vertex from the edge, meaning $V' = \{v \mid e \in S_1 \wedge v = f(e)\}$. Our ILP-formulation is then as follows:

$$
\begin{aligned}
&\text{Minimize} && \sum_{e \in E} x_e && && (1)\\
&\text{subject to} && \sum_{e \in S_i} x_e = 1 && \forall i \in [k] && (2)\\
& && \sum_{u \in N(v), e=uv} x_e = 2y_v && \forall v \in V && (3)\\
& && 2x_e \leq y_v + x_u && \forall e = uv \in E && (4)\\
& && \sum_{uv \in E} f_{uv} = \sum_{vu \in E} f_{vu} + y_v && \forall v \in V \setminus V' && (5)\\
& && -nf_e \leq f_{uv} \leq nf_e && \forall e = uv \in E && (6)\\
& && -nf_e \leq f_{vu} \leq nf_e && \forall e = uv \in E && (7)\\
& && -nf_e \leq s_e \leq nf_e && \forall e \in E && (8)\\
& && \sum_{uv \in E} f_{uv} = \sum_{vu \in E} f_{vu} + y_v - \sum_{e \in S_1, v=f(e)} s_e && \forall v \in V' && (9)\\
& && x_e \in \{0, 1\} && \forall e \in E &&\\
& && y_v \in \{0, 1\} && \forall v \in V &&\\
& && f_{uv}, f_{vu} \in \mathbb{Z} && \forall uv \in E &&\\
& && s_e \in \mathbb{Z} && \forall e \in S_1 &&
\end{aligned}
$$

First we explain the variables from the ILP: we introduce for every edge e and vertex v the choice of including it in the solution as a binary variable x_e and y_v. Next we define for every edge uv a flow variable f_{uv} and f_{vu} to model a flow over the graph. We also introduce some sink variables s_e for every edge in S_1, that is once for an endpoint $v = f(e)$. These values are constrained to be integer values. Each of these variables plays a crucial role in defining a feasible solution. The edge and vertex selection variables determine the structure of the solution, while the flow and sink variables ensure connectivity, which we explain next.

The optimal solution is characterized by minimizing the number of vertices across all simple cycles that satisfy the constraints. Therefore, this solution also minimizes the number of selected edges (1). Condition 2 ensures the set constraints. For each constraint set S_i exactly one edge has to be active. Condition (3) forces that every vertex has degree either zero or two. Condition (4) ensures that, if an edge is active, its endpoints have to be active as

well. Together they force simple cycles, but the constraints allow multiple disconnected cycles. Conditions (5) − (9) describe a flow between all active vertices and are needed to ensure us, that we only produce one simple cycle. Every vertex contributes a value to the flow if and only if it is active and used for the solution y_v by Condition (5) and (8). Only active edges can transfer flow by Condition (6) and (7). Because the components of the graphs are cyclic, a valid component cannot only consist of active vertices $v \in V \setminus V'$. This is because they only add contribution to the flow that cannot vanish. To fulfill the constraint, a vertex $v \in V'$ is needed that can act as a sink if and only if one of its incident edges $e \in S_1$ is used by constraint (9). Because an active vertex in V' corresponds to an active edge S_1, only one $v \in V'$ and one of its edges can act as a sink. Additionally, all active vertices must be connected to this sink, ensuring that the graph forms a single connected component. In other words, every active vertex is reachable from the active sink. Due to our cycle characterization, the solution can only contain one simple cycle.

Thus, if we have a valid ILP-solution, we also have a valid SNESC-solution, because the constraints enforce one simple cycle, with exactly one edge taken from each constraint set. The reverse direction holds by construction. The implementation can be found in src/ilp.solve_snecs_using_gurobi and it makes use of the Python Gurobi library. Certainly, this program can be improved, but it is sufficient for our applications.

5 Experiments and Results

5.1 Experimental Setup

We aim to analyze the running time and error probability of the randomized FPT-method in a practical setting. To achieve this, we build up a synthetic graph generator, which produces random graphs of different quality. We consider two classes of graph types:

- Erdős Renyi (ER) graphs are random graphs following a specific probability model. Each edge will be taken with a certain probability, we choose the value $\frac{5}{n}$, such that we have a linear number of edges on average, in expectation $\frac{5}{2}(n-1)$.
- Barabási–Albert (BA) graph is a scale-free random graph, where edges are preferentially attached to existing vertices based on a given attachment rate[14]. The degree distribution will then follow a power law distribution and by choosing the attachment rate as 5, where we expect $5n-5$ edges.

We implemented a generator function, which creates instances with ids and stores them in a single CSV file. Every instance contains metadata, including the graph seed, set seed, graph size, constraint set size. The graphs are generated with the NetworkX library and require a graph seed and structural specification from the metadata. The edges for the constraints sets will be randomly picked. The size of the constraint set size will be fixed on exactly 3 edges. Note, if a seed is set before the random choices, the generation process becomes fully deterministic. We also explored several different storing approaches like human-readable serialization and binary serialization, but ultimately found that storing only the seed is the most space-efficient method.

Furthermore, we also process each instance by the ILP-method to label the optimal solution length. Graph with no solution will be filtered in our instances, because their runtime is very predictable. We also provide a separate small analysis over the gurobi ILP method in B.

We also hash the solution, graph, and constraint sets, so that we can verify the reproducibility of the objects. All instance information is captured by a single row in a

CSV file. To retrieve an exact instance, we provide a graphloader, which automatically maps every row in the CSV file into an object.

For our experiments we focus on graphs with size 15 and ranges k over $3, \dots, 10$. We restrict ourselves to this range, because the running time explodes rapidly for bigger k, i.e. for one experiment with $k = 11$ we ruffly spend 1 hour. We generated about ten thousand instances for every k. See figure B.3 for an overview of the instance number.

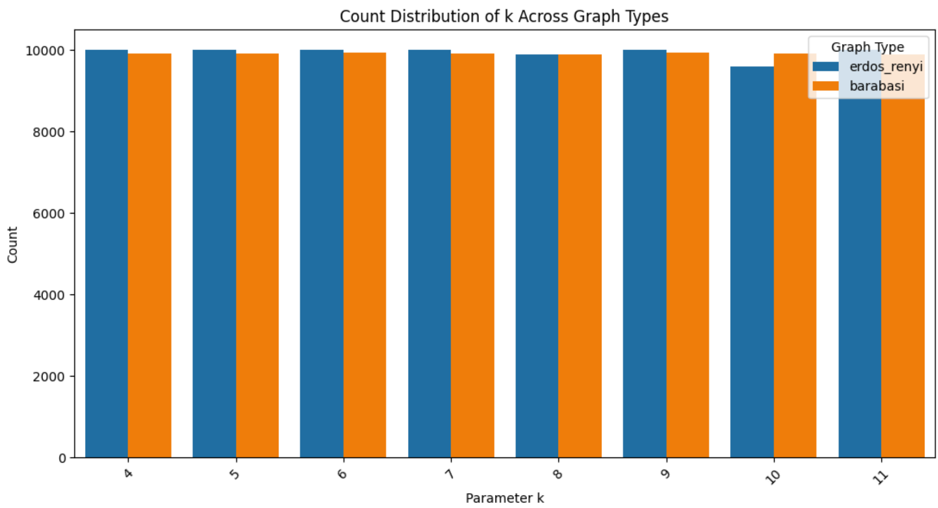

Figure 5.1: Barchart of the number of instances for each k

One experiment consists of an instance with its id, a random seed for the algorithm, and the size of the finite field. Each of them is also stored in a row of a different CSV file. All experiments were conducted on an AMD Ryzen 3 PRO 4450U processor (4 cores, 8 threads) and 16 GB RAM.

We also created a second group of experiments, where we filtered the zero element of the finite field in the assignment of edge weights. Additionally, we also created an instance runner, which calls instances with the decision and search automatically. It chooses an instance randomly from the CSV file of the generator and runs it three times with different random seeds. The random seed decides the edge weights.

5.2 Decision Results

Figure 5.2 gives statistics on the runtime of the decision as a function of k from the runner. We splitted the experiments in two groups and for each point we averaged the runtime over the experiment with the same k value and graph type. As expected, both groups grow exponentially in the runtime to the parameter k. This is demonstrated by the linear behavior in the log-scale plot.

A curious anomaly occurs at the beginning of the top diagrams, where all elements from the finite field are in use. This peak might suggest that CPU resources were not fairly distributed. Further investigation is required to understand the specific instances that are causing this peak.

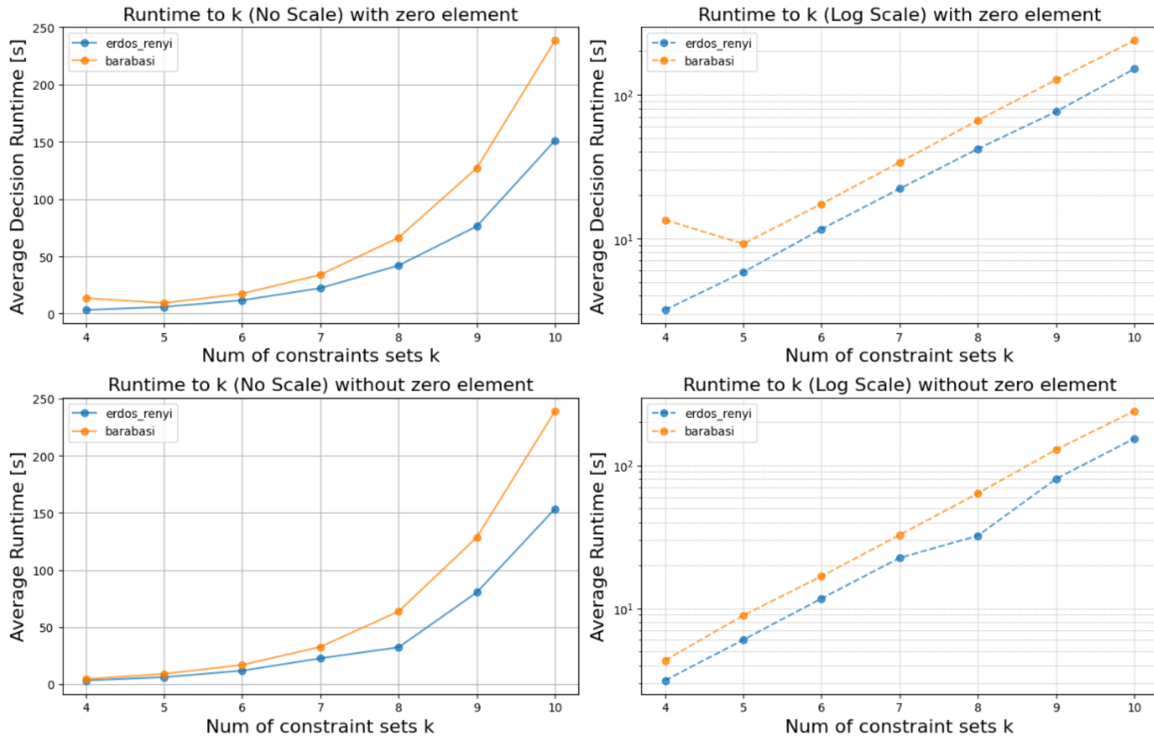

Figure 5.2: Runtime Decision over the number of constraint sets k

For a fixed k and graph type, the runtime is not significantly affected by the field size. Table 5.1 shows the runtime over all BA graphs with $k = 8$. This is just a subset of the data from the figure.

Field Size	Runtime with Zero [s]	Runtime without Zero [s]	Difference in [s]
4	65.50	68.04	2.53
8	66.76	67.30	0.54
16	67.45	67.23	-0.22
32	66.11	64.28	-1.82
64	65.51	62.85	-2.65

Table 5.1: Barabasi graph decision runtime over field size with $k = 8$

5.2.1 Element distribution and error probability

We were also interested in the distribution of the elements, produced by the algorithm. We can infer error probability directly from the frequency of the zero element, because from the beginning we filtered false instances from our data set. Our results indicate that the distribution is not uniform, with certain elements being more frequently selected. Table 5.2 presents error probabilities as a function of field size.

If we now only focus on the error probability, we see that the percentage of error is always cut in half when doubling the field size, see table 5.2. Here we partitioned only all experiments, which include the zero element as an edge weight, by their field size. We counted their occurrences and computed for each element its frequency. We observe that the zero element is always the element with the smallest frequency in this group.

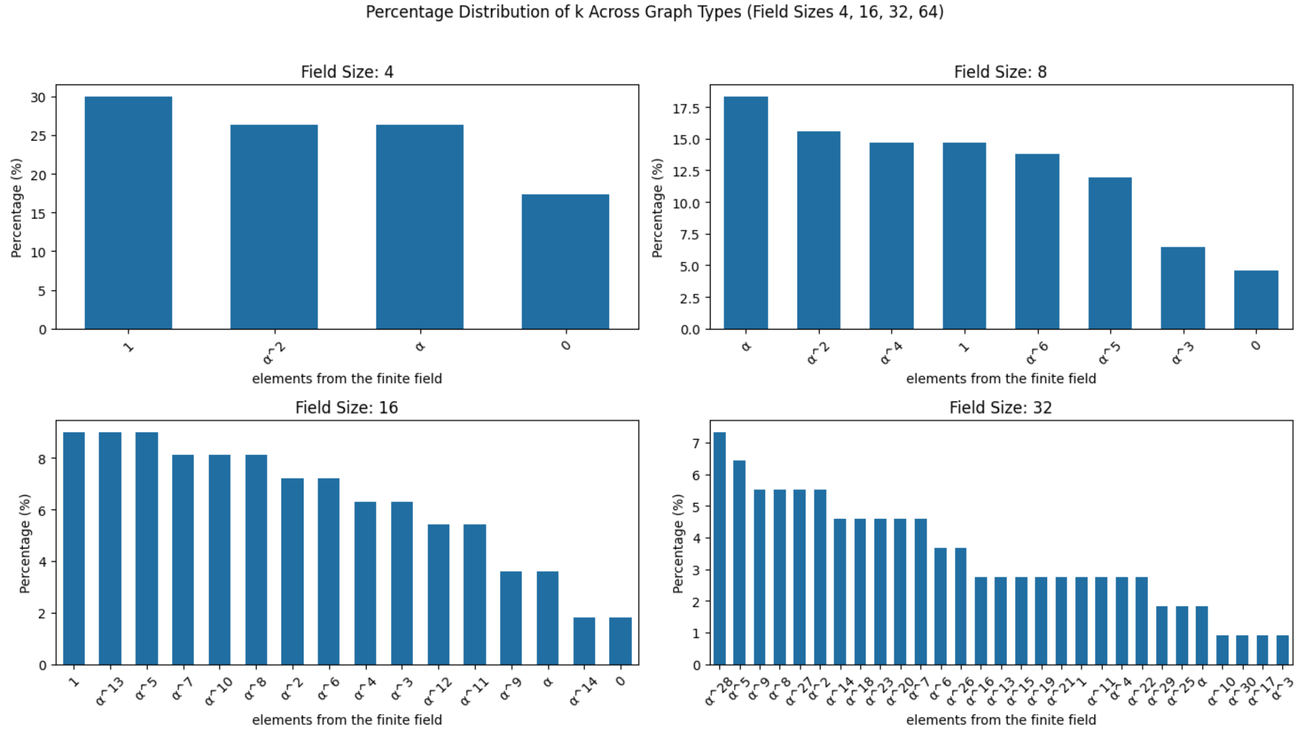

Figure 5.3: Elements distribution from the algorithm with

Field Size	smallest frequency of elements [%]	highest frequency of elements [%]	Error Probability Decision [%]
4	17.27	30.00	17.27
8	4.59	18.35	4.59
16	1.80	9.01	1.80
32	0.00	7.34	0.00

Table 5.2: Mean, Min, and Max Percentage for Different Field Sizes

Note also for the table 5.2 for a field size q its practical error $\epsilon(q) \leq \frac{1}{q} < \frac{l}{q}$, where l is the solution length. The theoretical bound is given by the lemma 4.1. This suggests that the practical error rate is significantly lower than the theoretical upper bound, likely due to the limited graph and solution size. Further studies may be required to approve this observation.

When analyzing the second experiment group, where the zero element is excluded, we observe a significant reduction in error probability. In this setting, failed cases become extremely rare. To provide concrete numbers, out of 1500 experiments, there was no failed case, where the field size was uniformly chosen from $\{4, 8, 16, 32\}$. Figure 5.4 shows the distribution, how frequent each element is over the experiments. It confirms that no incorrect zero-element assignments occurred in our experimental dataset. However, past unbalanced datasets have shown some errors, suggesting that a more rigorous statistical test should be conducted to validate this phenomenon. In the bottom-right table, the distribution appears to be stepwise. This effect can arise due to the relatively small number of experiments—around 100 were conducted. So each element has a probability between $\frac{1}{100} \cdots \frac{8}{100}$.

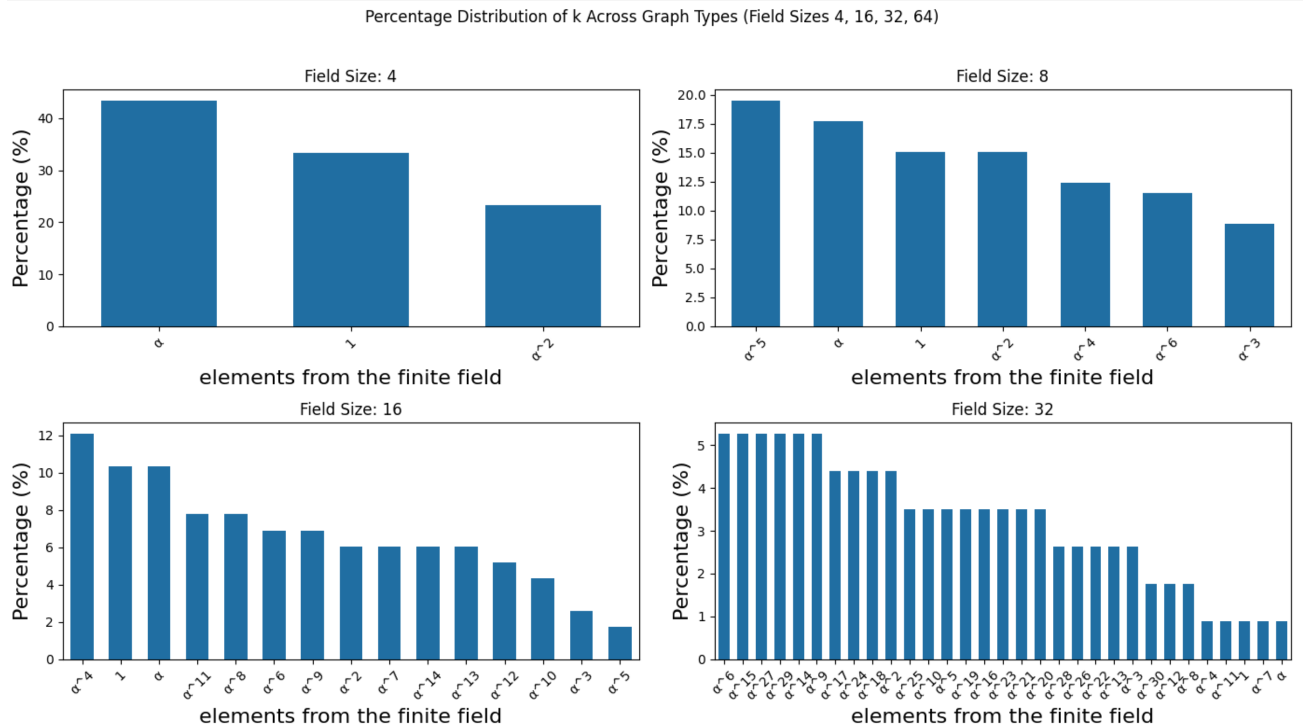

Figure 5.4: Finite elements distribution from the algorithm without use of the zero element.

5.3 Search Results

We provide data of the search in figure 5.5, showing the running time for a search per query on the DP. As in the decision process, we see an exponential growth in the parameter of k even for the average over all queries. However, the curves are noticeably less smooth compared to the decision process. This can be explained by the inconsistency in individual run after a certain number of queries, the decision-making becomes easier due to the increasing sparsity of the graph as edges are deleted.

For each run, we also counted the number of calls to the DP function. In table 5.3 we present the average number of queries to the number of constraint sets. The number of queries grows with the number of k. This is probably because larger edge constraints typically correspond to longer solution paths. We also can observe that in each row the number of queries from the BA graph is larger than the ER graph. This indicates, that in expectation BA could be much denser due to our choice of parameters.

k	Average Num of Queries of Erdos Renyi Graph	Average Num of Queries of Barabasi Graph
4	16.71	21.66
5	21.05	23.02
6	24.89	28.00
7	24.43	27.31
8	23.95	31.00
9	26.73	36.57
10	30.00	35.53

Table 5.3: Number of Queries over k

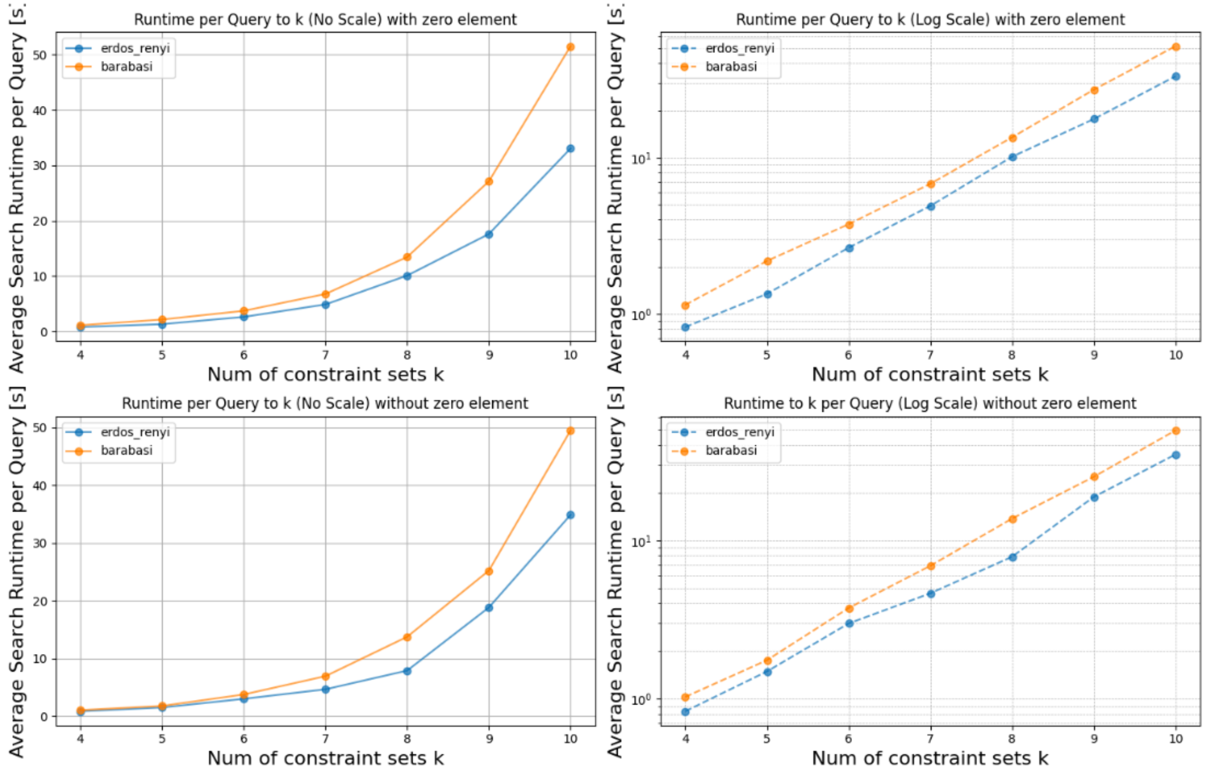

Figure 5.5: Search Running time per Query

5.3.1 Overall error Probability

Table 5.4 shows error probabilities for search operations with and without the zero element. Similar to the decision process, excluding the zero element significantly reduces error probability.

field size	**Error Probability** (with zero element) [%]	**Error Probability** (without zero element) [%]	Percentage of not optimal solutions [%]
4	62.64	3.33	45.37
8	32.69	0.00	28.1
16	18.35	0.00	16.55
32	6.42	1.25	6.42
64	7.41	0.00	7.41

Table 5.4: Error Probabilities for Search with and without Zero Element

Interestingly in the table is that the error probability increases with the larger field size 64 from 32. This could indicate a bug in the program or some unexpected effect. Additionally, we want to highlight that for a field size of 2^{32}, we never encountered any suboptimal or incorrect decisions.

Besides a suboptimal solution is approximately double that likely than a falsely decided solutions. This is because a failed decision requires that all valid cycles have to cancel out for all lengths l, including the replication by the rebound effect. In contrast, obtaining a non-optimal solution requires only a subset of cycles, where all optimal cycles are included, to cancel out, making non-optimality more common.

Also interesting is that the rebound effect is weaker than we expected. Out of 150 experiments, decided to be not-optimal, only two were reconstructed optimally. The real probability could be also way lower or a bit higher. No case was found in the group without the zero element.

6

Conclusion

In this thesis, we introduced the theoretical foundations behind removing popular faces in curve arrangements. Furthermore, we explained the implementation of our ideas and demonstrated how we resolved all IPE files provided for us. We present several interesting example cases of the algorithm and gave some practical insights on the running time and error probability. A crucial finding is that the empirically measured error probability deviates by a factor of l from the theoretical bounds. We can even decrease the error probability significantly by filtering the zero element from the finite field.

6.1 Open work

Explaining the gap to the theoretical. A theoretical explanation is still missing for why the observed error probability is lower than expected and why removing the zero element significantly decreases it.

Support of multiple curves. This is motivated by the fact that we then can solve more complex popular faces, meaning popular faces with more than 2 curve segments of the same curve. The algorithm currently supports only one curve or by manual setup multiple restricted curves, that can't intersect between them by cloning the outer face. A correct extension would be to iterate through all partitions of the edge sets in separate SNESC-instances and solve them independently.

Automatic curvature and smart choices. We mentioned that the choice of dual edges in the solution drawings is done arbitrarily and that we only support the drawing of straight lines. A first improvement would be to automatically compute and insert smooth and continuous solution curves.

Support of data reduction. The SNESC-instance could be reduced in many aspects. The elimination of degree 1 and 2 vertices. Also in the recovery of the solution, we mentioned some techniques to accelerate the search. Note that for data reduction, we also have to think about

substituting reduced vertices or edges back. We also have to extend the model of the dual graph to a multi graph, because by reducing we may get edges with the same endpoints, but inherit different dependence on the edge sets.

Dependence of randomness. As mentioned, fully derandomizing PIT is unlikely. However, it may be possible for certain restricted classes of polynomials, making this an interesting direction for future research.

Bibliography

[1] P. de Nooijer et al. “Removing Popular Faces in Curve Arrangements”. In: *Journal of Graph Algorithms and Applications* 28.2 (Nov. 2024), pp. 47–82. DOI: 10.7155/jgaa.v28i2.2988.

[2] ”N. Ueda and T. Nagao.” “”NP-completeness results for nonogram via parsimo- nious reductions.”” In: *”Technical Report TR96-0008” Department of Computer Sci- ence, Tokyo Institute of Technology”* (”1996”). URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=1bb23460c7f0462d95832bb876ec2ee0e5bc46cf.

[3] T. de Jong. “The concept and automatic generation of the Curved Nonogram puzzle”. Available at https://studenttheses.uu.nl/bitstream/handle/20.500.12932/23941/CurvedNonograms-thesis.pdf?sequence=2&isAllowed=y. MA thesis. Utrecht University, 2016.

[4] M. van de Kerkhof et al. “Design and Automated Generation of Japanese Picture Puzzles”. In: *Computer Graphics Forum* (2019). DOI: doi:10.1111/cgf.13642..

[5] *halfedge*. URL: http://graphics.stanford.edu/courses/cs368-00-spring/TA/manuals/CGAL/ref-manual2/Halfedge%5C_DS/halfedge.gif.

[6] *Duals_graphs*. URL: https://en.wikipedia.org/wiki/Dual_graph#/media/File:Duals_graphs.svg.

[7] M. van de Kerkhof. “Improved automatic generation of curved nonograms.” Available at https://studenttheses.uu.nl/handle/20.500.12932/28187. MA thesis. Utrecht University, 2017.

[8] J. T. Schwartz. “Fast Probabilistic Algorithms for Verification of Polynomial Identities”. In: *J. ACM* 27.4 (Oct. 1980), pp. 701–717. ISSN: 0004-5411. DOI: 10.1145/322217.322225.

[9] Richard Zippel. “Probabilistic algorithms for sparse polynomials”. In: *Symbolic and Algebraic Computation*. Ed. by Edward W. Ng. Berlin, Heidelberg: Springer Berlin Heidelberg, 1979, pp. 216–226. ISBN: 978-3-540-35128-3. URL: https://link.springer.com/chapter/10.1007/3-540-09519-5_73.

[10] Richard A. Demillo and Richard J. Lipton. “A probabilistic remark on algebraic program testing”. In: *Information Processing Letters* 7.4 (1978), pp. 193–195. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(78)90067-4.

[11] Valentine Kabanets and Russell Impagliazzo. “Derandomizing polynomial identity tests means proving circuit lower bounds”. In: *Comput. Complex.* 13.1/2 (Dec. 2004), pp. 1–46. ISSN: 1016-3328. DOI: 10.1007/s00037-004-0182-6.

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

[13] Matt Hostetter. *Galois: A performant NumPy extension for Galois fields*. Nov. 2020. URL: https://github.com/mhostetter/galois.

[14] Albert-Laszlo Barabasi and Reka Albert. “Albert, R.: Emergence of Scaling in Random Networks. Science 286, 509-512”. In: *Science (New York, N.Y.)* 286 (Nov. 1999), pp. 509–12. DOI: 10.1126/science.286.5439.509.

Nonogram test set

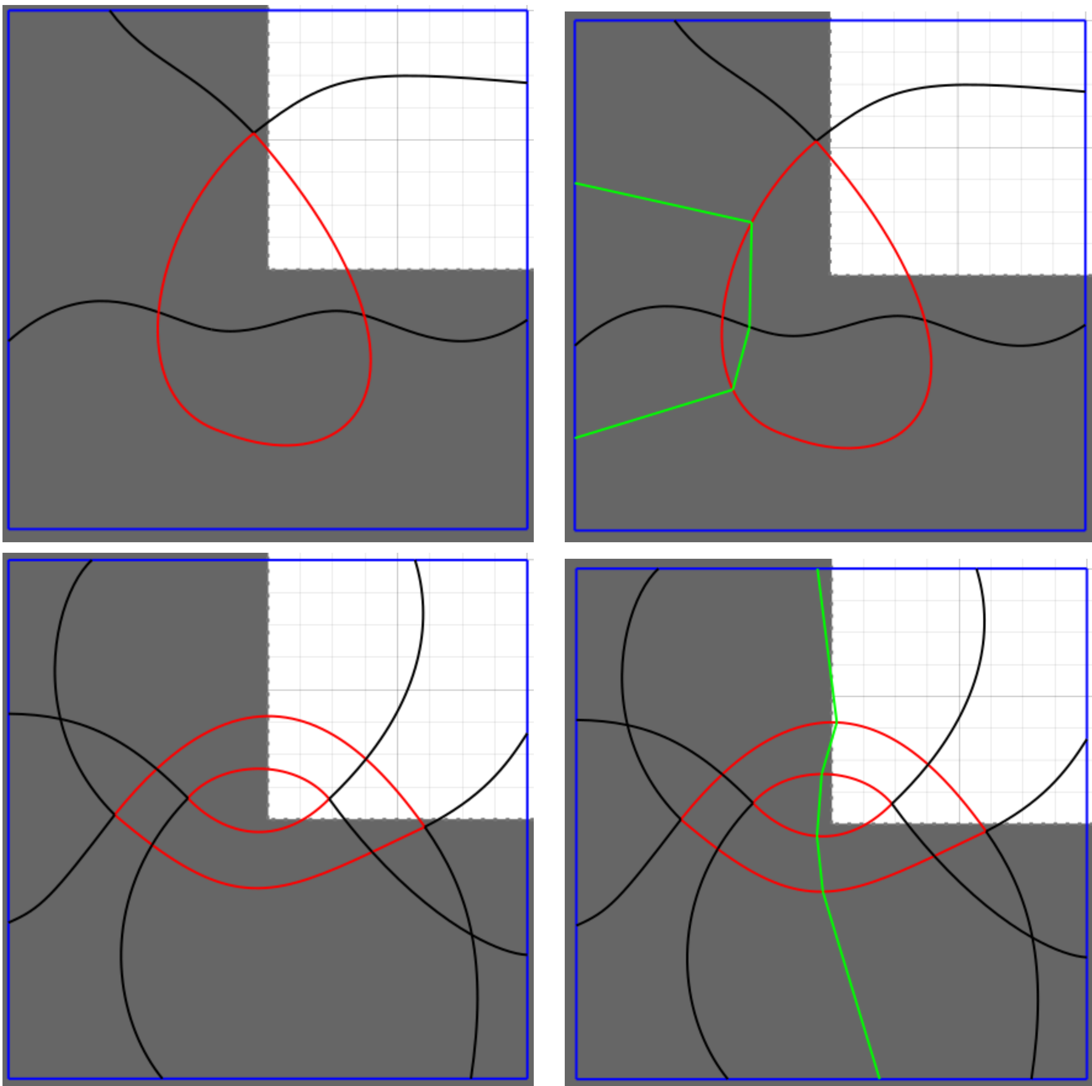

Figure A.1: Left the original ipe image from M. Löffler; right the new created ipe image from the pipeline

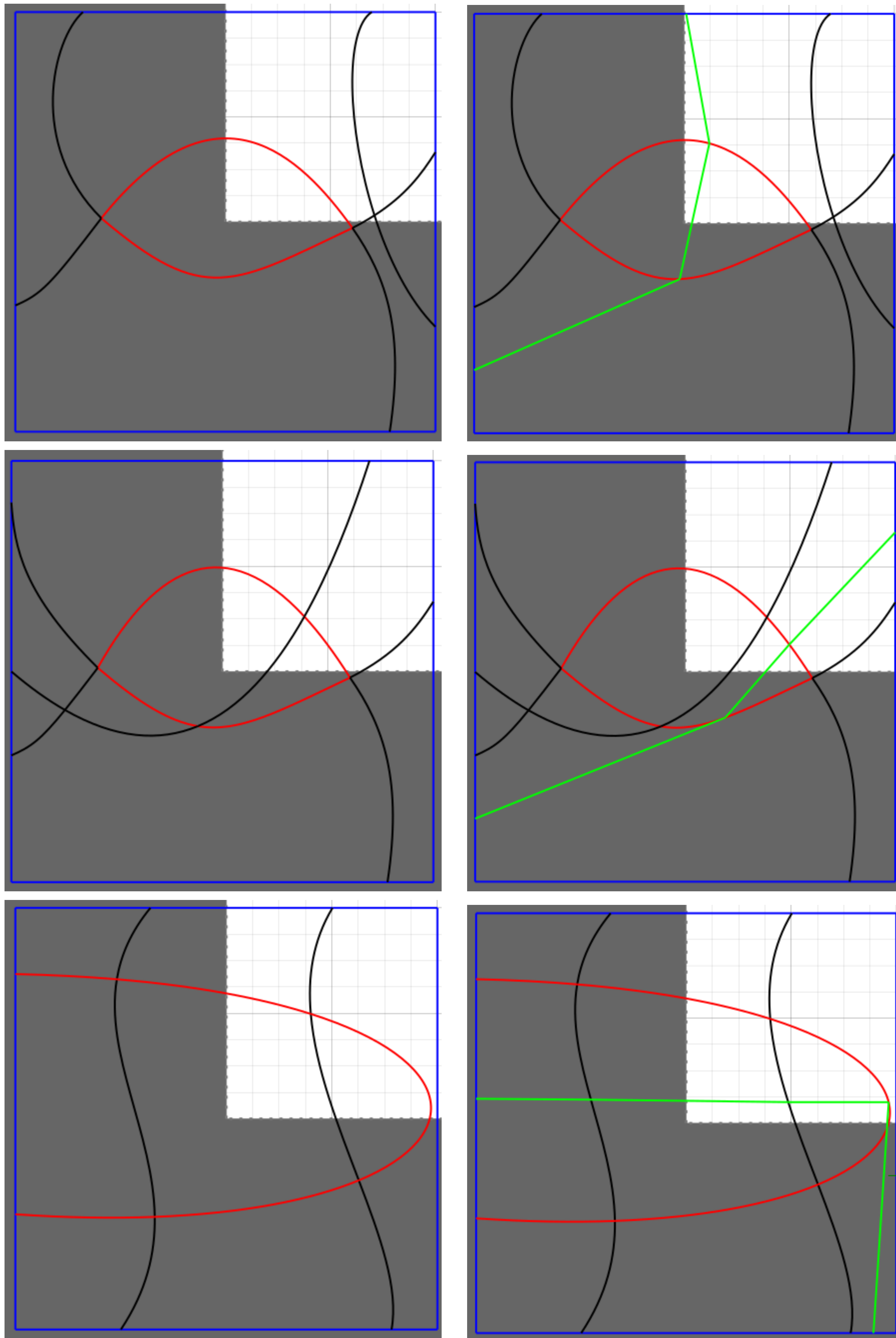

Figure A.2: Left the original ipe image from M. Löffler; right the new created ipe image from the pipeline

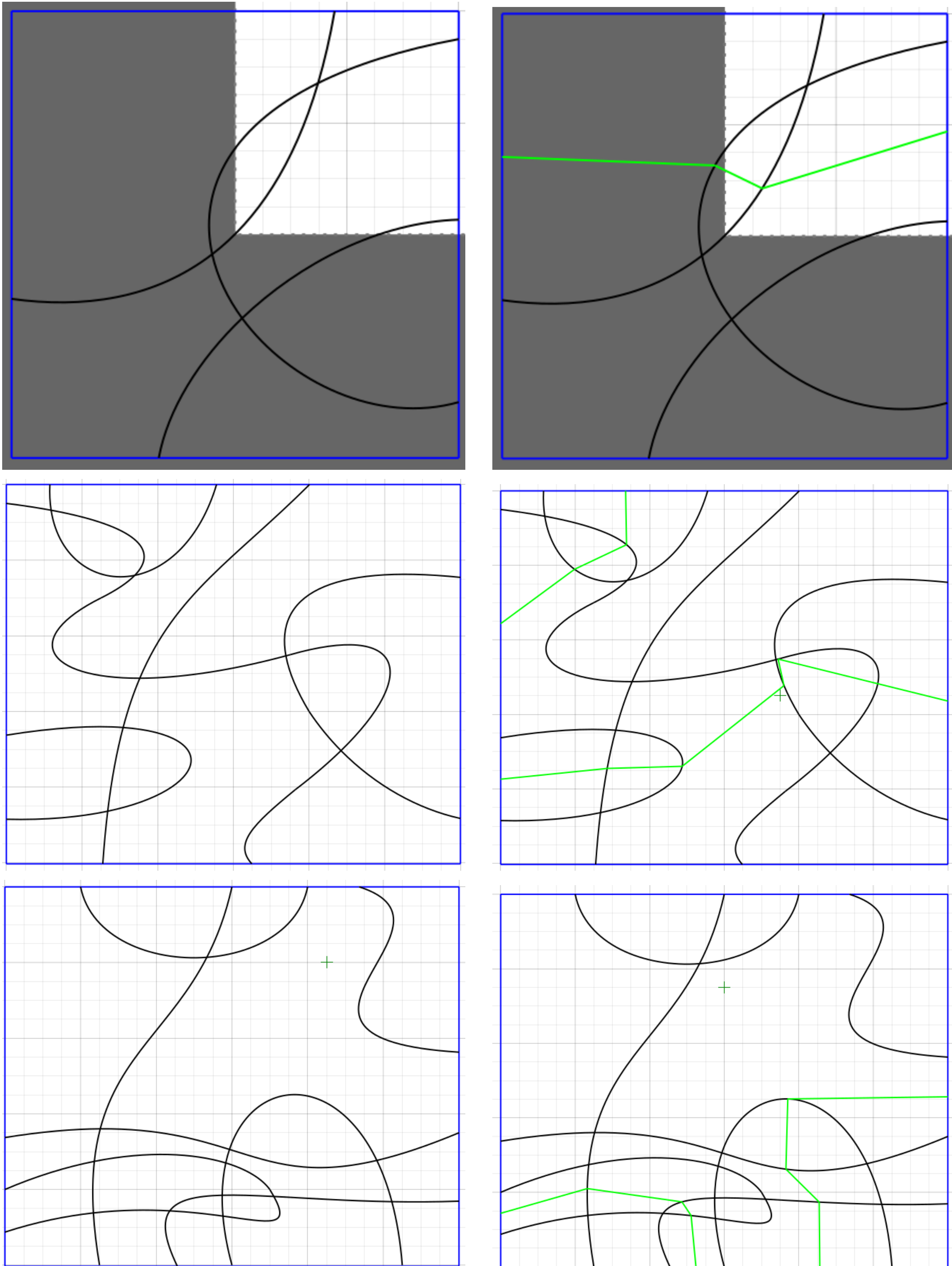

Figure A.3: Left the original ipe image from M. Löffler; right the new created ipe image from the pipeline

B
ILP Performance

The randomized FPT -method is an interesting approach; however, the ILP -based method demonstrated impressive performance, successfully labeling all instances in the experiment from Chapter 5. In this section, we provide statistics to illustrate the practical efficiency of the solver. In this section, we give some statistics on how strong the solver in practice is. In figure B.1 is the average running time of all positive labeled data shown to the parameter k. To offer a rough comparison with the DP approach, for $k = 12$, the latter required approximately two hours to compute a solution, whereas the ILP-based method completed the task in mere milliseconds.

For larger values of k, a reasonable approximation suggests that the runtime of the DP and search-based method doubles with each increment of k . Remarkably, the runtime of the ILP-based method appears to grow linearly with the number of constraint sets, highlighting its efficiency.

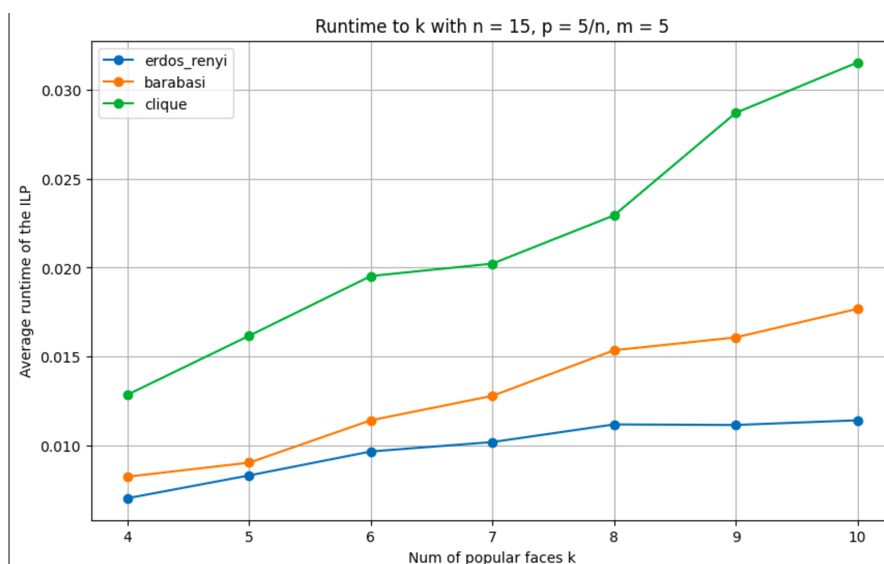

Figure B.1: Running time from ILP on the same test data set as the DP

To gain a better understanding, we also provide a boxplot showing the running time. We observe from B.2 that for some clique instances, the runtime seems to grow more than linearly. This suggests that the harder instances might be necessary to truly assess the limits of what the ILP-method can handle.

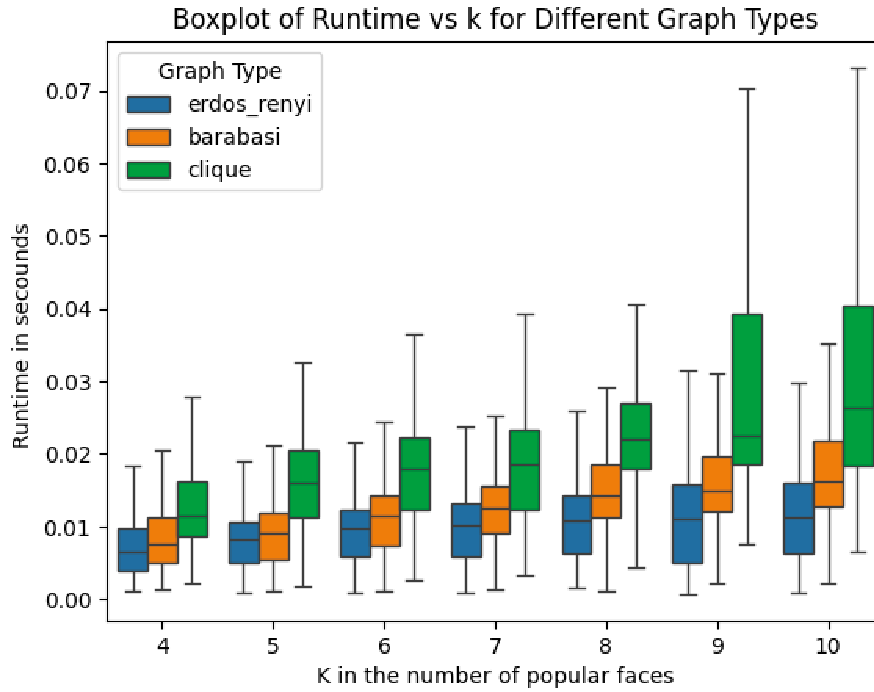

Figure B.2: Running time on the same test set as the DP with boxplots

To further investigate, we construct a more challenging set of instances with values of k in $\{4, 6, 8, 10, 16, 20, 32\}$, while keeping the graph size fixed at 60. All other parameters used in the instance construction remain unchanged. The data set is shown in and is also generated by the instance generator.

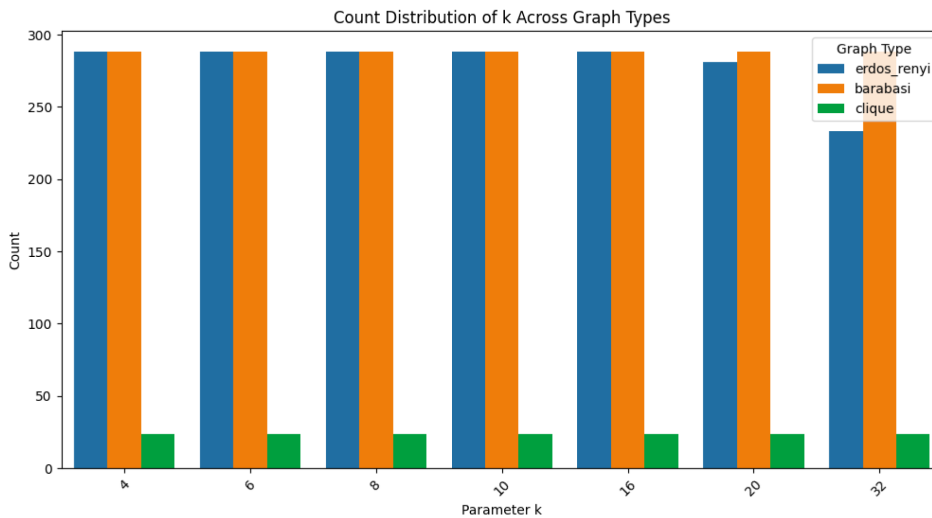

Figure B.3: Number of new generated instances

Figure B.4 is showing the runtime, which now follows more of an exponential curve. Interestingly, we now observe that the (ER) graphs tend to be harder than the (BA) graphs.

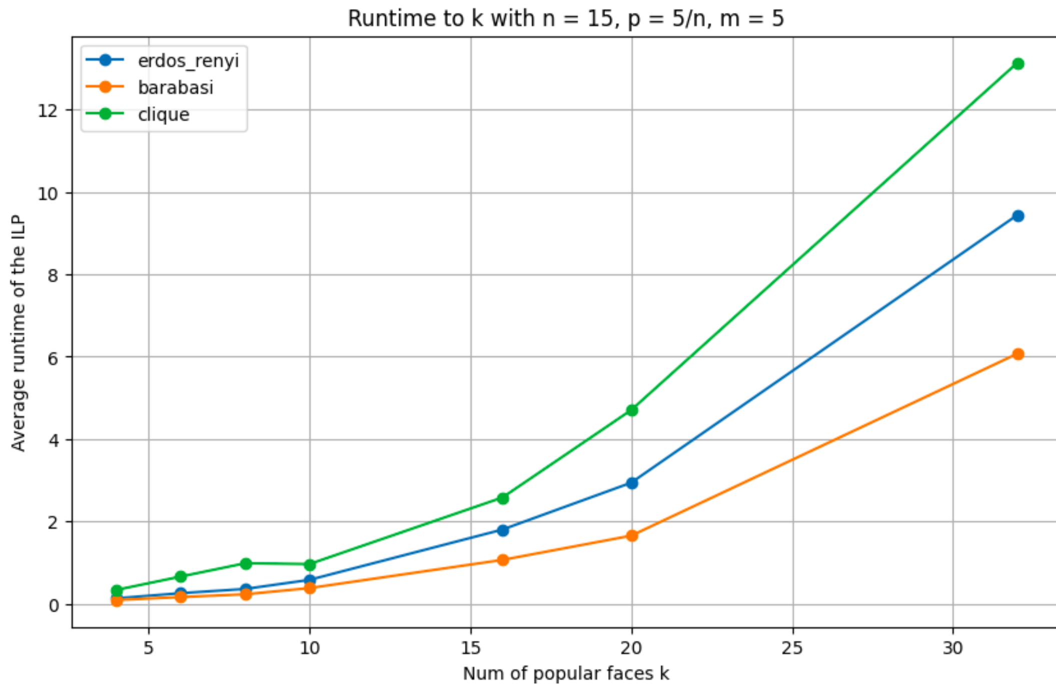

Figure B.4: Results on the same test set as the DP

We also plotted the data on a log scale and used a boxplot for further insights. The deviation remains in the BA and ER graphs much bigger than in the clique, because of the random choice of the graph. The clique graphs consistently remain very hard, as the high density in these graphs leads to a large number of possible configurations.

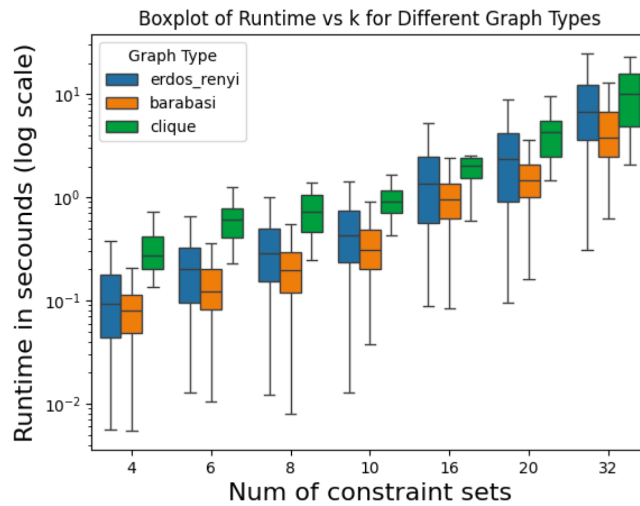

Figure B.5: Results on the same test set as the DP

An intriguing observation is that even though the ER graphs often have simpler instances, there are cases where they are just as hard as the clique instances. This could potentially be explained by the fact that an ER graph can also construct a clique. We also see no clear linear curve in the log plot, which implies no real strict exponential function.

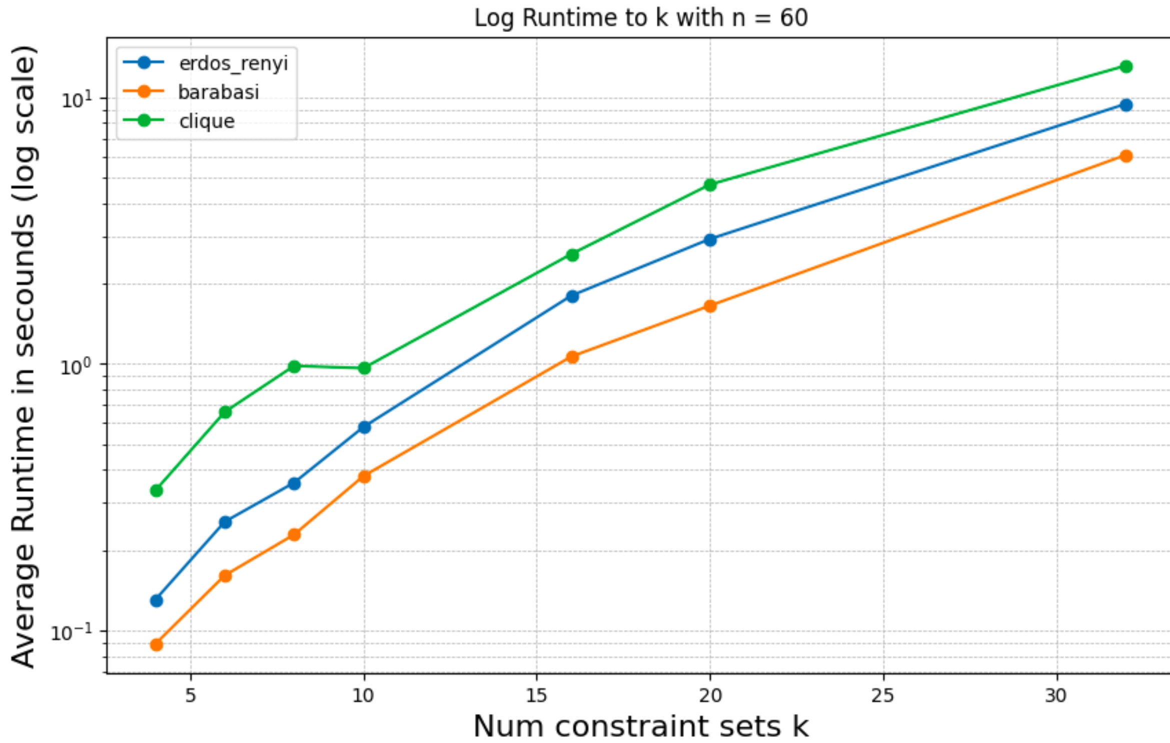

Figure B.6: Results on the same test set as the DP

This hints at the possibility that there may be other techniques or heuristics existing that could further improve the performance.