Utrecht University

Computing Science
Utrecht University

Improving the layout of Curved Nonograms

Author: David Bos

Supervisor: Tamara Mchedlidze
Supervisor: Maarten Löffler

MASTER THESIS, ICA-4137930

A thesis submitted for the degree of Master of Science in Computing Science

May 23, 2022

Abstract

In this thesis we build upon earlier work that generated curved nonogram puzzles. By taking generated puzzles we improve their layout with a user guided tool utilizing a force-directed approach. We preserve the topology of the puzzles to ensure they stay solvable. Generally, the puzzles end up looking more organic and visually pleasing.

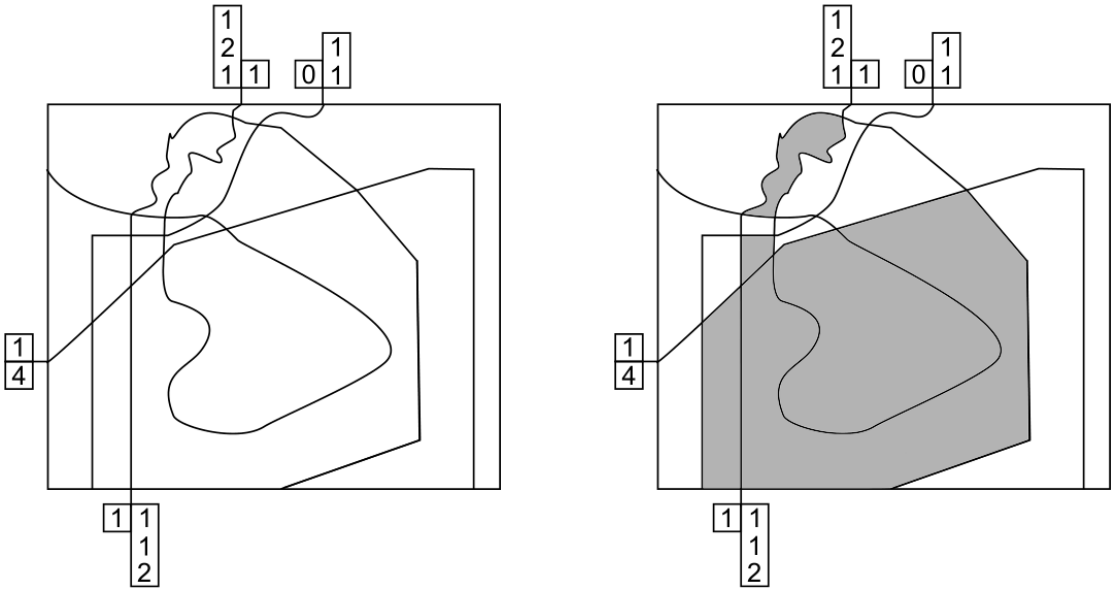


Figure 1: Example of a curved nonogram and its solution depicting a house with a smoking chimney.

1 Introduction

Curved nonograms are a variant on the well known nonogram or Picross puzzles introduced by de Jong [3], an example can be seen in Figure 1. De Jong also introduced an algorithm that can generate the curved nonograms by taking a desired solution vector image as input. Further research was done by van de Kerkhof [16], who introduced a new algorithm that is also capable of automatically generating curved nonograms. Both of their algorihms take a vector image as input, cut up the curves in the image, extend the curves with new curves to connect them to a border, and add hint labels to the curves to create a puzzle. They also try to ensure that the newly added curves look good and do not create ambigiously interpretable situations. However, the resulting puzzles can still contain some ambiguity, like curves running very close to each other so that it is unclear whether they are intersecting or not and curves intersecting at small angles such that it is unclear whether they intersected or just ran very close to each other. We build upon their work by taking the generated puzzles as input and moving around curves to get an unambigous and aesthetically more pleasing puzzle. To achieve this we developed a tool which can improve the layout of an existing puzzle while preserving the properties needed to solve it. We use a force-directed method to optimize the layout of the puzzles, which has not been previously tried. The tool works by moving around curves in real-time based on user defined parameters of the forces, which can be changed at any time. The ability to change parameters

in combination with being able to view the real-time effect of it, helps the user get an intuitive understanding of the forces and find good values for parameters interactively.

2 Related work

As mentioned earlier, de Jong and van de Kerkhof both worked on the automated generation and layout of curved nonograms (an example of which can be seen in Figure 4). Both used Bézier curves to create their puzzles, but utilized different algorithms to optimize their layout. De Jong uses a hill climbing algorithm and van de Kerkhof a simulated annealling algorithm. However, research has also been done on optimizing Bézier curves using force directed methods to draw metro maps [5] [9]. One of the earlier force directed methods is the one by Eades [4], which uses a vertex repulsive force and a vertex neighbour attractive force to achieve its aesthetic results. More specific aesthetics have been achieved using circular arcs with a force directed method to create Lombardi graphs [2]. Finding specific graph properties and their relation to graph comprehension and aesthetics has been studied [11] [12]. For example, the metro maps drawn using smooth Bézier curves were found to be easier to understand than their more blocky and traditional counterparts. Other work has been done on extending force-directed methods to preserve edge crossing properties of the graph by Bertault introducing the `PrEd` algorithm [1]. This algorithm was later improved upon resulting in the `ImPrEd` algorithm [14].

2.1 Force-directed graph layout

Drawing graphs on a screen or a piece of paper has been done for many years and used for many different applications. Example applications are visualizing the control flow of a computer program, displaying dependencies, drawing evolutionary trees, or drawing a social network to see who is friends with who. It saves people a lot of time if the visualization of all this data could be automated, especially if the data is updated frequently. Luckily, this is possible and all of these graphs can be layed out by using a force-directed method, although many other methods exist as well [13], like those found in the *Handbook of Graph Drawing and Visualization* [15]. A popular application for drawing graphs from a textual description is *Graphviz* and uses a force-directed method for one of its layout engines `fdp` [6], as well as `neato` which uses a method strongly inspired by force-directed methods [8].

Force-directed methods come in many flavours, but the idea for all of them is the same. The nodes of the graph are layed out in some initial configuration and then have their positions iteratively updated until a termination condition is met. In what direction and by how much a node is moved, depends on the force on the node. This force is the sum of multiple forces trying to improve the position of the node in different ways. There is practically always a repulsive force to prevent nodes from overlapping and an attractive force between nodes that are connected by an edge, since they are in some way related and should probably be closer together. Other forces can be added to change how the nodes are layed out to achieve a different look.

2.2 Nonograms

The regularly known nonogram is a picture logic puzzle played on a grid, and was created by Non Ishida who originally published three picture grid puzzles in Japan in 1988 calling them “Window Art Puzzles” [10]. A year later, in 1989, Ishida introduced the puzzles to James Delgaty, who expressed interest in commercialising them. In 1990, Delgaty coined the name Nonogram, a portmanteau of Non Ishida’s first name and the word diagram. He also convinced the British newspaper The Telegraph to publish the puzzles on a weekly basis in the Sunday edition. These

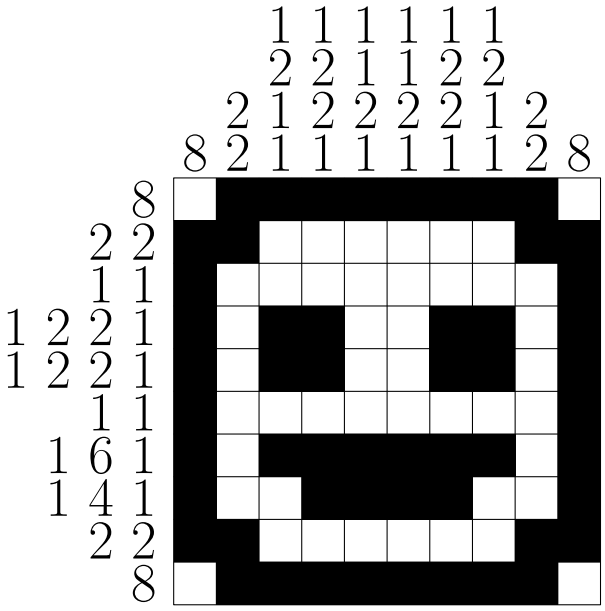


Figure 2: Solved nonogram, where the solution depicts a smiley face.

days it is known under many different names, such as Picross, Pic-a-Pix, Hanjie, Griddlers, Paint by Numbers, Japanese Puzzles, Logik-Puzzles, CrossPix and several other names [10] [18].

The goal of the puzzle is to fill the correct cells or pixels on the grid, revealing the hidden image. Which cells should be filled is hinted at by descriptions on the sides of the grid. Next to each column and row a series of numbers is present, denoting the number of adjacent cells that should be filled. For example, $(1, 2)$ next to a row means that the row contains a group of one filled cell and then a group of two filled cells. Each group must be separated by one or more empty cells and can start at any offset from the border. By combining the hints of the rows and columns the player figures out which cells to fill. Often, the filled cells form an image as a surprising reward for correctly solving the puzzle. Figure 2 is an example of a solved nonogram, where the solution is a smiley.

2.3 Curved nonograms

A variation on the nonogram is the curved nonogram introduced by de Jong [3], where the cells are no longer constrained to a grid but formed by a set of intersecting curves each ending on an enclosing border. Solving the puzzle still involves filling in the correct cells, but the hints are slightly different from the regular nonogram. Each curve will have two boxes with numbers associated with it. These two lists of numbers are a description for each side of the curve. The meaning of the numbers in the box is similar to the hints of the regular nonogram: the number of consecutive cells to be filled and lying along the same curve. Cells are considered consecutive if they are neighbours and lie along the same side of the same curve. Note that the same cell can be both the left and the right neighbour of a cell when tracing along a curve. For example, the cell labeled D in Figure 3 has cell C as both its left and right neighbour from the perspective of the curve that starts on the left. This can have an interesting effect on the solving process. The sequence of cells encountered by tracing curve z along the side labeled with a 4 is A, B, C, D, C, E. We know that 4 consecutive cells in this sequence must be filled. If we try each possibility of

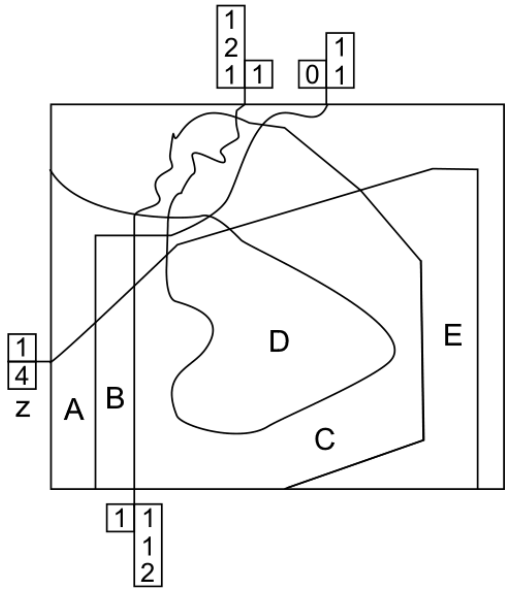


Figure 3: Tracing the curve z that starts on the left of the puzzle along the side that is labeled with a 4, the sequence of cells encountered is A, B, C, D, C, E.

filling 4 consecutive cells we find that the cells at position 3 (C) and 4 (D) are always filled, so we can safely fill these in. However, the cell in position 5 is also C, so by filling the cell at position 3 we also fill position 5! This means we can even eliminate the possibility of filling A, because we only need to fill one more cell and that cell must be next to an already filled cell, which A is not.

3 Preliminaries

First, we explain notations used in the force definitions of Section 4.1. Vectors will be presented in bold, where as vertices/points will not. To denote a vector that represents the displacement from vertex v to vertex w the notation $\overrightarrow{vw}$ is used (i.e., $\overrightarrow{vw} = w - v$). Vectors with a hat $\hat{\boldsymbol{d}}$ above them denote the normalised vector of $\boldsymbol{d}$ (e.g., $\hat{\boldsymbol{d}} = \frac{\boldsymbol{d}}{|\boldsymbol{d}|}$).

Any variables of the form k_i in the force equations are user defined scalar parameters.

The forces are defined in the form of a scalar denoting the magnitude of the force and a unit vector denoting the direction of the force, with the intent of separating the strength from the direction of the force. Also, we use a function $\text{clamp}(x, a, b) = \max(a, \min(x, b))$ to clamp a value x to the range $[a, b]$.

3.1 Curved nonogram

We treat the curved nonograms as planar graphs, meaning a graph that can be drawn in the plane in such a way that its edges only intersect at their endpoints. We model the edges as polylines. A polyline is defined by n vertices $v_0, ..., v_{n-1}$, that describe $n-1$ line segments where line segment $l_i = \{v_i, v_{i+1}\}$. A vertex $v \in \mathbb{R}^2$ is a 2-dimensional point representing a position. Some of the edges are considerd *fixed* and are not allowed to change their position. All edges

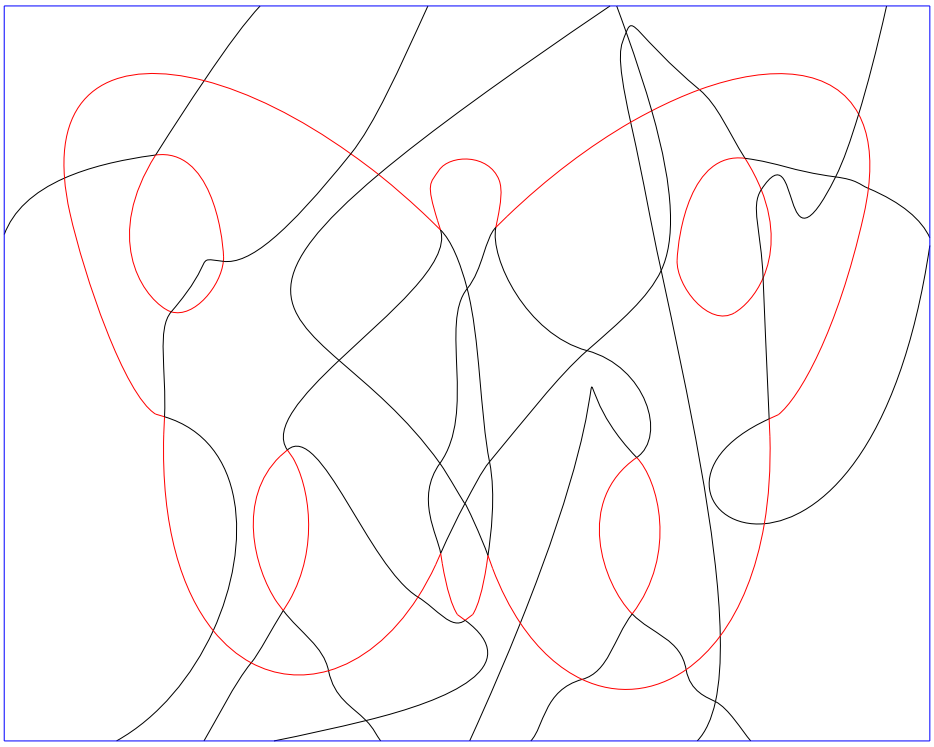


Figure 4: Generated puzzle with the input image of a butterfly shown in red. The black curves are added by a program.

representing the border are fixed. All the polylines that are not fixed are considered *moveable* and can change their position. Each edge is either completely fixed or completely moveable, never partially fixed. For example, in Figure 4 only the black curves are moveable. We fix the polylines that make up the original image, because we do not want to disturb the solution image. We define a *puzzle curve* as a sequence of edges that starts and ends on the border, informally a curve that you would trace when solving a curved nonogram.

3.2 Transitions

In some places in the input curved nonogram a puzzle curve transitions from being fixed to moveable. We call the vertices where this happens *transitions* or transition vertices, some examples can be seen in Figure 4 at the head of the butterfly. These transition vertices are fixed. Transitions are an edge endpoint v, which is the endpoint of two adjacent edges, of which one is fixed, of a puzzle curve q and the endpoint of two adjacent edges, of which one is fixed, of a puzzle curve p. An example of such a situation is shown in Figure 5.

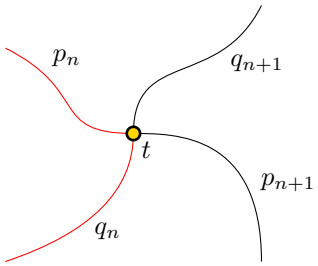


Figure 5: Puzzle curve p and q intersect, resulting in a transition vertex t.

3.3 Curve constrained vertices

A curve constrained vertex is an edge endpoint, which is the endpoint of two adjacent fixed edges on a puzzle curve and the endpoint of two adjacent moveable edges of another puzzle curve. More intuitively, it is an intersection between a fixed curve part and a moveable curve part (a red curve and black curve in Figure 4). These vertices are allowed to move, but only along the fixed edges. We also call these vertices *trains* and the polylines on which they move *tracks*, because they move along a fixed track just like a train. In Figure 4, there are many such intersections along the top of wings of the butterfly. At every point where a black curve intersects with the wing a train is created. The fixed border is also considered a track and each vertex that lies on the border can slide along it.

4 Algorithm

The nonogram graphs are updated using a force-directed method. This means that the graph layout gets optimized iteratively by computing a force for each vertex in the graph and moving it according to this force. The force on a vertex is a sum of multiple forces that each try to optimize a different aspect of our layout and are further explained in Section 4.1.

A global temperature parameter scales how much each vertex moves at an update. Finding a good temperature is important, but tricky and depends on the values of other force parameters and the graph layout. Generally, you want the temperature to be as high as possible because

this allows the vertices to take larger steps and converge faster to a good position. However, if the temperature is too high the simulation can become unstable and introduces oscillatory movement as vertices continually overshoot their prefered positions.

Two mechanism are added to this force-directed method to aid the forces in finding a better solution: curve lengthening/shortening and curve constrained points, each of which are explained in their own subsections.

Additionally, we are not allowed to change the topology of the puzzle, because the puzzle already has a solution and corresponding hint descriptions. If we would change the topology of the puzzle this could change the solution of the puzzle and the solution would no longer reveal the intended image. To guarantee the final layout has identical topology to the starting layout, each update is done in such a way that topology is preserved.

4.1 Forces

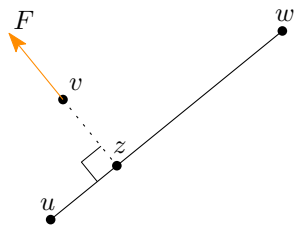


Figure 6: Repulsive force F exerted on vertex v by line segment uw. Vertex z is the closest point to v on the line segment.

We define a *vertex-segment repulsive force* to prevent edges from lying too close to each other. This force is similar to the "node-edge" repulsion of Simonetto et al. [14] [1]. Given a line segment $e = \{u, w\}$ and a vertex v where $v \neq u$ and $v \neq w$, define z as the point on the line segment e closest to v, and define $\boldsymbol{d} = \overrightarrow{zv}$. The repulsive force on v is then defined as:

$$F = k_0 \left|\boldsymbol{d}\right|^{-k_1} \cdot \hat{\boldsymbol{d}} \tag{1}$$

For each vertex, this force is only computed for line segments of incident faces, see Figure 7. This saves computation and usually has little effect on the result, because edges in non-incident faces are going to be further away than edges of incident faces and the force scales inversely with distance.

When applying this repulsive force to trains we ignore the line segments that are part of the track, because if we didn't the neighbouring line segments on the track would be repelling the train inhibiting smooth movement along the track.

We define a *smoothening force* to prevent very sharp corners from forming on the polylines. This force works by considering three consecutive vertices from the same polyline u, v, w at a time. Forces are applied to all three vertices to straighten angle $\theta = \angle uvw$. We always take the inner angle such that $0 \leq \theta \leq \pi$. The direction of the force on u is perpendicular to $\boldsymbol{d}_u = \overrightarrow{vu}$ pointing away from the inner angle and is denoted by $\boldsymbol{x}_u$. The force on u is F_u whose definition is provided below. We want the force to be stronger when there is a very small angle and have the force be zero when the angle is exactly straight. The constant k_1 is to give more control over how the strength of the force should grow depending on angle θ. Setting k_1 to a value between 0 and 1 makes the force grow faster, where the closer to 0 the quicker it gets close to

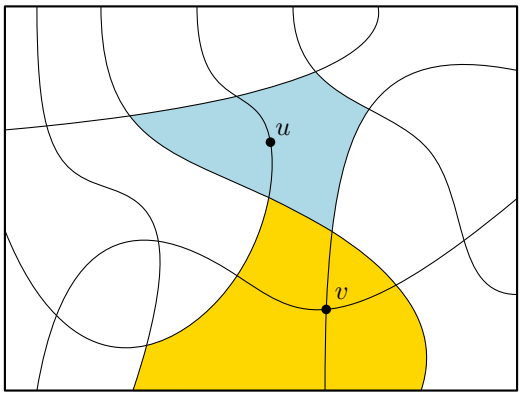


Figure 7: Incident faces of vertex u are highlighted using light blue and incident faces of vertex v are highlighted with yellow. Most vertices have only two incident faces, like u, only vertices at the endpoints of edges have more than two incident faces, like v.

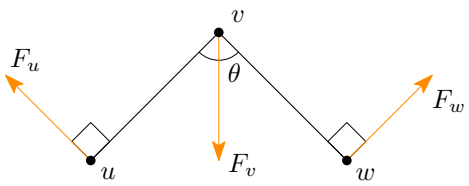


Figure 8: The smoothening forces working on a triplet of vertices u, v and w.

its full potential. At $k_1 = 1$ the force growth is linear in accordance to θ. For $k_1 > 1$ the force is grows slowly at first and only grows very quickly as θ get smaller. The larger k_1, the more this behavior is exaggerated. Intuitively, this results in a force that is mostly absent while angle θ is close to straight and only shoots into action when θ is sufficiently small. The force on w is similarly defined. We scale the force inversely by the length of the segment, because we believe moving larger segments should be costlier.

$$F_u = \frac{k_0(1 - \theta/\pi)^{k_1}}{|\boldsymbol{d}_u|} \cdot \boldsymbol{x}_u \tag{2}$$

The force on the middle vertex v has a different definition. Let $\boldsymbol{b}$ be a bisector of the angle $\theta = \angle uvw$, which will be the direction of the force. The force is then defined as follows:

$$F = k_2(\pi - \theta)^{k_1} \cdot \hat{\boldsymbol{b}} \tag{3}$$

We define a *vertex neighbour attractive force* [4], that attracts neighbouring vertices to each other to add elasticity to the curves. This force has a large influence on how tight or meandering the curves become. Given a vertex v and a neighbouring vertex u define $\boldsymbol{d} = \overrightarrow{vu}$, the attractive force is defined as:

$$F = k_0 \cdot \log \frac{|\boldsymbol{d}|}{k_1} \cdot \hat{\boldsymbol{d}} \tag{4}$$

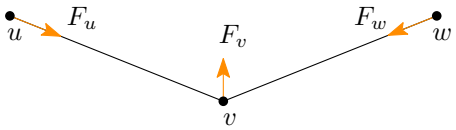


Figure 9: Neighbour attractive forces on the vertices of a polyline. u and w only have v as a neighbour in the polyline and thus are directed at v. Vertex v is attracted to both u and w resulting in an upward directed force.

Finally, a force is added to transition points. We want the puzzle curve to remain smooth at these places and introduce the *transition smoothening force*.

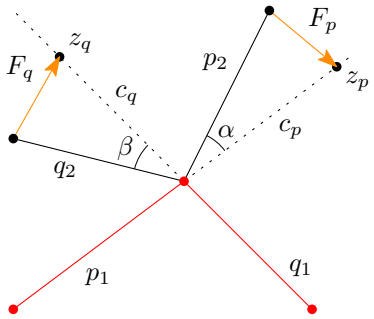


Figure 10: Polylines p and q intersecting and transitioning from being fixed to free. The transition smoothening force tries to keep the polylines smooth.

An example situation is shown in Figure 10, where polylines p and q intersect and both transition from being fixed to non-fixed. In the example, line segment p_1 is fixed and line segment p_2 is free to move. Similarly, only q_2 can move for q. c_p and c_q are lines representing the continuations of p_1 and q_1 respectively. To make sure these transitions stay smooth, we try to minimize the angles α and β. The force definition is provided in the following equation, where $\boldsymbol{d}$ is the direction of the force. The direction of the force for a vertex p is towards a virtual vertex z that lies on the continuation line c_p and is the same distance away from the intersection as p. This way we aim to not change the length of the segment and only rotate the vertex p towards the continuation line, to get a smooth transition.

$$F = k_0 \cdot \alpha^{k_1} \cdot \hat{\boldsymbol{d}} \tag{5}$$

4.2 Train movement

Trains are unable to move directly in the direction of the force applied to it, because their positions are constrained to tracks. At each iteration a train is allowed to move to a different position on the line segment it is currently on. To move to an adjacent line segment on the track, the train first has to travel to the endpoint that is shared by the adjacent line segment. To get the force F' of a train v along a line segment $p = \{u, w\}$ given a force F, we project F on the normalized direction of p to get the magnitude of F'. Let $\boldsymbol{s} = \overrightarrow{uw}$ and $d = \hat{\boldsymbol{s}} \cdot F$, then d is the magnitude of F'. The sign of d denotes the direction of the force along the line segment, so whether it moves forwards or backwards along the track. We store the position of a train v by storing its line segment offset along the track and a scalar $t_v \in 0, 1$ to denote how far along the

line segment it is. The updated value of t_v is $t'_v = clamp(t_v + T \cdot d/\|\overrightarrow{uv}\|, 0, 1)$, where T is the global temperature.

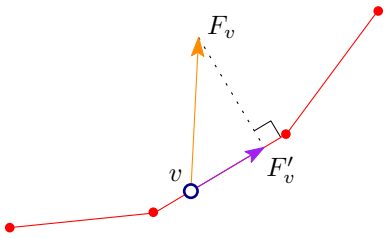


Figure 11: Train v has force F_v exerted on it, but can only move along the red line segments. The force F_v is projected onto the track, resulting in F'_v, to determine the force along the track.

4.3 Curve lengthening and shortening

A mechanism has been added for edges to change in length. This mechanism helps edges achieve a more relaxed state. In the case where the vertices of an edge get squashed together, the edge will form an undesirable zig-zagging pattern. This can be remedied by removing on or more line segments. Forces will move vertices of the curve to better positions, but forces can only stretch out/squash edges up to a certain amount to change their length. In the case that a line segment gets stretched out a lot, we will add an extra line segment. Similarly, in the case where line segments of the same edge get closely pushed together, we remove one or more line segments.

Lengthening an edge is done by adding an intermediate vertex on a line segment between two neighbouring vertices. This vertex is guaranteed to lie on the edge. An intermediate vertex is introduced in the middle of a line segment if the length of that segment has surpassed a certain threshold d_{split}.

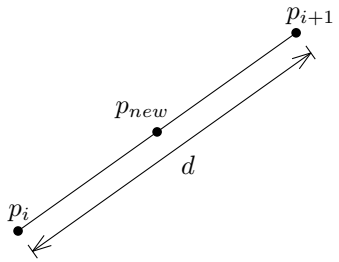


Figure 12: A new vertex p_{new} is added, because the distance d between p_i and p_{i+1} is larger than d_{split}.

Shortening an edge is done by removing one or more vertices and their corresponding line segments from it. Vertices are removed if the distance between two non-neighbouring vertices of the same edge is below a threshold d_{merge}. All the vertices of the edge between those vertices are then removed. However, we must be careful when removing part of an edge, because it can violate our topology preservation constraint. It is only safe to remove part of a polyline if the (potentially non-simple) polygon formed by the edge and new segment is empty. We test for this, thus ensuring our topology is preserved. Figure 13 displays a situation where vertex p_m and p_n are within distance d_{merge} from each other. All the line segments between p_m and p_n can be removed, shortening the edge, as long as area A has no vertices in its interior [14].

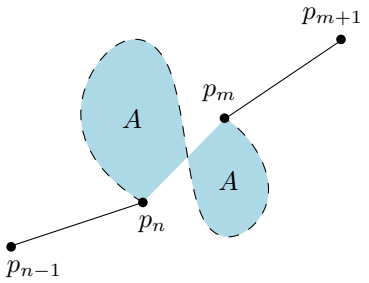


Figure 13: All vertices between p_n and p_m can safely be removed if there are no vertices in area A.

4.4 Topology preservation

Topology preservation is guaranteed by limiting the distance a point is allowed to move at each iteration. The logic of finding this limit follows that of the `PrEd` algorithm [1] [14]. To explain it in short, for each vertex-edge pair in the graph a *borderline* is created that none of the vertices in the vertex-edge pair is allowed to cross. If none of the vertices cross these borderlines, the topology is guaranteed to remain the same.

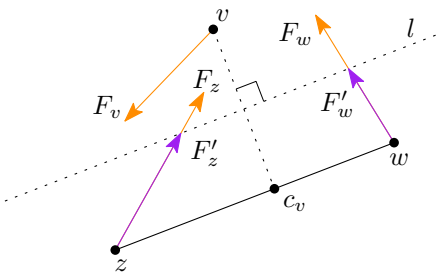


Figure 14: The forces of F_z and F_w are capped to F'_z and F'_w so that they do not cross borderline l between vertex v and edge zw. Vertex c_v is the closest point to v lying on segment zw.

An example situation is sketched out in Figure 14, where the vertex-edge pair is formed by vertex v and edge zw. The borderline l is found by first finding the closest point c_v to v on the edge zw, and then taking the line that goes through the midway point of $\overrightarrow{vc_v}$ and is orthogonal to $\overrightarrow{vc_v}$. If any of the force vectors intersect this borderline, they are rescaled so that intersection is prevented.

`PrEd` and `ImPrEd` both compute the maximally allowed movement for a vertex in all directions. They simplify this computation by quantizing all the directions to eight sectors. When updating the position of a vertex they check in which sector the update direction lies and limit the movement of the vertex to the maximum distance allowed for that sector. In our case we don't need to know the maximally allowed movement in all directions. We only need to know the maximal movement in the direction of the force, since that is the only direction the vertex will be moving in. As a result, we don't save the maximal movement for eight sectors but only in the single direction of the force. Not only does this simplify the calculation, but this also makes the maximally allowed movement calculation more exact. `PrEd` and `ImPrEd` both have to ensure that the sectors do not cross the borderline to guarantee that no vertices that move in the direction of the sector cross a borderline. This means that some vertices have their displacement limited

more than is necessary, since the distance to the border of the sector is always smaller or equal to the distance to the borderline.

For each vertex only the edges of incident faces are considered. We can safely do this, because it is impossible to cross an edge that is not part of an incident face without first crossing an edge of an incident face. So by limiting movement such that no edges of incident faces are crossed none of the other edges in graph are crossed either. At the start of the simulation we can determine what faces are incident to what edges. This is guaranteed to remain the same since we preserve our topology.

The trains have their movement limited separately. Trains move on a polyline from segment to segment, while the `PrEd` algorithm tries to prevent vertices from moving close to segments. This inhibits the movement along the track and thus another method is used. To guarantee moving a train does not change the topology we have to ensure that it does not cross another train that moves along the same edge. For each pair (a, b) of neighbouring trains we calculate a point c on the track that is halfway between train a and b. We limit their movement such that neither train is allowed to move past point c, thus guaranteeing the trains do not cross each other.

5 Experiments and results

Our experiments consist of taking several existing puzzles and interactively improving them using the tool. For each puzzle the original is shown alongside the end result. Additionally, based on relevant work in graph drawing we define quality metrics to get objective scores for the puzzles which we use to compare the initial puzzle to the result. Also, we show various results for the same puzzle produced with different parameter settings to display their effects.

5.1 Metrics

To evaluate our results we define metrics in two categories: *ambiguity metrics* and *aesthetics metrics*. The ambiguity metrics measure the ambiguity of the puzzle or how easy it is to misinterpret the puzzle. Each ambiguity metric counts the number of ambiguous situations in the puzzle, so we want our sum to be zero: no ambiguous situations. The aesthetic metrics score certain properties of the graphs which influence the aesthetics or look of the graph.

We define two metrics to detect ambiguous situations in the puzzles. One metric to detect intersection angles that are too shallow to reliably make out if two curves intersected or only ran very close to each other, and a metric to detect curves that lie too close together, which can cause curves to look like they are intersecting at a shallow angle when they aren't.

For the *ambiguous intersection angle metric* we test for each intersection angle if it falls below a predefined threshold α_{min}. We set $\alpha_{min} = \pi/20$ radians or $9°$ for all our experiments. The total cost of this metric is the amount of ambiguous angles in the puzzle. We don't want any ambiguous angles and thus a cost of zero is ideal. Let A be the set of all intersection angles in the puzzle, we then define the metric as follows.

$$\kappa = \sum_{\alpha \in A} \begin{cases} 1 & \text{if } \alpha < \alpha_{min} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

We have defined an *ambiguous curve spacing metric* that finds situations where curves are too close to each other. This metric is computed by iterating over each face. For each face we have a closed polyline representing the border of the face. For each vertex p in the polyline, we

exclude part of the polyline that lies within a threshold distance d_t along the polyline. For the remainder of the line segments that make up the polyline, we test if any are within a distance d_{min} to p. If there are such line segments, then we deem the line segment to be ambiguously close to the vertex. In our experiments we set $d_t = 3 \cdot d_{min}$.

Additionally, we define three metrics to capture aesthetic properties of the puzzles: one to measure the smoothness of all the curves in the puzzle, one to measure how orthogonal all the intersection angles are, and one to measure how uniform the face sizes are.

The *aesthetic curve smoothness metric* is to measure how smooth curves are, as we deem curves with very sharp bends to be aesthetically unpleasing. In the definition of this metric we make use of the Menger curvature [17]. The Menger curvature is defined for a triplet of points and returns the reciprocal of the radius of the circle that passes through those points. We use $c(x, y, z)$ to denote the Menger curvature function. We iterate over all four sequential vertices x, y, z and w in every puzzle curve and sum the squared difference in Menger curvatures for each such group of vertices. In our definition P is the set of all puzzle curves modeled as polylines.

$$\kappa = \sum_{C \in P} \sum_{x,y,z,w \in C} (c(x, y, z) - c(y, z, w))^2 \tag{7}$$

The *aesthetic intersection angle metric* provides a measure for how orthogonal the intersection angles are, because Huang et al. found that angles close to 90° (or $\pi/2$ radians) result in easier to trace paths [7]. We also find that orthogonal or near orthogonal angles tend to look best. We look at how much each angle deviates from our ideal angle and sum the squared differences to define our metric, which is also given in Equation 8.

$$\kappa = \sum_{\alpha \in A} (\alpha - \pi/2)^2 \tag{8}$$

The *aesthetic face size metric* measures how close to the average face size each face is, because we thought it would look aesthetically pleasing if each face was approximately the same size. The cost of the metric is the sum of all the squared differences in face size compared to the average face size. In our definition, m is the number of faces in the puzzle and $A_{avg} = \frac{1}{m} \sum_i^m A_i$.

$$\kappa = \sum_{i}^{m} (A_{avg} - A_i)^2 \tag{9}$$

5.2 Experimental setup

All the curves in a puzzle are first converted to polylines with segments of length 10. Vertices that are part of the border or the original image are fixed and are not allowed to move. At the start of the simulation the parameters have the values as shown in Table 1. For each experiment we will explain what parameters were tweaked while using the tool and in what way.

An important parameter is the *temperature*, which is mainly used for keeping the simulation stable. If the force on a vertex at a certain time is too high it can lead to oscillating behaviour, which prevents proper convergence. To prevent this you can scale down the offending force, but this would relatively scale up all the other forces. Alternatively, you could lower the global temperature. This keeps all the force magnitudes the same relative to each other and stabilises the simulation, but at the cost of slowing down convergence [15].

The weight of the edge attraction force (k_0) also has a large impact on the layout of the puzzles, where a high value leads to very straight and tight curves, and a low value leads to very loose and meandering curves (as seen in Figure 18.b).

Description	Parameter	Value
Vertex-edge repulsion	k_0	40
	k_1	3.0
Curve smoothening	k_0	7.0
	k_1	2.5
	k_2	1.5
Edge attraction	k_0	1.8
	k_1	2.2
Transition smoothening	k_0	9.0
	k_1	0.9
Temperature	t	1.0
Edge split threshold	d_{split}	50
Edge merge threshold	d_{merge}	15
Min. edge distance	d_{min}	2.0
Min. intersection angle	α_{min}	$\pi/20$ rad

Table 1: Initial program parameters.

5.3 Results

We show three example puzzles before and after improving them with our tool; a butterfly, an elephant and a beaver. For each puzzle their metric scores are provided in Table 2.

	Original				Tuned			
Puzzle	a_{smooth}	a_{angle}	a_{faces}	?	a_{smooth}	a_{angle}	a_{faces}	?
Butterfly	0.326	121	**1.30**	Yes	**0.233**	**59.0**	1.40	**No**
Beaver	0.353	83.1	2.58	Yes	**0.194**	**36.9**	**2.54**	**No**
Elephant	0.509	133	1.42	Yes	**0.396**	**81.9**	**1.04**	**No**

Table 2: Metric values for puzzles. ? denotes if the puzzle is ambiguous. a_{faces} scores are all a factor 10^{10} larger than presented in the table.

The butterfly puzzle's before and after can be seen in Figure 15. This result was achieved by first increasing the edge attraction force to pull curves tighter. This causes the lower curve on the right border to shrink in length. Afterwards the weight was reduced to about 1.1 to give the curves some slack and relax. All the metric scores have improved for the butterfly (except for the face size metric) and the original is even ambiguous. The worse score for the face metric seems valid, as the resulting layout shrunk some already small faces. We will discuss this in more detail in the Discussion. The curves are smoother where some curves in the original have sharp peaks. Angles are closer to being 90°, especially the angles between the curves and the border. In general, the result looks more organic and the distance between lines is generally more uniform.

The process of changing the beaver puzzle is similar to that of the butterfly and can be seen in Figure 16. All metrics have improved and curves look a lot smoother. Obfuscation of the beaver is not much improved in our opinion, although obfuscation with fewer curves seems more difficult.

The resulting elephant puzzle is an improvement in all metrics and also manages to obfuscate the elephant image better. Again, the curves are smoother and in cases where the curves ran close to each other in the original puzzle there is now more space. For this puzzle we lowered the weight of the edge attraction more than we did for the beaver and butterfly to 0.75. This results in wavier lines. Additionally, parameters can be tuned to influence the result's aesthetics, an example can be seen in Figure 18.

6 Discussion

The goal of the thesis was to develop a program that could improve the layout of automatically generated curved nonogram puzzles by making the puzzle better looking and less ambiguous. For both these criteria we defined metrics to measure these qualities and found that we improved the puzzles in these metric measurements. The positive results of the metrics reflect our own findings. The puzzles look more natural, curves no longer have sharp corners, in most cases the curve spacing is more equal, and intersection angles are straighter. We do produce some undesirable small faces: top right in the beaver puzzle, and bottom left in the butterfly puzzle. The formation of these small faces can be prevented by increasing the weight of the vertex-segment repulsion force, but this also influences other parts of the puzzle layout in potentially undesired ways.

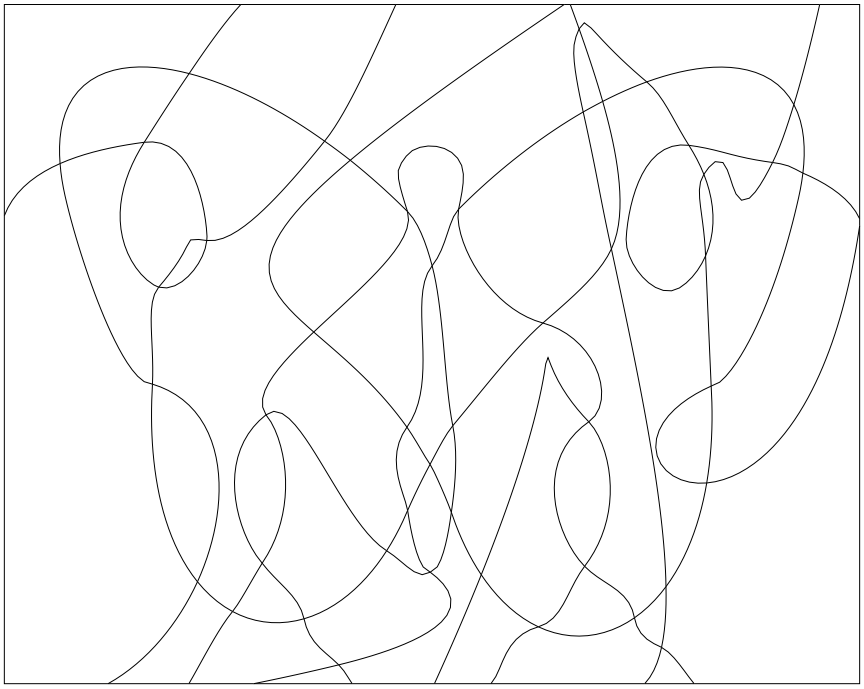
Also, the forces often don't manage to make large faces smaller, which we think would look better. This can be seen in the lower left and right of the butterfly, and the left of the beaver. A new force that specifically tries to shrink the large faces could be added to tackle this problem. This force could also be responsible for making the small faces bigger.

At the moment, the forces seem good for local improvements, like smoothening a sharp bend, but are less effective in more global improvements like equalizing the face sizes. Adding more user interactivity to the tool would help with creating better looking puzzles. Allowing users to drag curves would improve the layout of the puzzles as the user can drag the curves in a way that can make the large faces smaller or obfuscate the solution better.

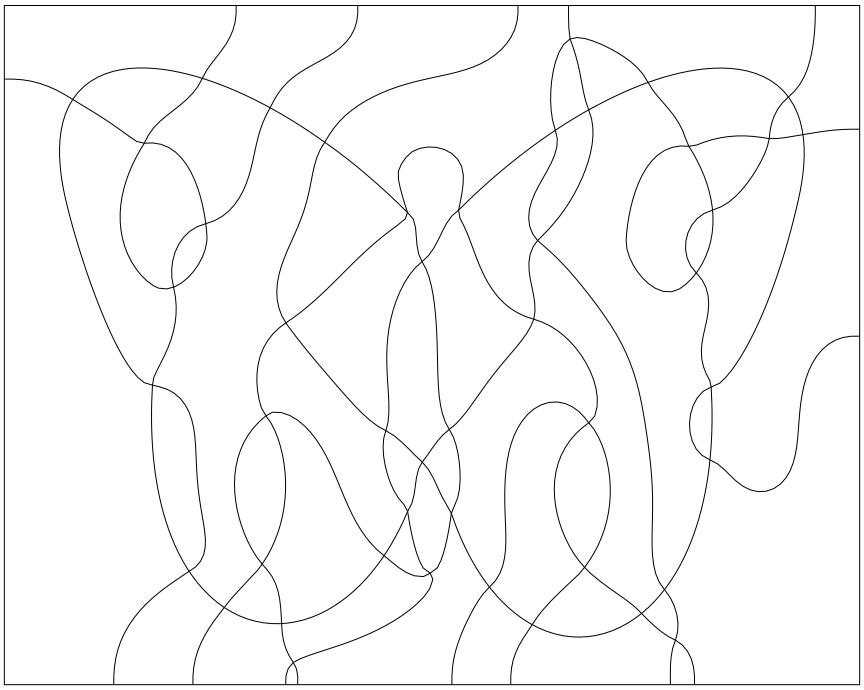
Since we are using polylines to model our curves, the curves can look blocky in places where it bends more tightly. This is most evident in Figure 18.b. This is undesirable as we want our curves to look smooth. A possible solution to this problem is to initialize the puzzle polylines with shorter edges and tune the forces in such a way that they also prefer shorter edges. The downside to this approach is that it uses more vertices increasing the computational cost of each iteration.

In Figure 18 some very sharp bends can be seen at the bottom of the almond shapes on the left and right wing of the butterfly. Here the transition smoothening force attempts to keep the curve going in the same direction as the curves that make up the almond shape. In those areas strong repulsive forces are also present that push the curve into a not so smooth position. The transition smoothening force manages to keep the vertices on which it acts in a decent position, but does not influence the vertices further along the curve resulting in a sharp bend. To mitigate this problem we could change the transition smoothening force to act on multiple sequential vertices in a curve. To avoid pushing the problem of these sharp bends appearing further down the curve, the weight of the transition smoothening force would have a gradual fall-off instead of stopping abruptly.

The vertex-segment repulsive force only considers segments from incident faces. Unfortunately, this approach is not very robust. Let's say all the incident faces of a vertex v are very small and their adjacent faces are also very small, then those faces are close to vertex v. The edges of those faces are ignored with our current implementation, even though they are close to

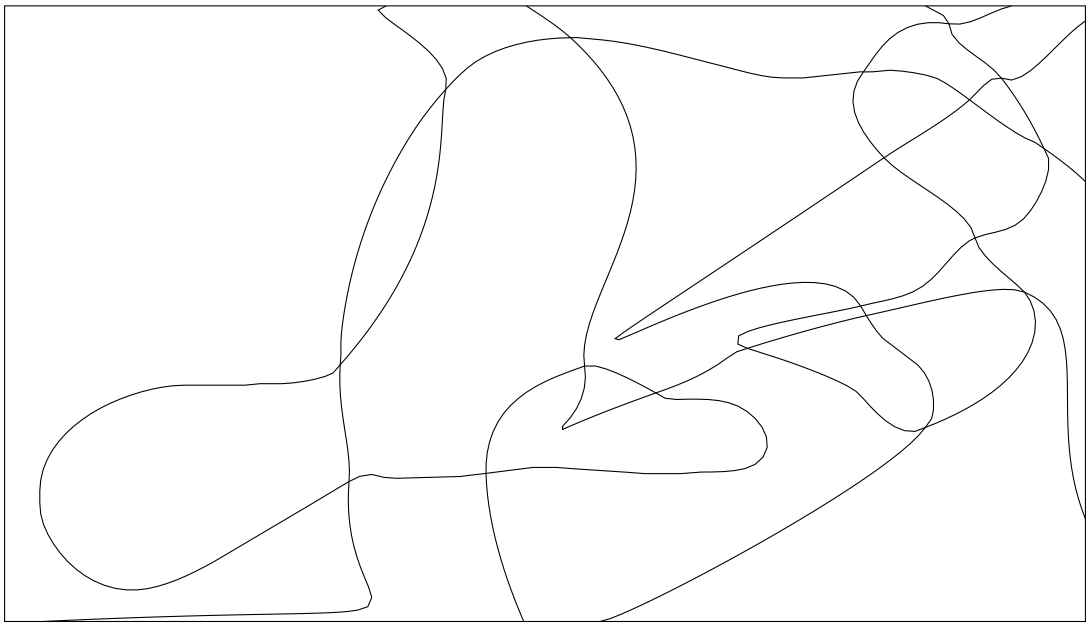


(a) Input (above)

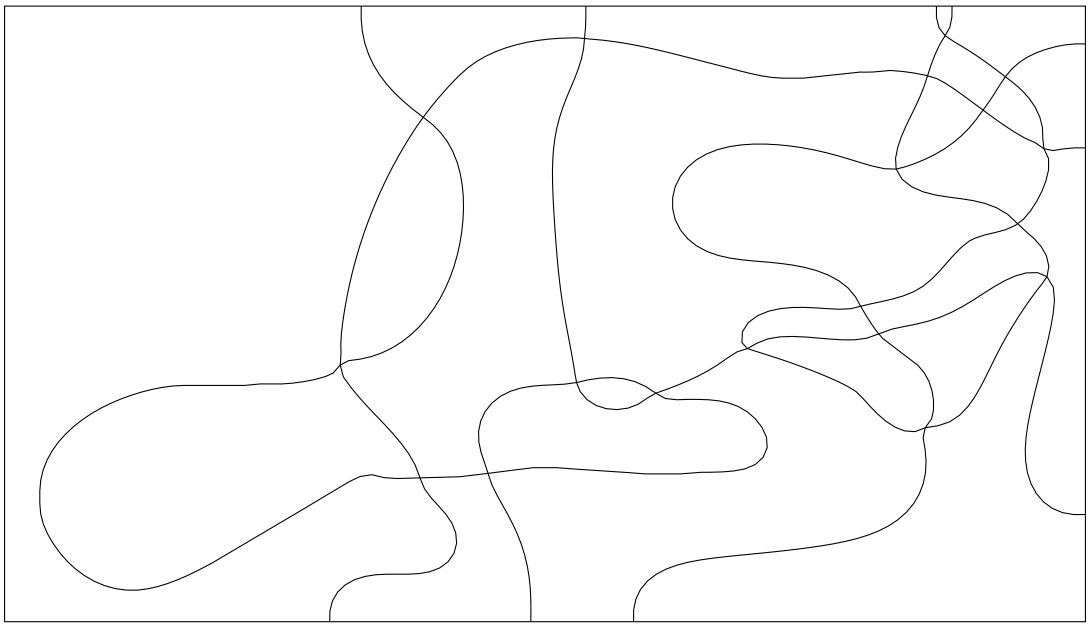


(b) Output

Figure 15: A puzzle of a butterfly, before and after modifying layout.

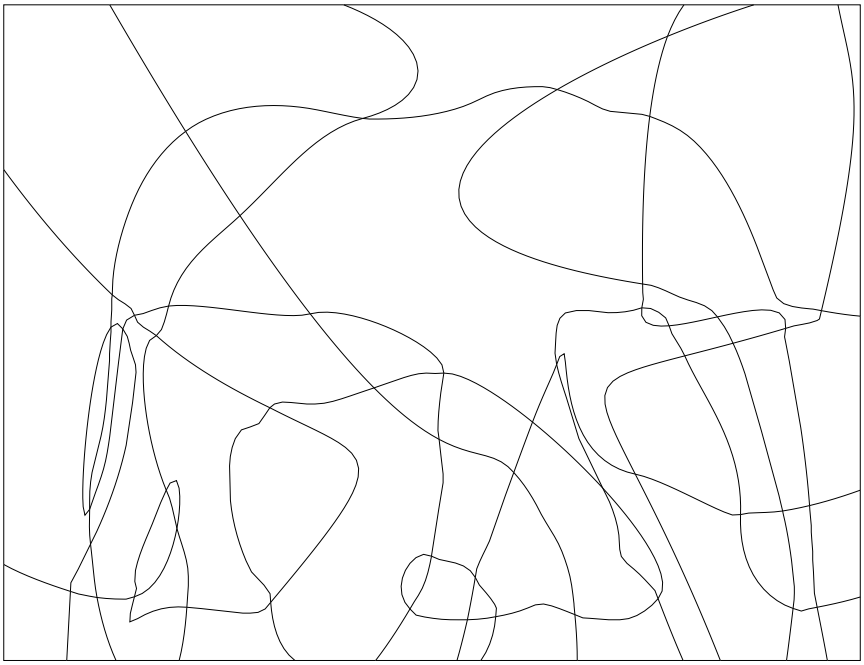


(a) Input (above)

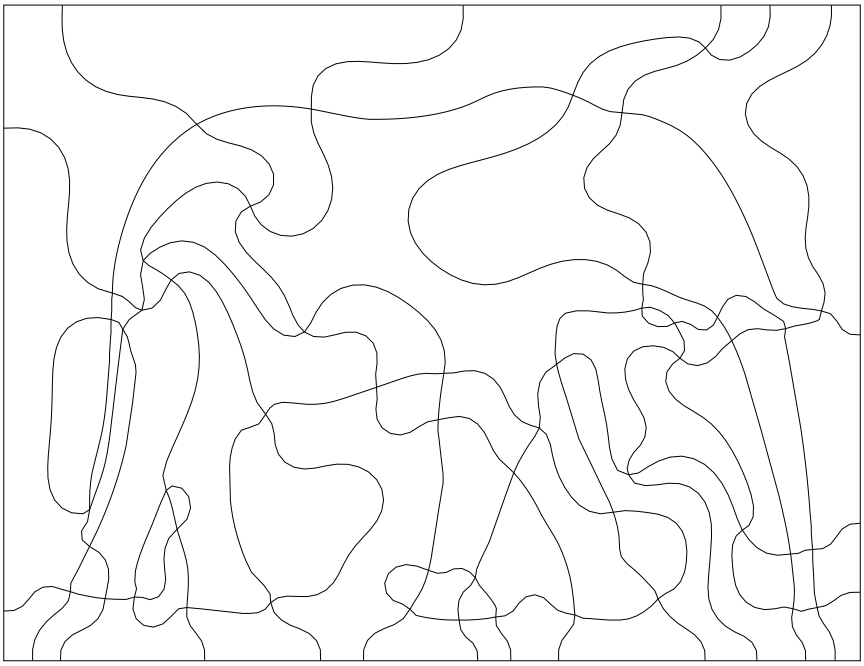


(b) Output

Figure 16: A puzzle of a beaver, before and after modifying layout.

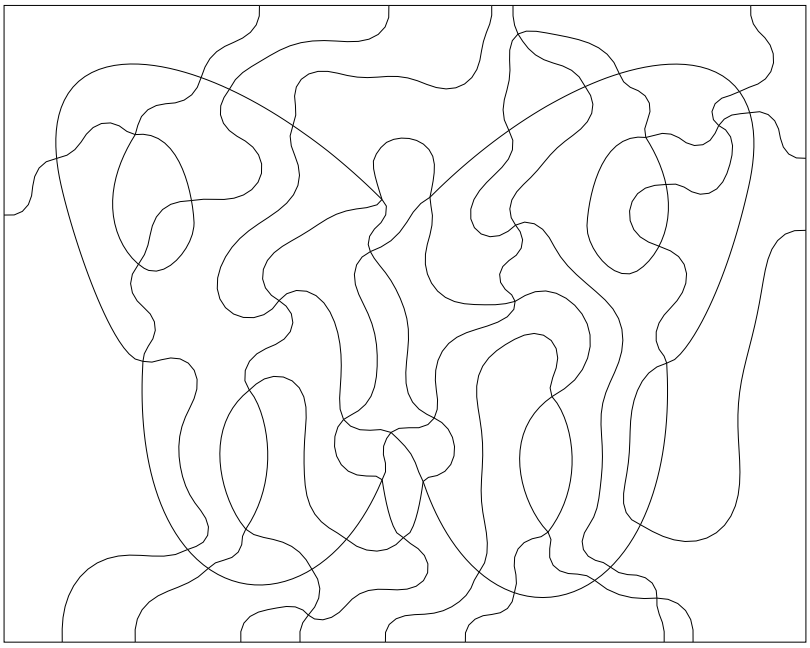


(a) Input (above)

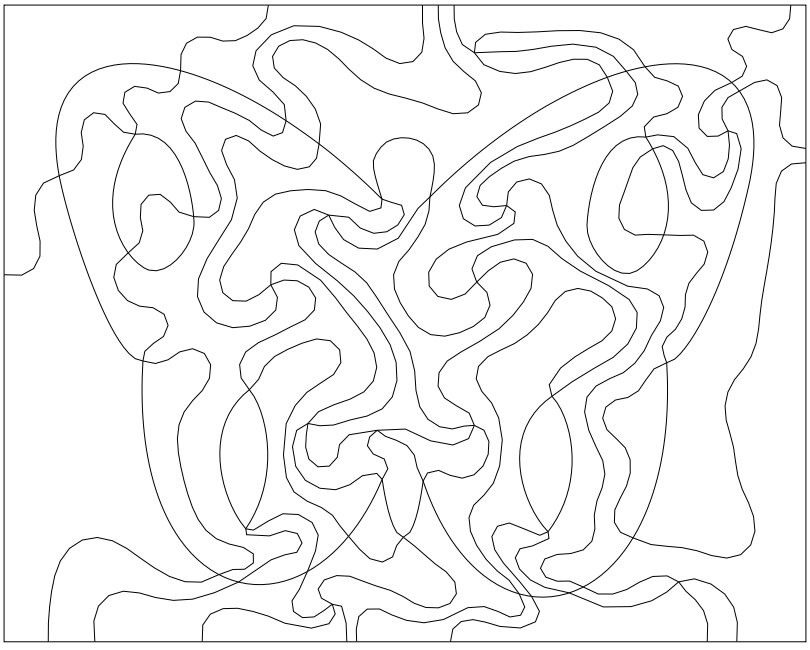


(b) Output

Figure 17: A puzzle of a elephant, before and after modifying layout.



(a) Butterfly puzzle created with higher repulsive force (vertex-edge repulsion $k_0 = 151$).



(b) Butterfly puzzle created with higher repulsive force and lower neighbour attraction (vertex-edge repulsion $k_0 = 102$, vertex neighbour attraction $k_0 = 0.45$).

Figure 18: Butterflies

v and provide a non-negligible contribution to the force on v. We initially thought this would be good enough, and that we could change it later if it worked poorly. A more accurate approach is to select candidate edges that lie within a certain distance, and accelerate these queries using a quadtree for example. This proximity based approach has been used successfully by others [1] [14]. However, due to time constraints this change has not been implemented. We are unsure of the impact of this change on the puzzle quality, but it would seem more reasonable to use instead of the incident faces approximation being used right now.

7 Conclusion

In this thesis we present a tool to improve the layout curved nonogram puzzles using a force-directed approach. This work builds upon the work of de Jong and van de Kerkhof as their programs generate these curved nonograms from input images, where we work with the generated puzzles. Our tool works in real-time allowing users to tweak parameters at any time. This instant feedback gives users an intuition in how the parameter values influence the curves, which helps in finding good values for these parameters. It also becomes easy to see if any parameters have bad values as it will result in erratic behaviour, which is easily recognized. To ensure the puzzle remains solvable and it's solution does not change we preserve the topology of the puzzle during the updating process. We defined several quality metrics to compare if and by how much our resulting puzzles have improved over the originals. Using these quality metrics we find that our puzzles score better in almost all cases, meaning they are unambiguous and more aesthetically pleasing. We find that the resulting images look more organic than the originals and less ambiguous.

8 Future work

The tool can be improved in terms of interactivity by allowing a user to drag around vertices. This way a user can improve the overall layout of the curves by dragging them to their final destination and letting the forces smooth out the kinks introduced by this dragging. Also, small faces tend to stay small, even though it would often look better if they were bigger, but the currently defined forces struggle to enlarge small faces. A face inflation tool can be added to allow the user to manually select faces that can use enlargement.

The maximally allowed movement of vertices can be improved in certain cases. Taking Figure 14 as example, we assume that vertex v, w and z can all move. However, in our puzzles some vertices and edges are fixed. This fact allows us to move the restricting borderline in cases where any of v, w or z is fixed, allowing for more potential movement of the other vertices. This constraint rarely restricts vertex movement, however this is a strict improvement nevertheless.

The curves are modelled as polyline which causes them to look pointy instead of smooth. To mitigate this effect a postprocessing step can be added to smooth out the curves.

Hard constraints can be added to ensure that output is unambiguous or has other potentially desirable properties. For example, not allowing vertices to move fully if that makes an intersection angle fall below the minimum.

References

[1] François Bertault. A force-directed algorithm that preserves edge crossing properties. In *International Symposium on Graph Drawing*, pages 351–358. Springer, 1999.

[2] Roman Chernobelskiy, Kathryn I Cunningham, Michael T Goodrich, Stephen G Kobourov, and Lowell Trott. Force-directed lombardi-style graph drawing. In *International Symposium on Graph Drawing*, pages 320–331. Springer, 2011.

[3] TK de Jong. The concept and automatic generation of the curved nonogram puzzle. Master's thesis, 2016.

[4] Peter Eades. A heuristic for graph drawing. *Congressus numerantium*, 42:149–160, 1984.

[5] Martin Fink, Herman Haverkort, Martin Nöllenburg, Maxwell Roberts, Julian Schuhmann, and Alexander Wolff. Drawing metro maps using bézier curves. In *International Symposium on Graph Drawing*, pages 463–474. Springer, 2012.

[6] Thomas MJ Fruchterman and Edward M Reingold. Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164, 1991.

[7] Weidong Huang, Seok-Hee Hong, and Peter Eades. Effects of crossing angles. In *2008 IEEE Pacific Visualization Symposium*, pages 41–46, 2008.

[8] Tomihisa Kamada, Satoru Kawai, et al. An algorithm for drawing general undirected graphs. *Information processing letters*, 31(1):7–15, 1989.

[9] MD Miermans and HJ Haverkort. A purely force-directed algorithm for drawing curved metro maps.

[10] Puzzle Museum. Griddler puzzles and nonogram puzzles. `https://www.puzzlemuseum.com/griddler/gridhist.htm`, 2017. [Online; Accessed 20-May-2022].

[11] Helen C Purchase. Metrics for graph drawing aesthetics. *Journal of Visual Languages & Computing*, 13(5):501–516, 2002.

[12] Helen C Purchase, Robert F Cohen, and Murray James. Validating graph drawing aesthetics. In *International Symposium on Graph Drawing*, pages 435–446. Springer, 1995.

[13] Schnorr. Visualizing dynamic clustered data using area-proportional maps. Master's thesis, 2020.

[14] Paolo Simonetto, Daniel Archambault, David Auber, and Romain Bourqui. Impred: An improved force-directed algorithm that prevents nodes from crossing edges. In *Computer Graphics Forum*, volume 30, pages 1071–1080. Wiley Online Library, 2011.

[15] Roberto Tamassia. *Handbook of graph drawing and visualization*. CRC press, 2013.

[16] Mees van de Kerkhof, Tim de Jong, Raphael Parment, Maarten Löffler, Amir Vaxman, and Marc van Kreveld. Design and automated generation of japanese picture puzzles. In *Computer Graphics Forum*, volume 38, pages 343–353. Wiley Online Library, 2019.

[17] Wikipedia. Menger curvature. `https://en.wikipedia.org/wiki/Menger_curvature`. [Online; Accessed 20-May-2022].

[18] Wikipedia. Nonogram. `https://en.wikipedia.org/wiki/Nonogram`. [Online; Accessed 20-May-2022].