Connect the dots puzzles with direction indicators

On the generation and evaluation of a new type of connect the dots puzzle.

Master program Game and Media Technology

Faculty of Science

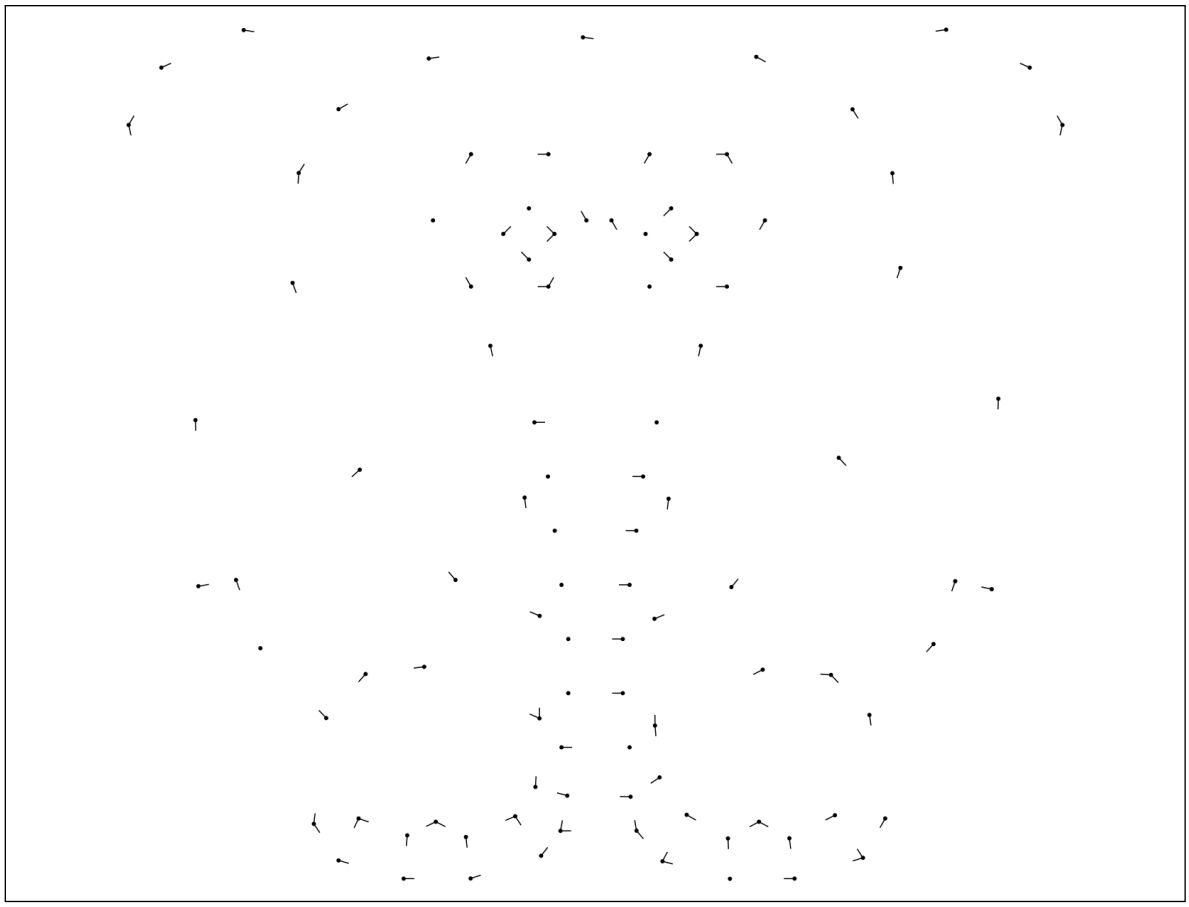
Department of Information and Computing Sciences

Gerwin Klappe (3734676)

June 2, 2014

Utrecht University
Faculty of Science

Figure 1: Example of a generated puzzle



Abstract

Traditionally line puzzles or 'connect the dot' puzzles are solved by connecting numbered dots. These puzzles are solved by connecting points in a specific order indicated by numbers. These puzzles are handmade by placing dots on the outline of a drawing or line drawing. However there are some limitations and problems with this type of puzzle.

For this thesis we researched an alternative puzzle type and propose an algorithm to extract a line puzzle from a given input, such as a line drawing. We researched the different aesthetics in this type of puzzle and a number of visual variations. Also we developed a proof of concept of an environment in which the algorithm can run. Using this proof of concept we processed a dataset with the algorithm and evaluate the resulting data and drawings.

We conclude that there is a viable alternative for numbered 'connect the dot' puzzles that does not use numbered labels. For this alternative it is possible to generate puzzles by an algorithm that produces solvable and recognizable line puzzles with a running time of $O(n^2 \log^3 n)$.

Acknowlegdements

The puzzle mechanics and the algorithm were developed in collaboration with Marc van Kreveld, Maarten Loffler and Frank Staals. The proposed algorithm that achieves $O(n^2 \log^3 n)$ running time was proposed by Marc van Kreveld. The dataset used for the experiment contains SVG drawings traced from reference images, drawings supplied by Mira Kaiser and free available SVG drawings from the internet.

Contents

1 Puzzle design **5**
- 1.1 Traditional line puzzles . . . 5
- 1.2 Puzzle idea and applications . . . 6
- 1.3 Criteria . . . 7
- 1.4 Specific criteria . . . 7
- 1.5 Goals . . . 9

2 Related work and literature **10**
- 2.1 Line simplification . . . 10
- 2.2 Partially drawn graphs . . . 10
- 2.3 Graph orientation . . . 11

3 Algorithm design **12**
- 3.1 Subdivide graph . . . 12
- 3.2 Line simplification . . . 12
- 3.3 Heuristics . . . 13
- 3.4 Orient graph . . . 14

4 Algorithms **17**
- 4.1 Subdivide graph . . . 17
- 4.2 Line simplification . . . 17
- 4.3 Dijkstra, shortest path . . . 20
- 4.4 Evaluation . . . 20

5 Implementation **24**
- 5.1 Data structure . . . 24
- 5.2 Loading and saving data . . . 24
- 5.3 Calculations . . . 25
- 5.4 Algorithm . . . 25
- 5.5 Interface, editor . . . 25

6 Experiment **27**
- 6.1 Setup . . . 27
- 6.2 Puzzle observations . . . 27
- 6.3 Performance . . . 31

7 Conclusion **33**

8 Future work **34**

A Impact of input parameters **36**

B Visual variations **38**

C Data set **40**

D Generated puzzles **41**

1 Puzzle design

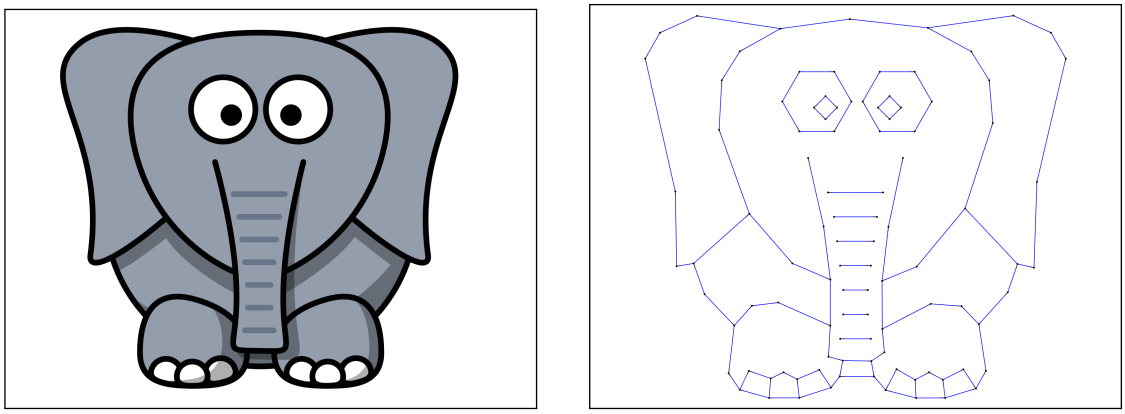
Our puzzle is based on traditional connect the dot puzzles with numbers. We take a look at the specifics of these puzzles and the existing variations. We propose an alternative puzzle and explain the specifics of this alternative puzzle and possible variations.

1.1 Traditional line puzzles

Traditional line puzzles are composed of a sequence of points drawn as dots with numbers. These points are to be connected with line segments according to the sequence as indicated by the numbers. Numbers can be coloured to indicate a different colour for the following stroke. Also multiple sequences can be used to add flexibility by replacing a dot with a symbol to specify the sequence the point belongs to. When the dots are correctly connected a figure can be recognized in the collection of drawn lines. The puzzle mechanics are intuitive and the challenge is to find the next dot that can lie anywhere within the puzzle. Because the dots are numbered a line can easily cross the puzzle from side to side without creating confusion.

However there are a number of disadvantages to this type of puzzle. The sequence must follow a path, similar to an Euler path, however points cannot be visited multiple times. This is often solved by backtracking over the original line and through previous points. To solve the puzzle a person has to have the ability to understand the numbers. There are people that are unable to understand numbers, either they are too young or incapable. The labels for the numbers is unwanted visual clutter that doesn't add anything to the solved puzzle, it only aids to solving the puzzle. Once the puzzle is solved it can distract from the resulting figure.

Figure 2: Input drawing and output puzzle



(a) Input drawing, Openclipart.org

(b) Solved output puzzle

1.2 Puzzle idea and applications

Our new puzzle also consists of a collection of dots that needs to be connected. However the method in connecting the dots is different. Instead of using numbered labels to indicate which dots are to be connected, the points have one or multiple direction vectors. In our case this is represented by a line which links the point to another point in the puzzle. Between these points the player must draw a line.

This approach requires no labels, therefore visual clutter is reduced and a player does not need the ability to understand numbers. Also this approach is not restricted to sequences, and intersection points with 3 or more lines is a possibility. Since a direction is given in which the target point lies, the game of searching for the next point is less challenging. This was part of the fun and frustration of the numbered line puzzles therefore these puzzles could be a bit less challenging. Instead our puzzle gives an estimate direction in which to search for the next point.

1.2.1 Structure

Our puzzle consists of dots with an indication to a dot to which a line has to be drawn. We refer to these dots as points. The indication to where a line has to be drawn is called a link, because it links a point to another. The points to which links can be attached are called the puzzle points. Our final puzzle consists of puzzle points where each puzzle point has a number of associated links. The number of links attached to a point can vary from zero to many depending on the puzzle type.

1.2.2 Visual variations

We explored a number of visual variations, namely symmetric or asymmetric placement of links and the visual representation of the links. In symmetric link placement, as illustrated in figure 3a, two links are placed for each edge, while in asymmetric link placement, illustrated in figure 3b, only one link is placed on either one of the two points of a line segment. Additionally we explored a number of visual representations for the link. The goal was to minimize recognizability of the unsolved puzzle to the drawing and maximize solvability. In this case this means that the link should give as little information about the puzzle as possible while the direction of the link should be as clear as possible.

- **Line**
  This variation is easy to understand. A link is represented by a line that points to the next point and the player has to draw the missing line in between. This line includes some visual information about the solved puzzle, in that sense the links are actually partially drawn edges.

- **Variable line**
  A variation on the line is one in which the length of the line gives an indication of the distance to the associated point. When the distance increases the line length also increases, as illustrated in figure 3d. This way other points can be placed closer together with less chance of causing confusion. However this eliminates another search element of the puzzle and therefore could be less challenging. The length of the lines would have to be larger to allow for the variable length and therefore it could become easier to guess the original drawing from the unsolved puzzle.

- **Point wise**
  This method of link representation, as illustrated in figure 3c, includes the smallest amount of visual information, but is harder to solve without additional tools besides a pen, such as a ruler. A link is represented by a smaller point in the neighbourhood of its associated puzzle point. The line from the puzzle point to the smaller point represents the direction to the area where the next point lies. To further distinguish the smaller point or link from the puzzle point, the smaller point can be presented in a different colour.

We find the line representation to be the most intuitive and most easy to solve without any tools. We prefer the asymmetric variation since this allows for more flexible link placement and reveals less of the solved puzzle. Point wise links reveals even less of the solve puzzle, since the outline of the figure is more difficult to perceive from a collection of points. However we use asymmetric lines for our case due to its intuitive nature and flexibility.

1.3 Criteria

The puzzle must obey a set of general criteria, which defines a puzzle as a good puzzle. From this we extract a set of quantifiable specific criteria which we use as input for our algorithm design. In general our puzzle must obey the following rules: it must be solvable without any tools other than a pen, the unsolved puzzle should not be recognizable as the input drawing, and the solved puzzle must resemble the original line drawing.

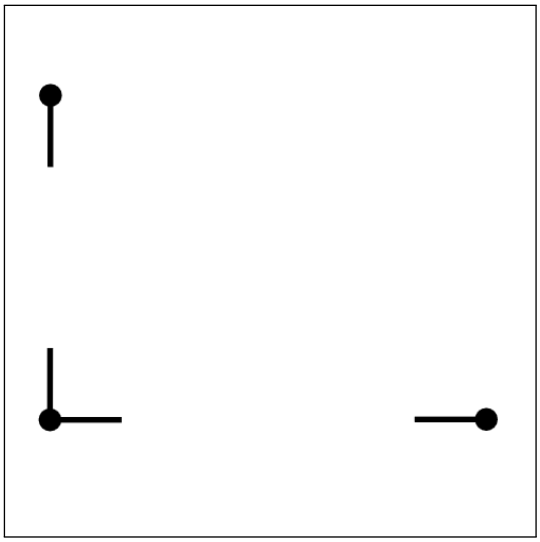
- **Solvability**
  It must be clear which lines have to be drawn between points, i.e. the destination point must be distinguishable from other points in the area of the direction.
- **Recognizability**
  The collection of links and points, i.e. the unsolved line puzzle, must not be clearly recognizable as the solution to the drawing.
- **Resemblance**
  The solved line puzzle must have a recognizable resemblance the original line drawing.

1.4 Specific criteria

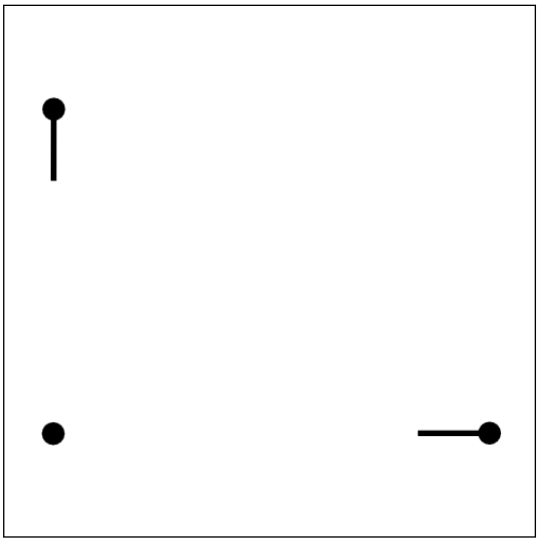
From the general criteria we formulate a number of specific criteria, that we can use as a set of rules to determine the effectiveness of the algorithm.

- **Multiple links on one point must be distinguishable from each other.**
  When multiple links exist on a point the angle should be large enough so that a link does not overlap with other links on the same point and the direction they are oriented in is clearly visible.
- **Two connected points must be at least three link lengths from each other**.
  The distance of two points which we connect with a link should be at least three link lengths apart, including the link itself. Therefore the empty space is at least two times the link length.
- **A point in the line puzzle should be on a line of the line drawing.**
  Each point that we use in the line puzzle must lie on a line in the original line drawing.
- **Lines cannot intersect other lines**
  Our assumption is that our input drawing is planar, and we require our output to be planar as well.
- **Links from different points cannot overlap each other.**
  One link cannot overlap another link in our resulting line puzzle. This criterion is in compliance with the criteria that our output is planar since our input is planar as well.
- **The solved puzzle must resemble the input drawing.**
  The puzzle resembles the input drawing within the allowed maximum error specified in the input.

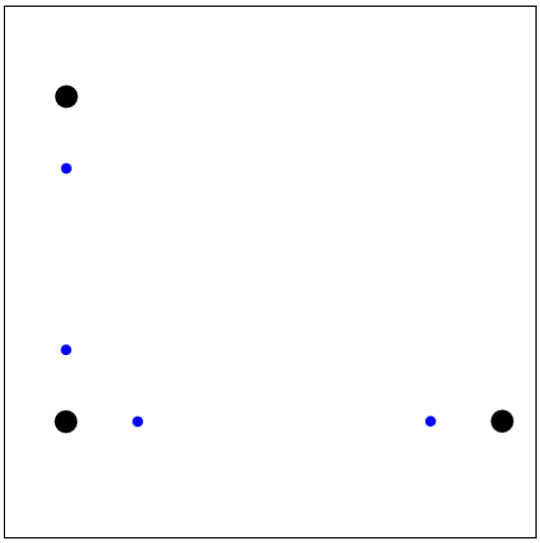
Figure 3: Visual variations of the links



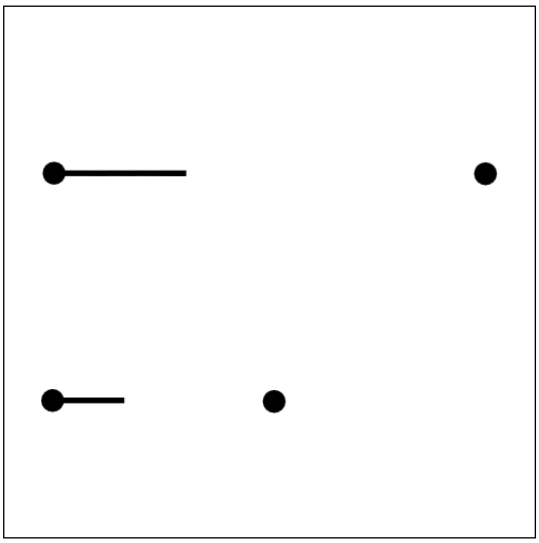
(a) Symmetric



(b) Asymmetric



(c) Pointwise



(d) Variable length

1.5 Goals

Besides our specific criteria we also defined a number of goals for our algorithm, to optimize the visual properties of the puzzle and include the general criteria that cannot be specified in specific criteria.

- **Minimize the number of points in the puzzle**
  We want to include the smallest number of points in the puzzle to minimize recognizability. We aim for a near optimum solution, which is the minimum number of points in the simplification within the maximum error that is valid with respect to our criteria.
- **Minimize the maximum outbound degree of points in the puzzle**
  We can define the optimum solution for this problem without any additional criteria. However the optimal solution may not be valid with respect to our criteria. Therefore we minimize the maximum outbound degree with respect to our criteria.

2 Related work and literature

Our puzzle design requires us to solve a number of problems in different areas of geometric algorithms research. Our first step is to simplify a subdivision while taking the surrounding environment into account. During this process we minimize the recognizability of the graph as unsolved puzzle without losing visual relevance because the simplification process attempts to minimize the number of points in the output. Finally we need to orient an undirected graph to a directed graph. Line simplification and graph orientation problems are well known research topics in algorithms and graph theory, but we also take a look at partially drawn graphs, which is similar to our link representation.

2.1 Line simplification

Line simplification or generalisation is well-known topic of research within computational geometry. There are a number different approaches to this problem, each have been improved since their original proposal. Whether a method is suited depends on the type of application and requirements to the output. A well-known simplification algorithm is Douglas and Peucker [1], which uses a distance measure to add points to the simplification until no points fall outside this distance measure. This is one of the best known and utilized line simplification algorithms and is very straightforward. Further work has been done on this algorithm by Hershberger and Snoeyink [2] to improve running time from $O(n^2)$ to $O(n \log^2 n)$. This algorithm is also known as Ramer-Douglas-Peucker, since a similar algorithm has been suggested in 1972 by Ramer [3].

Another approach to line simplification was proposed by Imai and Iri [4] to achieve an optimum solution with respect to the minimum number of points. The proposed algorithm runs in $O(n^2 \log n)$ but later research by Chan and Chin [5] showed that this could be improved to achieve $O(n^2)$ running time. The versatility of this approach was demonstrated by Berg et al [6]. They propose a line simplification algorithm which is an extension to Chin and Chan's implementation [5] of Imai and Iri's algorithm. This algorithm includes an efficient way to determine any intersections of lines and shifting interior points for subdivision simplification. The proposed algorithm has a $O(n(n+m) \log n)$ running time, where n is the number of points in a polyline subdivision of a graph and m is the number of extra points that cannot cross the polyline in the simplification.

An area based approach was proposed by Visvalingam and Whyattl [7] to define the most significant features of a line. This is in particular useful for mapping and geographic data simplification. Depending on the desired granularity, small enclaves or other geographic features do not need to be visible. With angle and distance based simplification methods these narrow features are still considered significant. This is solved by determining eliminating points based on the area they form with surrounding points. This way the method has a stronger preference for larger global features.

2.2 Partially drawn graphs

Our puzzle is in essence a graph with a different representation for the edges. In the line representation these are actually partially drawn edges. Various studies have been performed on this subject. The goal of these studies is to improve graph navigation in cluttered non-planar graph representations. The problem is that in a non-planar graph with high level of connectivity a lot of visual clutter can occur. These studies aimed to reduce this visual clutter and therefore improve usability. The user studies that were performed focused on task completion and navigation problems. Our problem is related but also inherently different. We also need to perform navigational tasks, however we do not want maximize the usability, there has to be a challenge. Also we want to minimize recognizability of the input drawing in contrast to related studies, which focused on maximizing overall usability and recognizability of the graph.

Burch et al [8] have evaluated partial drawn links in two variations, a straight line and a tapered variation. The link distance that is drawn is relative to its length and expressed in percentages. They use 5 different step sizes and for the tapered variation 100% link length is also used. For the user study they used a task based approach and divided three different types of tasks. Two task consisted of evaluating navigation, e.g. is it possible to go from one point to another, and the third task was to determine the point with the highest outbound degree. It showed that accuracy in the graph navigation tasks tends to suffer from partial drawn links. In the third task the error rate decreased when as the link length was reduced.

Didimo and Patrignani [9] focussed on a method to maximize the fraction of the distance with the edges of a partial edge drawing can be drawn. They focus on a symmetric model, in which two parts of the edge must be drawn of equal length. They propose different solutions for graphs of different planarity.

2.3 Graph orientation

We want to orient our graph to determine the placement of links on the puzzle points in our puzzle. Since we want to minimize the number of links on a single puzzle point, we want to minimize the outgoing edges on a single point in the oriented graph. This is known as a graph orientation problem and has been researched extensively.

Chrobak and Eppstein [10] studied the orientation problem for planar graphs. They give proof for the boundedness of certain orientations. That is, every planar graph has a 3-bounded orientation and a 5-bounded acyclic orientation. Algorithms to calculate these orientations are given which run in linear time. Additionally they address a parallel method that can compute a 3-bounded orientation and a 6-bounded acyclic orientation in $O(\log n \log^* n)$ parallel time with $O(n \log n \log^* n)$ processors.

Asahiro et al [11] studied the problem of orienting a graph with weighted while minimizing the maximum out-degree of the vertices, i.e. the sum of the weights of the outbound edges. They give a proof that this problem is $\mathcal{NP}$-Hard and propose an approximation algorithm with an approximation guarantee of $2 - \epsilon$. However ϵ is highly dependent on the average weight of the edges as it is defined as 1 divided by the average weighted out-degree of the graph.

3 Algorithm design

In this section we analyse the different algorithmic problems we need to tackle to incorporate the criteria in our proposed method. We propose a number of methods and strategies to achieve the criteria and goals as stated in the design.

3.1 Subdivide graph

We need to subdivide the graph to create the subdivisions on which we apply line simplification. Our input is planar so all subdivisions, apart from isolated cycles, are the lines that go from one end point or intersection point to another. We simply take these points and traverse them until we reach another endpoint or intersection. Finally we detect any un-traversed points, these are isolated cycles and are split on an arbitrary point.

3.2 Line simplification

Line simplification is applied to reduce the number of points required for our puzzle. Currently there are a number of line simplification algorithms researched, best known is the Ramer-Douglas-Peucker algorithm. We evaluate a number of existing algorithms to see if any specific characteristics would make it appropriate to our needs. Each of these algorithms employ different strategies to preserve the most significant properties of a line.

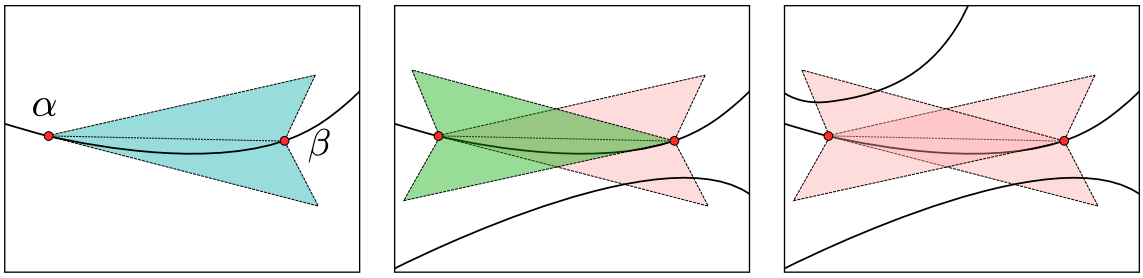
- **Algorithms for the reduction of the number of points required to represent a digitized line or its caricature, Ramer-Douglas-Peucker**
  This is a selection based algorithm [1], which selects furthest points from in between line as most significant and adds these points to the simplification. This is repeated until no points from the input fall outside of the distance measure with respect to the simplification. This is a very straightforward line simplification algorithm but does not always provide the simplification with the lowest possible number of points.

- **Line Generalisation by repeated elimination, Visvalingam and Whyatt**
  Rather than selecting a point that is the furthest away from the line, thereby indicating it would be the most significant feature, this method [7] uses an area based approach. They use the area that is covered by the face that is created by connect two points on the line, like a cut-off from the original line. In this way this method discriminates against small narrow features, and considers them insignificant. This is very practical for map generalization in which depending on the scale, the focus has to shift from narrow features to global features.

- **Computational-Geometric Methods for Polygonal Approximations of a Curve, Imai and Iri**
  This method [4] calculates all possible shortcuts and builds a graph on which the shortest path is calculated, i.e. the minimum number of points. Since it employs Dijkstra's shortest path algorithm to find the simplification with minimum number of points and the possibility to apply different heuristics during graph build-up makes this a very versatile method. This is demonstrated by Berg et al [6] who extended this algorithm to prevent intersections and border crossings from occurring in the simplification.

We want to simplify a line, not a surface, therefore the method proposed by Vislingam and Whyatt would not be the most straightforward approach since their method is surface based. The Ramer-Douglas-Peucker approach is very good in line simplification while keeping the most significant features, however we also want to minimize the number of points and apply different heuristics based on our

Figure 4: Ambiguity check



(a) area check α and β (b) partial valid (c) invalid

criteria during the simplification process. Therefore the method proposed by Imai and Iri is the one we use as a base for our algorithm.

3.3 Heuristics

Based on our criteria we define a number of heuristics that need be incorporated in the shortcut selection of the line simplification algorithm. Often there are multiple allowed simplifications, therefore we include these additional heuristics to steer the simplification process to the solution that would be the most optimal for our case. This could mean that there is a trade-off between number of points and puzzle quality. Sometimes a simplification that incorporates these additional heuristics will have more points than the optimal minimum point simplification.

3.3.1 Solvability

Our puzzle needs to be solvable, which in our case means it must be clear to which point a link indicates the line that needs to be drawn. For this we devised an ambiguity check to determine whether a link can be seen to direct to more than one point. We define an area in which there cannot be any other part of a line except the part that is eliminated by the candidate shortcut. This is illustrated in figure 4a. The area has two parameters, namely α and β, these are the angles with respect to the line of shortcut, in which the area spans. This is done for both directions. When only one direction is invalid the shortcut is still allowed, since we only need one valid direction to orient the edge.

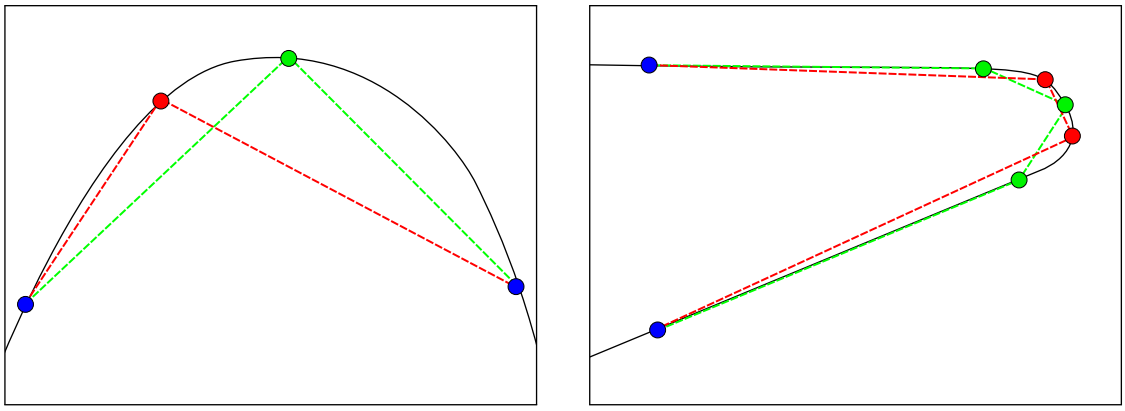
3.3.2 Recognizability

When the puzzle is solved, it must resemble the puzzle as closely as possible. Since there is a possibility that for a single segment there is more than one simplification with an equal number of points, we want to choose the simplification that is in our case the best fit. Since a simplification reduces the length of the line, i.e. curvy lines become straight lines, we choose the simplification with highest length. This metric can be included in the shortest path algorithm in the Imai and Iri approach when the shortest path is computed from the graph of valid shortcuts. Figure 5a illustrates the way this method emphasizes the more significant features. In this simplified example both shortcuts would fall within in the ϵ distance, but it is clear that the longest route is preferred over the shorter route.

3.3.3 Minimum edge length

We specified a minimum edge length in our criteria. If an edge is too small it would not make sense draw a link, because it would either fill up the gap, overshoot or leave a small gap. Instead we use

Figure 5: Additional heuristics



(a) longest simplification

(b) minimum edge length

the original curve and draw this instead of a link. However this increases the recognizability of the unsolved puzzle, so we want to avoid this whenever we can. This is illustrated in figure 5b. We can observe that even though there is a more optimal simplification, there is also a simplification with one point more where we don't have to pre-draw a part of the puzzle. Since we want to avoid this as much as we can this trade-off makes sense.

Therefore we defined a minimum edge length parameter. Shortcuts are not allowed if they do not meet this requirement. However this would mean that the original route is not allowed since the high detail polyline approximation has very short edge lengths. This could result in an unsolvable shortest route problem in the graph of shortcuts, meaning there is no possible route from start to end. Therefore we have a fall-back method that reduces the minimum edge length by 10% of the original value until it reaches 0, in which case all original edges are allowed and therefore the shortest route problem in the graph of shortcuts becomes solvable.

3.4 Orient graph

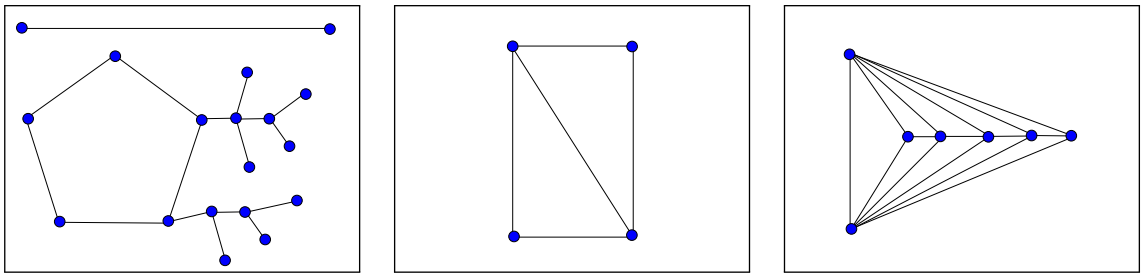
The result of the line simplification process is an undirected graph, we need to direct this graph so all the edges are one way. This determines the link placement in the puzzle. Since we want to minimize the number of links placed on a point we want to achieve to lowest maximum out-degree across the graph. To achieve this we want to minimize the maximum degree of outbound edges on a point in the directed graph. The outbound edges will be used for the placement of the links.

The minimum value we can achieve for the maximum out-degree depends on the structure of the graph. In some cases we can always achieve an out-degree of 1 or 2 and in worst case this is 3. This is without the constraints as formulated in the criteria. Due to these constraints parts of the graph may already be oriented. Furthermore we assume the input graph is connected, if not we treat each disconnected part of the graph as a separate connected graph.

- **Out-degree of one**
  For a graph that is a path we can always achieve a maximum out-degree of 1. This is intuitively determined, since a point only links to two nodes we can easily choose a direction along the path to orient all points in. The last point does not need a link. Similarly trees can be drawn with maximum out-degree 1. We choose a root node arbitrarily and orient all edges towards the root.

  For a graph with a single cycle we can always achieve a maximum out-degree of 1. Our links

Figure 6: Example graphs and their out-degree bounds



(a) Out-degree one graphs (b) Out-degree two graph (c) Out-degree three graph

just indicate a direction in which the cycle 'rotates'. All points have a degree of 1. If there are trees lines attached to this cycle we could treat the cycle and trees as separate cases. Since we can do a single cycle with an out-degree of one and trees with an out-degree of 1, we can orient the attached trees towards the cycle. These types of graphs are illustrated in figure 6a.

- **Out-degree of two**
  For a graph with two cycles or more in a single connected component we need at least an out-degree of 2, this is illustrated in figure 6b. We need to connect the cycles, since the points in the cycles all have an out-degree of at least one, we need an additional link for connecting the cycles. Cycles can be only be created by adding a additional edge, as illustrated by Euler's formula and every additional face can be created with only one additional edge, no points have to be added. This increases the number of links, without adding points to place them on.

- **Out-degree of three**
  Since we stated as a requirement that lines do not intersect, our graph is planar, therefore our worst case would be a complete planar graph. A maximum out-degree of 3 can always be achieved on any planar graph as proven by Chrobak and Eppstein [10], they call this a 3-bounded orientation and show that they can achieve a 3-bounded orientation or less in linear time. We can be certain we need a 3-bounded orientation if the number of edges is larger than two times the number of points as illustrated in figure 6c. This figure illustrates the most basic graph where we need at least a 3-bounded orientation.

Depending on the number of cycles in a graph we can determine if we can achieve a maximum out-degree of 1 or 2 and higher. Whether we need a maximum degree of 2 or 3 is dependent on the structure in a graph. However if a graph is evenly connected, i.e. each point has a degree of 2, 4 or 6, there must be an Euler path. If we compute the Euler path then we can use this to achieve an outbound degree of 1, 2 or 3. We could use Fleury's algorithm to compute the Euler path, only if there are only points with an even degree. The highest degree divided by half will be the maximum outbound degree. However this can potentially be very high and therefore is not a desirable solution in most cases.

We can easily test whether it is possible to get a maximum outbound degree of 1. For this each connected component of the graph can only have one cycle. If a connected component has more than 2 cycles we have at least a maximum outbound degree of 2. When we want to test if we can handle a graph with an outbound degree of maximum 2 this becomes more difficult. We do have an initial indication that is if the average degree is higher than 4 a maximum outbound degree of 3 is necessary. However even with a lower average an embedded sub graph of the graph could cause potential problems, i.e. there are points with a degree higher than 4. Therefore we propose two methods, one that uses

a divide and conquer approach and one that orients the graph by orienting the points that have the lowest risk of becoming high outbound degree points first.

We take the graph and select all points with a degree of 1 and 2. We oriente the associated edges from these points with the points as origin and remove these from the graph. We repeat this step with the resulting graph until there are no points left or that every point has a degree of 3 or more. If there any points of the graph left, this graph can be oriented using the algorithm proposed by Asahiro et al [11] to calculate the maximum 3-bounded orientation in linear time.

Our second approach is very simple and straightforward. In this approach we iterate through the collection of points until no edge is undirected involves a priority queue and a calculated value. For each point in the graph we define a value that indicates the risk of the point becoming a point with a high number of links, we call this the $LinkRisk$. This value is calculated by the number of connected undirected edges E and the number of attached links with the point as the source. When calculating this value for the first time we include the already orient edges due to the constraints from the criteria.

$Linkrisk$ is then calculated as $LR = \frac{e}{2} + L$. These points are put in a priority queue, with the points with the lowest $LinkRisk$ are given the highest priority. When a point has no more undirected edges left it is removed from the priority queue. This is done until the priority queue is empty and all the undirected edges are directed. This way only points with low connectivity are preferred and local hotspots, i.e. points with high connectivity, are avoided. Additionally we could make this method more versatile, by introducing a value that is based on an additional heuristic that determines which points are the most preferable to place links on. $LinkRisk$ is then calculated as $LR = \frac{e}{2} + L + H$.

4 Algorithms

Our algorithm is divided in a couple of steps, where the output is passed on as input to the next step. In the GUI our algorithm is meant to assist the user and all of these steps can also be individually executed, in which case the user decides what calculation to perform on a data collection. The first step is to import the drawing to our graph data structure. This graph is then pre-processed to ensure there are no long edges, these are cut in smaller edges. Also this step ensures that the graph is made planar if it was not already planar. The simplification step can be performed multiple times, set by the parameter $Passes$. This creates a cleaner output, since in certain cases the ambiguity check can be too restrictive, because it only should prevent placement of confusing points, these points are not there yet in the first pass. After the first pass the influence of ϵ is minimized, since the curves are already simplified.

After the simplification passes have been completed we rebuild the graph from the simplified polylines. This is necessary due to the step where we replace the edges with shortcuts and original edges area lost. However we still have a point ID and for each connected endpoint of the polylines this must be the same, therefore we can easily rebuild the graph. After this the graph is directed to determine link placement for the asymmetric variation. Algorithm 1 breaks down the global algorithm and the order of the specific steps that need to be executed transform a line drawing into a graph that can be used to render a connect the dots puzzle with our renderer. The renderer determines the style and the visual representation of the links and the puzzle.

Algorithm 1 Process drawing

Input: Drawing D
Output: DirectedGraph DG
  Import D to Graph G
  PreProcess(G)
  $P \leftarrow$ Subdivide(G)
  for $i \leftarrow 1$ **to** $Passes$ **do**
    $S \leftarrow$ Simplification(G, P)
  end for
  DirectedGraph $DG \leftarrow$ DirectGraph(G)
  Return DG

4.1 Subdivide graph

We need to subdivide the given graph into polylines because each polyline is processed individually during the simplification pass. To do this we search for the endpoints of a polyline, i.e. any node with a degree other than 2. Then we simply traverse the connected edges and report each point of each traversed edge until another endpoint has been found. If there are any nodes left we consider these to be isolated cycles and pick an arbitrary point on the cycle and let this be the end and start point of the polyline.

4.2 Line simplification

For our line simplification we use a modified Imai and Iri's algorithm. This algorithm determines the valid shortcuts from the points on the polyline, from which it builds a graph and employs Dijkstra's shortest path algorithm to find the optimal line simplification. We process each polyline individually and all the operations and queries are performed on the input, results are stored separately and the input graph remains intact until all polylines are processed, after which the graph is rebuilt from the

result. Our implementation focuses on finding the simplification with the smallest number of points within our criteria.

Algorithm 2 Simplification

Input: Polylines P
Output: Polylines R
  for all Polyline p **in** P **do**
    $R \leftarrow$ simplify(p, $DefaultMinLength$)
  end for

We prefer a simplification that has only edges longer than our minimum link length, this is used in our edge evaluation. However this can lead to unsolvable graphs, therefore when Dijkstra does not find any path, we recursively simplify the original polyline with a lower minimum length until this value becomes 0, in which case the original polyline is considered valid and therefore Dijkstra's shortest path always returns a result. When we have our simplified polyline we post-process to detect edges that are too short to place links on. At those edges we insert the points from the original polyline chain, so the original detailed outline is preserved on the predrawn lines in the puzzle.

Algorithm 3 Simplify

Input: Polyline p, $MinLength$
Output: Polyline r
  $g' \leftarrow$ FindShortcurts(p, $MinLength$)
  $g'' \leftarrow$ FindShortcurtsReversed(p, $MinLength$)
  $g \leftarrow g' \cap g''$
  $r \leftarrow$ DijkstraShortestPath(g)
  if r is nil **then**
    $r \leftarrow$ Simplify(p, $MinLength$ - 10%)
  end if
  PostProcess(r, p)
  return r

As stated before we can easily modify and extend this algorithm to suit our needs by introducing additional criteria. In our case we use the default epsilon based distance measure that defines the maximum error a simplification is allowed have to the original polyline and a minimum distance for a shortcut. Additionally we check the ambiguity of a candidate shortcut by checking an area for any points that are ambiguous to the candidate shortcut. We do this by building a search tree in which we put all other points sorted on their angle with respect to p_i. This approach and its running time is further discussed in the evaluation section. We base our algorithm on the algorithm as proposed by Chan and Chin[5]. We explain their method briefly.

Consider a polyline p with n points. A graph g with valid shortcuts is constructed. A shortcut $p_{i,j}$ is only allowed when the maximum induced error is below the given value ϵ. The error of a shortcut is defined as the maximum error of the halflines $h_{i,j}$ and $h_{j,i}$, therefore we iterate through the input forwards and backwards, giving g' and g''. The intersection of g' and g'' results in g. When iterating through p for each point at a wedge W is constructed by the point p_i and a disk of the size ϵ centered at p_j. Foreach following p_j the shortcut $p_{i,j}$ is only accepted if the point lies within W. W is updated after each evaluation of $P_{i,j}$ as the intersection of the wedge W and the wedge formed by p_i and the disk p_j. When W becomes empty any following point cannot be valid within ϵ and further evaluation for p_i is terminated. With g being the intersection of the edges of the graphs g' and g'', the optimal simplication is found by calculating the shortest path within g for p_1 and p_n.

When an edge is accepted with respect to ϵ distance and the minimum edge length, we evaluate the ambiguity of the edge with procedure 5. This is done by looking for any other points that could create confusion when this shortcut is used in the puzzle. We do this by checking the surrounding area of the edge in which there should be no points other than the points that the shortcut would exclude from the polyline. This area is defined by two angles, α and β, beginning from the source point and ending at the destination point of the edge as illustrated in figure 4a. If there are any points in that specific area the shortcut is considered invalid. Since the simplification algorithm iterates through the polyline in both directions this check is also performed in both directions. For a candidate shortcut to be valid, at least one of the directions has to be valid, since we only need one valid direction to place a link in the asymmetric link placement and exclude all invalid edges. Therefore a shortcut is tagged on its ambiguity instead of being excluded and this value is evaluated during the calculation of the intersection. In short this means that a shortcut is valid, when it is within ϵ error, longer than the minimum length and has at least one of the two directions being valid on its ambiguity.

Algorithm 4 FindShortcurts

Input: Polyline p with n points, Graph G, $MinLength$
Output: Graph g with valid shortcuts

```
for i ← 0 to n − 2 do
  g.Points_i ← p_i
  T = BuildAngleSearchTree(p_i, G)
  j ← i + 1
  T.Remove(p_j)
  W ← Wedge(p_i, p_j)
  while W ≠ ∅ and j < n do
    if p_j ∈ W then
      if Eucladian(p_i,p_j) < MinLength then
        e = Edge(p_i,p_j)
        e.Ambiguity ← EvaluateAmbiguity(e,T)
        g_i.Edges ← e
      end if
    end if
    W ← W ∩ Wedge(p_i, p_j)
    s ← s + 1
    T.Remove(p_s)
  end while
end for
return g
```

Procedure 5 EvaluateAmbiguity

Input: Edge e, Tree T
Output: $true$ or $false$

```
Define 2 halfplanes S along the e with α and β
if T.Empty(S) = false) then
  return false
end if
return true
```

4.3 Dijkstra, shortest path

When we have built the graph with all the valid shortcuts, we perform a slightly altered version of Dijkstra's shortest path algorithm to determine the shortest path, which results in our simplification. The number of steps per point is our first weight, which results in the path with the lowest number of points. However, often there are many different paths, with an equal number of steps, since the graphs are often very dense and highly connected. In the case that a path is found with a an equal number of steps to a point we compare a second metric. This is the total Euclidean distance of the path. In our case we maximize this component. So the path with the longer Euclidean distance is preferred over the shorter path. The reason for this is that the longer path must have placed its points more in the curves and therefore contain more of the more significant features of the line.

4.3.1 Direct graph

Since the output is an undirected graph and we only place one link for each edge, we direct the graph to determine the link placement with algorithm 6. In the input each undirected edge is represented by two directed edges. First we remove all directed edges that are invalid for that direction, since a shortcut is allowed in the previous step when only one of the directed edges was valid. We add the valid directed edge as link. Then we calculate the $LinkRisk$ value for each point and store the points in a priority queue according to the $LinkRisk$ value, with the lowest $LinkRisk$ given the highest priority. Then we process the points with the highest priority until there are no more points left. Processing a point is assigning a single link and updating the $LinkRisk$ of the affected points or removing them from the queue. When a link is assigned, the edge and the corresponding edge in the opposite direction are removed from the graph. If a point has no more edges we remove the point from the priority queue.

4.4 Evaluation

4.4.1 Intersections

We do not provide a definitive method to prevent a polyline simplification from self-intersecting and intersecting with other simplified polylines. However since we assume our input is high in detail, i.e. the interdistance between points on the polylines is small, we reduce the chance of intersections to a minimum. This is due to our ambiguity check, which requires some distance between polylines and eventually defaults to the original polyline which cannot intersect since our input is planar. To eliminate the chance of intersections entirely, we should extend our method to incorporate the method proposed by Berg et al [6] in our modified Imai and Iri's simplification algorithm. This can easily be done since their approach is similar to our approach and therefore our method only has to be extended to incorporate their method.

4.4.2 Angular distance

A link should be clear and not overlapping to be able to indicate a direction to the next point. For this a certain angular distance between links incident on a single point is required. In our algorithm our ambiguity check ensures this angle is at least α. When a link is placed on a point, the ambiguity check is performed with this point as origin. Therefore this point must have a wedge shaped area with angle α in the direction of the destination point that is clear of other points.

Lemma 4.1. *The angle between links incident on a point is at least α.*

Proof. Consider an puzzle point p with two or more connected lines. Any link placed on p must have a wedge shaped area with α that is clear of any other lines. Therefore a link placed on p will have an angular distance of at least α to any other link incident on p. □

Algorithm 6 DirectGraph

Input: Undirected Graph G
Output: Directed Graph DG

```
for all DirectedEdge e in G do
  if EvaluateEdge(e) = false then
    AddLink(G.FindOpposingEdge(e))
  end if
end for
PriorityQueue Q
for all Point p in G.Points do
  Q ← p
  p.LinkRisk ← q.Edges.Count / 2 + p.Links.Count
end for
while Q ≠ ∅ do
  p ← q.Top
  e ← p.Edges.First
  AddLink(he)
  if p.Edges = ∅ then
    Q.Remove(p)
  else
    p.LinkRisk ← p.Edges.Count / 2 + p.Links.Count
  end if
  if e.Destination.Edges = ∅ then
    Q.Remove(e.Destination)
  else
    e.Destination.LinkRisk ← e.Destination.Edges.Count / 2 + e.Destination.Links.Count
  end if
end while
for all Point p in G.Points do
  DG.AddPoint(p)
  DG.AddEdges(p.Links)
end for
return DG
```

4.4.3 Output size

The size of our output is not bounded by a multiplication or factor of a constant c to the input n, because the number of extra points we place in the output is dependent on the interdistance of the lines in the input. The number of nodes of a subdivision simplification in the output is relative to the interdistance between the lines in the input. Since this is valid for a single subdivision simplication, this is valid for the complete output. Our output is the graph composed from the subdivision simplifications.

Lemma 4.2. *Let n be the number of points in the input graph and n' the number of points in the output. There is no constant c so that the product or factor n and c defines the upperbound of puzzle points n' in the output.*

Proof. Consider three parallel lines with interdistance d, for every x distance on each line there must be a point. With d defined as $d = \frac{\frac{1}{2}\sin 2a}{x} . \frac{1}{2}$, which can be written as $d = \frac{\sin 2a}{4x}$. □

Lemma 4.3. *The number of points in a simplication depends on the interdistance of the line and other polylines in the input.*

Proof. Consider line L with length l and interdistance d to other elements e, then the number of added points n on L is at most $n = \lceil \frac{l}{x} \rceil$ with x defined as $x = \frac{d}{\sin a} . \cos a - \frac{d}{\sin b} . \cos b$ □

However, since we have a minimum length and ensure the edge distance of the input is smaller than this minimum length our output is at most our input. In this worst case example, the drawing would be entirely pre-drawn. However if we would not have the minimum distance requirement and not default to the input, then the algorithm had to add intermediate points until the other requirements have been met. So when we do enforce the minimum length of a shortcut, then we can say there is no constant c lower than 1 that defines the upper-bound of our output to the input n and m.

4.4.4 Running time

Given a graph with n points and m edges we can easily detect the subdivisions in linear time, by iterating over n and m. This results in the linear running time of $O(n + m)$. Our simplification algorithm for a polyline is not as straightforward and has multiple steps, which are computationally more intensive. This is largely due to the ambiguity check, which checks a potentially large number of shortcuts on ambiguity, which includes simplex queries on a two dimensional plane. The large number is a result of the use Imai and Iri's simplification algorithm [12], which results in a possible quadratic number of shortcuts. As demonstrated by Chin en Chan [5], Imai and Iri's can be performed in $O(n^2)$ time worst case, resulting in $O(n^2)$ shortcuts.

For the ambiguity check we need to query 2 simplexes for each shortcut in both directions. However we need to exclude the points that a candidate shortcut would bypass. For this we need a search structure with which we can do fast counting queries and fast updates. For this we could combine a partition and a cutting tree to create a trade-off between storage space, update- and query time. This trade-off is proposed by Agarwal and Erickson [13]. With this structure we can achieve with $m^{1+\epsilon}$ storage for any $n \leq m \leq n^2$ a query time and update time $O(\frac{1}{\epsilon} n^{\frac{1}{3}})$. If we choose for storage $m^{1+\epsilon} = n^{\frac{4}{3}+\epsilon}$ we can achieve $O(\frac{1}{\epsilon} n^{\frac{1}{3}})$ query and update time, which results in $O(n^{\frac{1}{3}})$ time for every ambiguity check.

For every subdivision with n points and at most n^2 shortcuts we need to check 4 simplex counting queries and 2 updates against a search tree of $O((n + m)^{\frac{4}{3}})$ storage, with m being the other points of the graph in the plane. Each query and update costs $O((n + m)^{\frac{1}{3}})$ time. This results in the running time of $O(n^2(n + m)^{\frac{1}{3}})$ for the simplifcation of a polyline subdivision.

However we could achieve a faster running time. To do this we construct a search structure in which we sort all other points on angle around p_i. This search structure only works for p_i and is built

for every p_i instead of once for every subdivision. For this we use a balanced binary search tree. At each node a subset with nodes is stored for a particular angular interval. For the associated structure in which the subset is stored we use a data structure for half plane emptiness querying as proposed by Chazelle et al [14]. This structure allows for $O(\log^2 n)$ time to decide whether the area is empty or not and has a size of $O(n \log n)$. We do need to make the structure dynamic, because for each possible shortcut we need to delete a point from the search structure. We do this by standard methods as proposed by Overmars [15], resulting in a search structure with $O(\log^3 n)$ update and query time.

Now for every subdivision with n points and n^2 possible shortcuts and m other points, we need to perform a query and deletion in the search structure in $O(\log^3(n+m))$ time. This is done for every possible shortcut resulting in a running time of $O(n^2 \log^3(n+m))$. Our worst case is that the entire graph is only a single line, since n and m are both a subset of the graph and both are bounded by the points in the graph. In this case n is equal to $n+m$, with m being 0. This results in worst case running time of $O(n^2 \log^3 n)$ time, where n is the total number of points in the graph.

5 Implementation

We propose a multithreaded user assisted environment where each step of the algorithm is executed individually and from each step the result can be evaluated. The user can use this to make changes in the parameters or alter the data with the editor.

5.1 Data structure

We use a custom data structure that consists of points and edges. A point holds a collection of all outgoing edges. An edge has a source point and a destination point. When our graph is undirected, a directed edge exists in both directions. Additional values can be added to both objects during calculation. For example length, distance metrics, if traversed and other flags. These are stored in a tag library. This makes the objects easy to use and algorithms can be implemented or extended quickly without having to overhaul the data structure. These tags are not saved, since they are only used during calculation. However some additional properties are saved, such as whether points only need to be drawn for detail and are not puzzle points. These points are used for the detail lines that are pre-drawn in the puzzle.

5.2 Loading and saving data

We use a custom XML schema to serialize and deserialize our internal data structure, which uses points and edges. The reason we use XML is that we can ignore or persist properties of our data structures and it is very easy to use. Another reason is that this is a readable format, this way output values can be read and evaluated. During the processing of a file, the data is saved so we can evaluate the raw data if any anomalies occur in the final puzzle. A puzzle can also be exported to a BMP image. During the export the selected renderer with its parameters is used and an image with the size of the canvas is produced.

5.2.1 Image importing

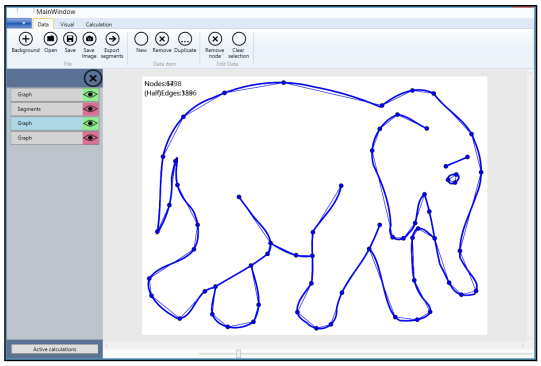
An image can be imported into the program and the lines can be manually traced to create a graph. Since this is painstaking work to achieve a high level of detail, additional points can be added during the segmentation step. However these are simple linear interpolated points and do not increase the level of detail, but do increase the number of possible shortcuts. This is a useful method to draw a fast test case, but for a more complex drawing or test case the use of a third party SVG editor is recommended since this method is otherwise very time consuming.

5.2.2 SVG importing

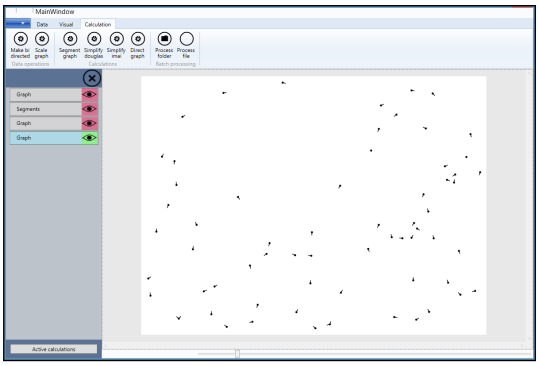
Since there is a wide variety of SVG drawings available and proper editors such as Inkscape are available to use, we implemented an SVG importer to import SVG drawings and convert Bezier curves and other shapes to a polyline graph. We use a method from the Microsoft WPF library to convert the curves to a polyline. This method requires a tolerance value and results in high detail polyline approximation of the curve. However long edges are still possible, this is the case when the line is straight. Therefore during post processing these long edges are cut in smaller edges with the length of the maximum allowed edge length to get an even spread of points across the drawing.

Not every SVG drawing is immediately fit for importing. In some cases drawings have to be prepared manually. This can be the removal of details, such as drop shadows and highlights or in some cases duplicate lines. The importer is able to fix some aspects, such as scaling to a bounding box, adding borders and repositioning. After the import is done, the drawing is post processed. As

Figure 7: Implementation, GUI



(a) Multiple visible layers



(b) Puzzle renderer

mentioned before during this step longer edges are cut in smaller edges but it also ensures that the input is a planar graph, because the SVG input is not necessarily a graph but merely a collection of shapes and lines. First we check for intersecting edges and remove all these intersections by replacing the two edges with an intersection point and four new edges. Endpoints that are close together are joined into a single intersection point. Noise such as single points, and very small lines are removed. All these operations have tolerances that can be adjusted.

5.3 Calculations

Calculations can be run synchronously and asynchronously. Most of the calculations run asynchronously and the progress can be monitored by the user. When the calculation finishes an event is launched and the output is processed or presented to the user. When multiple calculations have to be run, a parent calculation is defined, which runs the calculations synchronously and passes the data, the output and the input from the calculations, along to the next calculation. This parent calculation can still be run asynchronously. In each calculation a set of parameters is required. These are presented to the user when the button associated the calculation is clicked. When an iteration is performed over an input parameter these values are set by a parent calculation.

5.4 Algorithm

We implemented both Douglas-Peucker's algorithm as well as Imai and Iri's algorithm. However we continued development on Imai and Iri's algorithm and halted further development on Douglas-Peucker's algorithm. While our implementation of Imai and Iri's algorithm includes our additional criteria, our implementation of Douglas-Peucker's algorithm is based on the standard proposed implementation with no additional criteria. While we propose an efficient algorithm, we mainly focused on implementing the different criteria and creating an algorithm that creates a solvable and aesthetically appealing puzzle. Our implementation has a cubic running time, because we linearly evaluate all the other points in the plane, with our ambiguity check for at most a quadratic number of shortcuts to the input.

5.5 Interface, editor

We provide a basic editor for our graph data structure, in which points can be added, selected and removed. The editor can contain multiple data items, which are presented as layers. Each layer has a

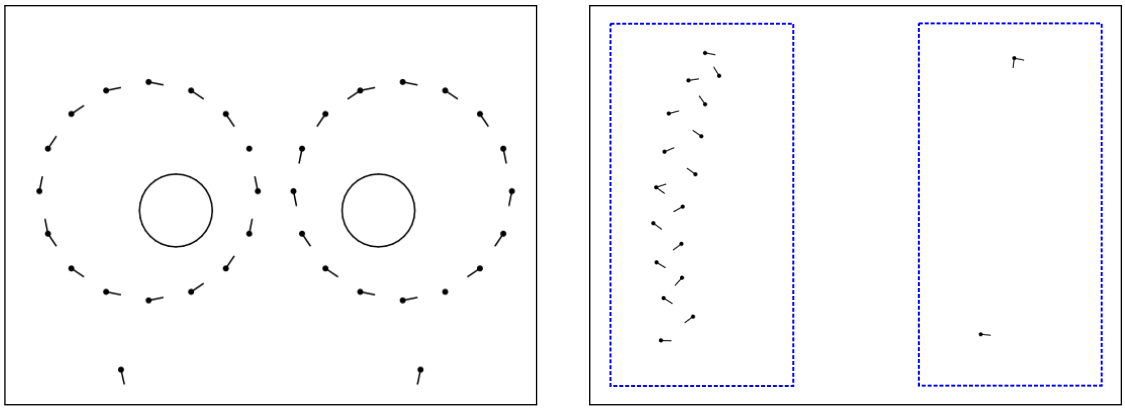
visibility toggle. Only the selected data item is active for editing, but the user can easily switch layers by clicking on one of them. Also other data is presented as a layer, such as intermediate data from a calculation, but they are presented read only. The editor also supports basic navigational operations such as zooming and panning.

By combining different layers, a simplification can be evaluated to the input drawing, as illustrated in figure 7a. This useful to observe where the algorithm placed the points on the lines and whether the resulting simplification is complete. For each layer a different renderer can be assigned, with different settings. This way it is possible to compare different styles of puzzle or the computed puzzle with the original input.

5.5.1 Renderers

For a data type multiple renderers can be defined, for example in case of a graph we have a simple renderer, an evaluation renderer and a final renderer. The evaluation renderer incorporates the area check so we are able to evaluate whether the area check has performed correctly. The final renderer is used to render the final puzzle. We incorporated different visual variations and parameters, such as link size, link type, pre-drawn line width, link width and point size. These can be evaluated in real-time and when a user modifies a parameter the visual representation is updated instantly. Figure 7b illustrates the different render options and shows an example of a puzzle renderer.

Figure 8: Observations ϵ



(a) Conservation of curves with low ϵ

(b) Loss of detail

6 Experiment

We have processed a set of drawings with various characteristics by our algorithm. These drawings differ in density, connectivity and in shapes, namely curved, straight or a combination of both. We evaluate different values for the parameters of the algorithm and evaluate which values from the range provides the most suitable line puzzles.

6.1 Setup

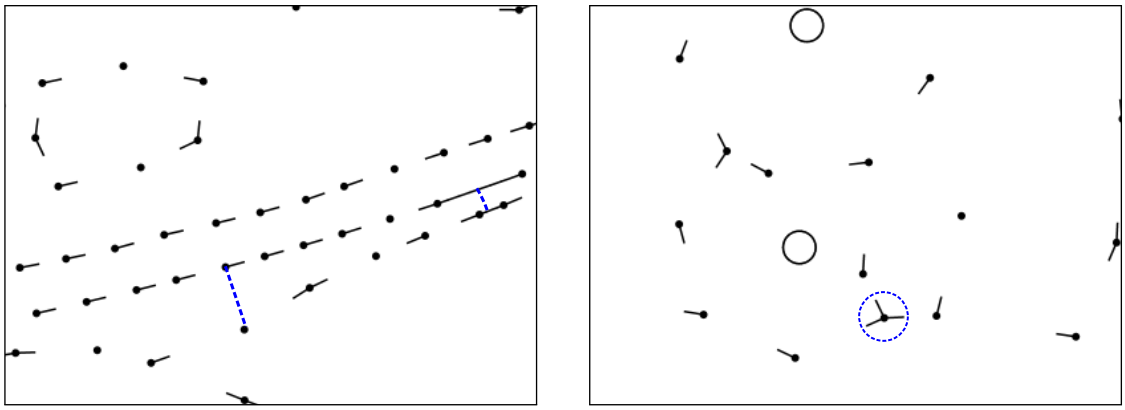
For our evaluation we collected data by running the algorithm over a set of SVG files. A visualisation and a table with the specifics of this dataset is provided in appendix **C** and **D**. These files are imported and processed automatically. During this process the drawings are scaled to a bounding box to make a comparable case. During these runs we iterated a single parameter over a range with intervals. We have chosen parameters that would likely have the most impact on the simplification and would make sense to alter in practical terms depending on the input or desired level of detail in the output.

The first parameter controls the maximum error for the Imai and Iri's line simplification algorithm, namely ϵ. We iterate from 2.5 to 52.5 with a step size of 2.5. The second parameter is the α angle that defines the area that is used in the ambiguity check. We use a range from 2.5, to 35 with a step size of 2.5. We don't use 0 values, because for ϵ this would mean no simplification takes place and for α it results in intersections and other artefacts which would make the use of the resulting data difficult.

6.2 Puzzle observations

We generated a set of images based on the dataset and the iterations over the parameters. We take a closer look to see how these parameters impact the different aspects of the puzzle. When the parameters are in the upper or lower range some visual artefacts can occur. We look how our algorithm performs at these values and what kind of artefacts occur. Furthermore we look at the resemblance and solvability or in other words the overall usability of the puzzles. From this we conclude an ideal range of values for the parameters and suggest a default value.

Figure 9: Observations α



(a) Influence of interdistance

(b) Increase in degree

6.2.1 Parameter ϵ

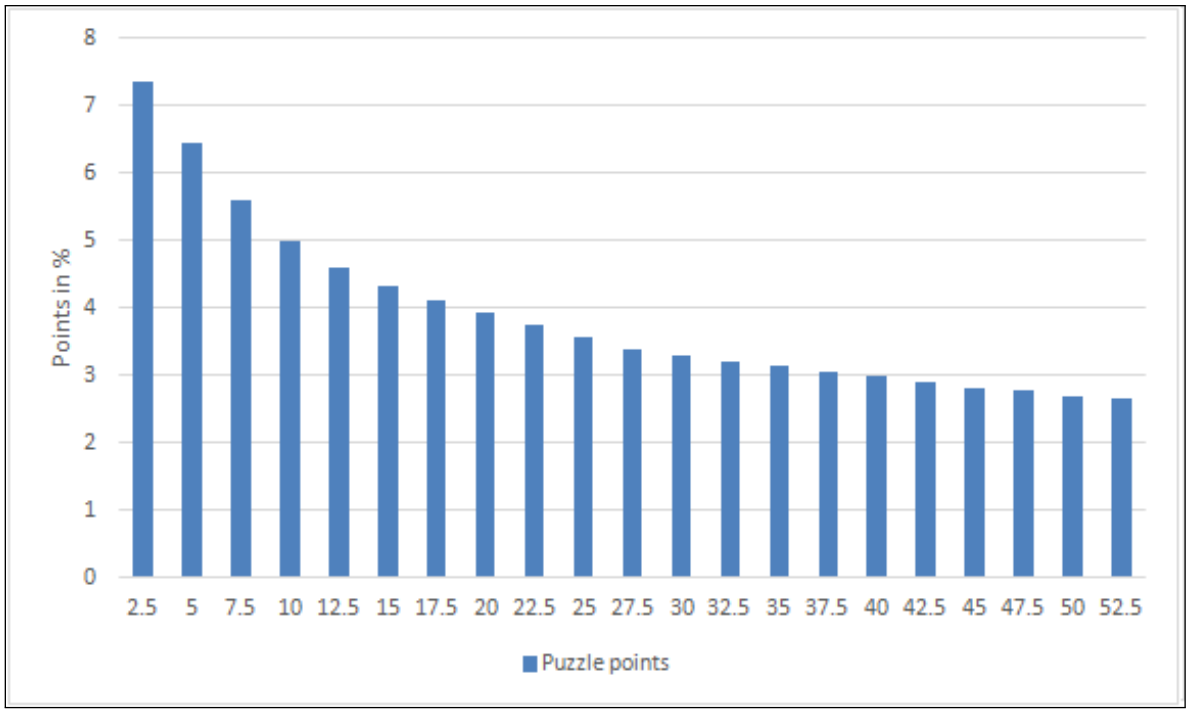
If we look at ϵ we observe that this value has a great influence on the overall simplification of the drawing. Low values result in detailed curves, as illustrated in figure 8a, with the drawback that a puzzle becomes easy recognizable. Since our implementation defaults to the original line when it is not possible to draw a link, large parts of the original drawing are pre-drawn, in particular the curvy parts. When we use a high value for ϵ then certain details are lost. Global curves are often mostly maintained but small features and texture-like patterns, as illustrated in figure 8b are lost. In high density areas our ambiguity check enforces a level of detail and is the impact of high ϵ value limited. The most optimum value for our set of data, which is dependent on the type of drawing and scale, is between 10 and 30. We have chosen 15 as a default value.

6.2.2 Parameter α

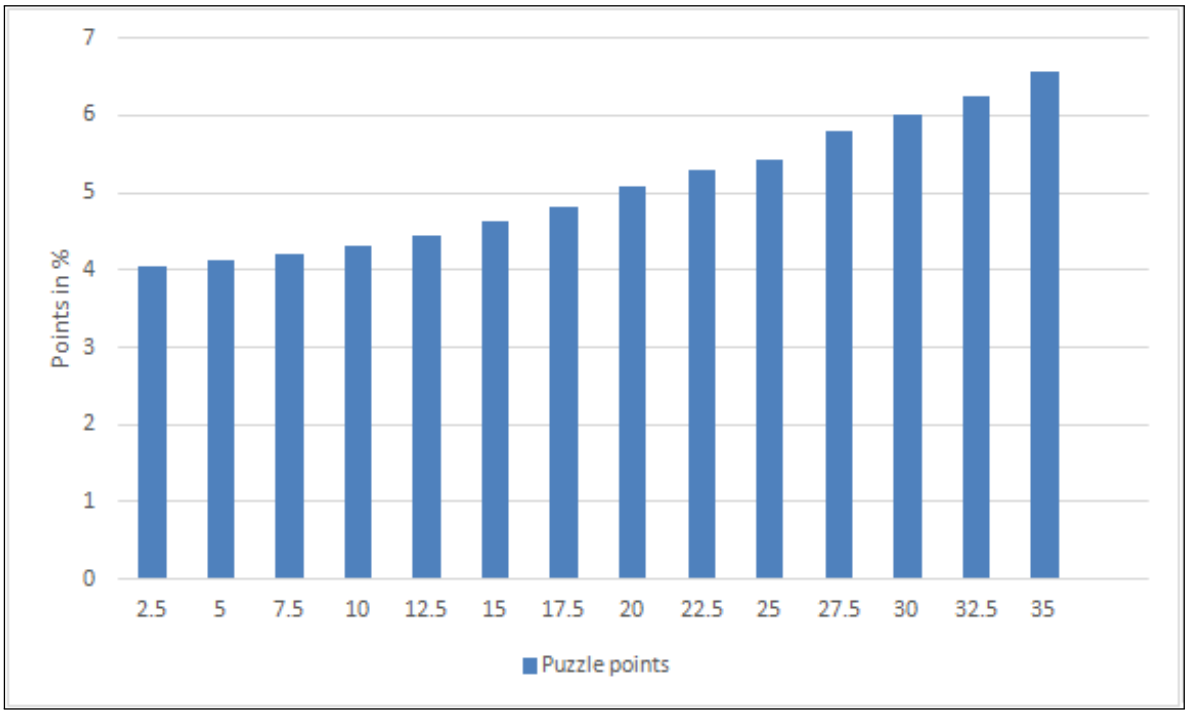
When the value of this parameter increases, the angle and therefore the surface of the area check increases and it becomes more likely a link is not valid due to confusing points of lines outside the shortcut segment. This results in a puzzle with a higher level of detail, since points have to be closer together. At first this becomes clear in high density areas, this is an area with high number of lines close together. Therefore the effect of this parameter is heavily influenced by the interdistance of surrounding lines. This is illustrated in figure 9a. Because of this increased point placement the drawing becomes more easily recognizable, especially when the length of pre-drawn lines increases. From 20° and up most of the finer details are pre-drawn instead of being represented by links.

A greater angle of α can also result in a higher maximum degree. Since all invalid half edges are removed before orienting the graph, this can result that in some cases we only have one option for link placement. The chance for this increases when α increases. This can lead to a point that has only directed edges with and therefore all links must be placed on that point. This is illustrated in figure 9b, where all the surrounding points prevent the orientation from any other point. However it should be noted that this only occurred on occasion with large angles for α, i.e. greater than 25°. Depending on the difficulty level that is desired a viable value for α would be between 5° and 17.5°, as default value we choose 10°.

Figure 10: Puzzle points

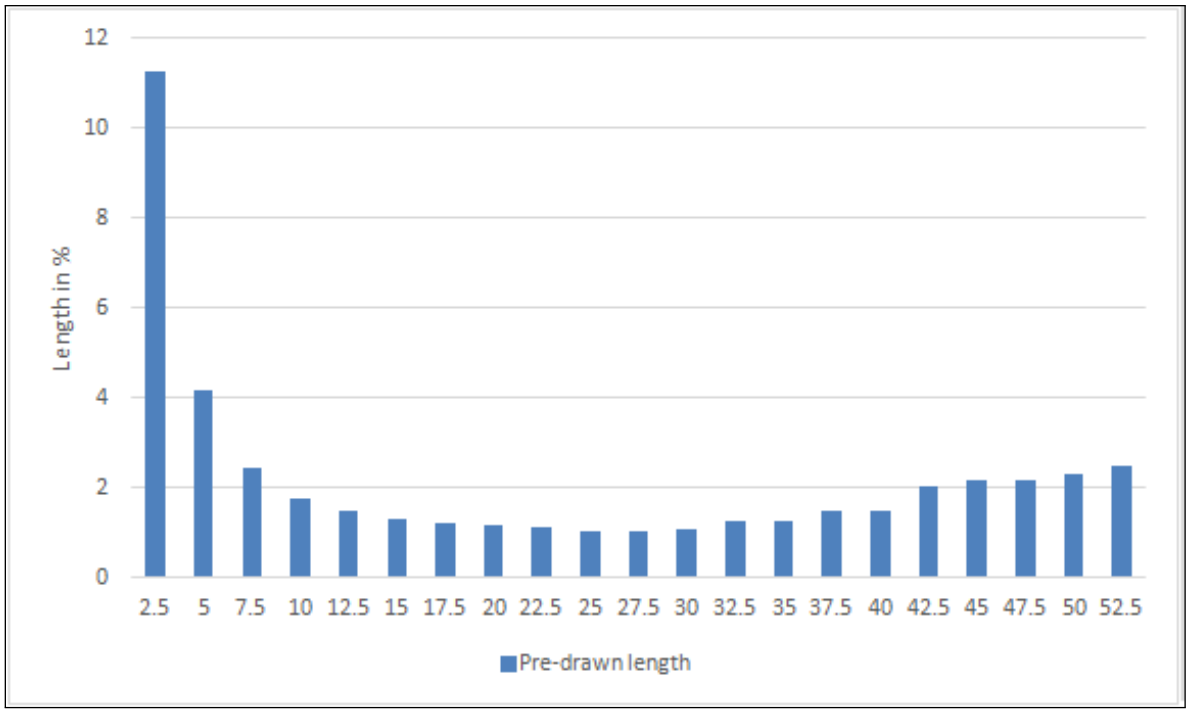


(a) Iterating over ϵ

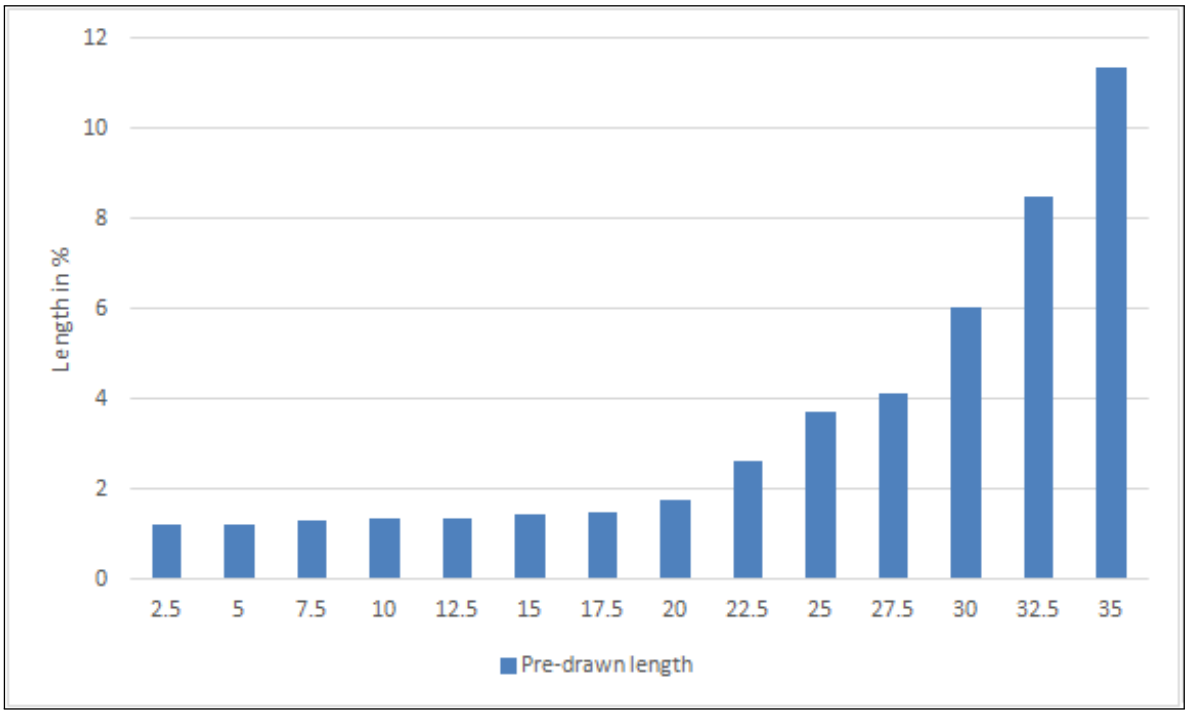


(b) Iterating over α

Figure 11: Pre-drawn length



(a) Iterating over ϵ



(b) Iterating over α

6.3 Performance

When evaluating the simplification we look at the relative number of points that remain in the puzzle with respect to the original drawing. In this we identify two different aspects, puzzle points, which are the points on which links are placed, and pre-drawn line length, this is the part of the puzzle that is pre-drawn from the input drawing. This is an important factor in the quality of the puzzle. We want to minimize this as much as possible, since this can easily give away the solution of an unsolved puzzle. Also we look at the length of the simplification with respect to the original drawing. We want to maximize this value, since we want the solved puzzle to look as much as possible to the input.

6.3.1 Puzzle points and pre-drawn length

The charts for the number of puzzle points, in figure 10, are the average percentages of puzzle points with respect to the input. The pre-drawn length, in figure 11, is the percentage of the input drawing that is pre-drawn in the puzzle. When we look at ϵ we see a very steep initial decline in the length of pre-drawn lines and a steady decline in puzzle points. A low ϵ will result in short distances between points and therefore a lot of puzzle points have to be placed or lines have to be pre-drawn. However the length of pre-drawn lines steadily rises again at about $\epsilon = 30$. After visual inspection this is caused by certain isolated features in the drawing that are formed by cycles, for example the eyes. When ϵ is large enough, that is larger than the diameter of the cycle, the cycle 'collapses' and loses visual relevance with the original feature. It is then represented by a line or a dot. We prevent this from happening, by replacing collapsed features with the original polyline. This explains the increase in length of pre-drawn lines.

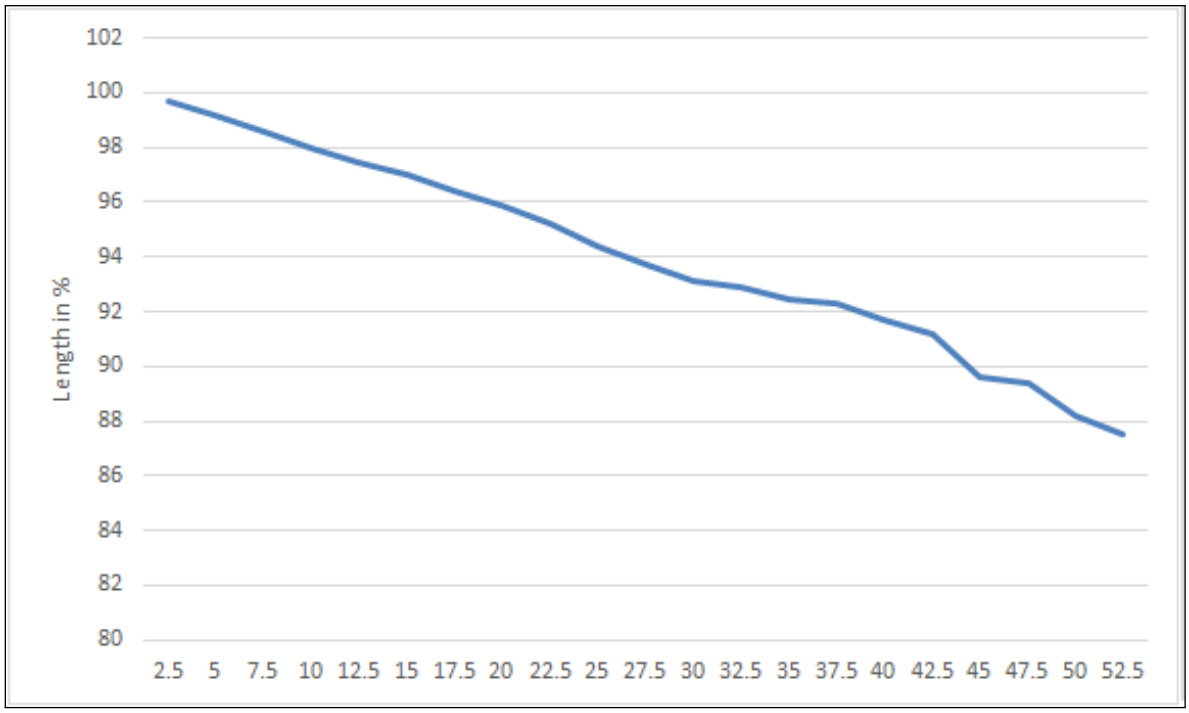
If we look at α we see initially a very stable trend, even though the point placement differs. This suggests that there are still enough alternatives to place a puzzle point instead of having to place a lot of additional puzzle points or pre-draw lines. Only after $\alpha = 20^\circ$ we see a negative impact on the length of pre-drawn lines and number of puzzle points. Both these number continue to rise steadily. This indicates that more puzzle points are placed closer together as well as more of the puzzle is pre-drawn. When lines are pre-drawn, then it consumes space on which otherwise puzzle points were placed. This explains the steeper rise of the pre-drawn lines in comparison to the puzzle points.

6.3.2 Conservation of length

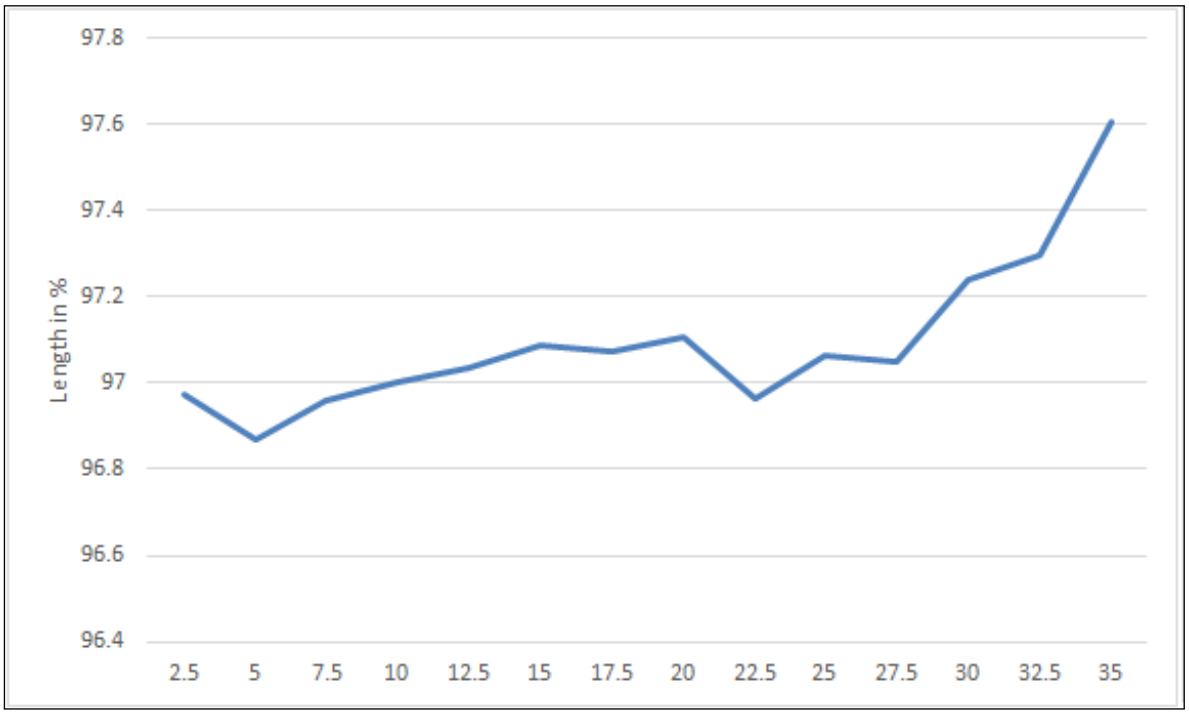
We use the conservation of length as a measure to evaluate the resemblance of the puzzle to the original drawing as shown in figure 12. With parameter ϵ we see a steady decline as detail is reduced. We do see some fluctuations in the decrease at ϵ values higher then 30. However this can be attributed to our observation earlier, when certain features, for example the eyes, lose visual relevance to the input and is replaced by the input.

When we look at the influence of α on the length of the puzzle, we see the same as with point reduction. At lower values, that is below 20°, α mainly influences point placement and has not a large impact on the length. However when α increases and the simplification becomes more constrained, we see a relative steep rise in length. We would expect this to continue to rise until the resulting puzzle is almost completely pre-drawn. From $\alpha \geq 22.5^\circ$ every increase has dramatic consequences for the evaluated area of the ambiguity check and therefore constrains the possible simplification severely.

Figure 12: Conservation of length



(a) Iterating over ϵ



(b) Iterating over α

7 Conclusion

We explored a viable alternative to numbered connect the dots puzzles and a number of visual variations. We identified the criteria for a good puzzle and proposed an algorithm that includes these criteria to create a puzzle that is solvable without additional tools. We implemented this algorithm in a proof of concept of a production environment with a GUI. We evaluated our implementation on a dataset with a wide variety of types of drawings and a range of parameters. We can conclude that there is a range of parameters that results in solvable puzzles within our criteria while remaining a strong resemblance to the input drawing. We can also conclude that for the default values we have selected, the algorithm will generate a solvable puzzle for a wide range of input.

8 Future work

Even though we propose a complete implementation there are still areas to do relevant research concerning our approach. These vary from user studies to additional theoretical research on the algorithm. We suggest a couple of possible targets that could be studied to improve our method.

Additional heuristics could be applied during the orientation of the graph. Our goal was to minimize the number of links placed on a point with respect to our criteria, this being the ambiguity. However there could be cases where a higher degree would be preferable over the link placement on a different point. For example a pre-drawn line with a point at the end, the links placed on that point are visually less appealing then links placed on other points. This and other factors could influence the preference to place a link on a point, even if there are other links placed on the same point. This could very well be an optimization problem depending on the applied heuristics.

Our orientation of the graph is not the most optimum, this is partly due to our criteria. However after the criteria are applied and some edges are oriented we still do not find the optimum solution. This problem could be defined as orienting a partial undirected graph. A deterministic approach could be applied to find the optimum solution with respect to our criteria. A different approach could be that we distribute weights on the edges, depending on an aesthetical criteria. Then solve the orientation for the lowest outbound degree based on this weight.

The simplification algorithm could still be improved and extended with additional criteria. Different distance measures could be applied in the simplification algorithm. We still have situations that are problematic, for example an intersection with outgoing lines with an angle lower than α. In that case link placement is not possible and the original line will be pre-drawn. Our current algorithm only places points on the original line, however in these cases a solution could be found outside of the original lines.

We also observed that the optimal values for the parameters differs for each type of drawing. We have a range of generally good values, but it could be possible to determine a best setting within this range depending on the input. Interdistance of lines, the curviness of the drawing, scale, intersections and crowdedness could play a role in determining these values. But in what way these factors influence the parameter values has to be determined. Another approach would be to set the parameter values depending on a difficulty level. This is even more complex when taking the input into account. Another approach would be to run the algorithm with different sets of parameter values and classify the resulting output to a certain difficulty level.

A digital variation of this puzzle could be studied. In a digital environment the interaction would be different. With a touch screen we could drag a line from one point to another. The line could then be matched with the link line which should make it easier to solve the puzzle, similar as a ruler could contribute in solving the puzzle in the physical variation. This could possibly allow for lower tolerances for the ambiguity parameter.

Further research could be done in the human experience of the puzzles. When does a point become confusing? When does an unsolved puzzle become easily recognizable? To answer these questions a user study should be done. For example results from different iterations over algorithmic or visual parameters could be presented to the participant, such as link length. Then we can evaluate whether the participant is able to recognize the object from the input and if so in which timeframe he or she is able to do so. The results from this user study could be used to specify the set of optimal values for the parameters.

References

[1] D. H. Douglas and T. K. Peucker, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature," *Cartographica: The International Journal for Geographic Information and Geovisualization*, vol. 10, pp. 112–122, 1973.

[2] J. Hershberger and J. Snoeyink, "Speeding up the douglas-peucker line-simplification algorithm," in *Proc. 5th Intl. Symp. on Spatial Data Handling*, pp. 134–143, 1992.

[3] U. Ramer, "An iterative procedure for the polygonal approximation of plane curves," *Computer Graphics and Image Processing*, vol. 1, no. 3, pp. 244 – 256, 1972.

[4] H. Imai and M. Iri, "Computational-geometric methods for polygonal approximations of a curve," *Comput. Vision Graph. Image Process.*, vol. 36, pp. 31–41, Nov. 1986.

[5] W. S. Chan and F. Chin, "Approximation of polygonal curves with minimum number of line segments," in *Proceedings of the Third International Symposium on Algorithms and Computation*, ISAAC '92, (London, UK, UK), pp. 378–387, Springer-Verlag, 1992.

[6] M. de Berg, M. van Kreveld, and Schirra, "S.: A new approach to subdivision simplification," in *In: Twelfth International Symposium on Computer Assisted Cartography*, pp. 79–88, 1995.

[7] M. Visvalingam and J. D. Whyatt, "Line generalisation by repeated elimination of points," *The Cartographic Journal*, no. 30, pp. 46–51, 1993.

[8] M. Burch, C. Vehlow, N. Konevtsova, and D. Weiskopf, "Evaluating partially drawn links for directed graph edges," in *Graph Drawing* (M. Kreveld and B. Speckmann, eds.), vol. 7034 of *Lecture Notes in Computer Science*, pp. 226–237, Springer Berlin Heidelberg, 2012.

[9] T. Bruckdorfer, S. Cornelsen, C. Gutwenger, M. Kaufmann, F. Montecchiani, M. Nllenburg, and A. Wolff, "Progress on partial edge drawings," in *Graph Drawing* (W. Didimo and M. Patrignani, eds.), vol. 7704 of *Lecture Notes in Computer Science*, pp. 67–78, Springer Berlin Heidelberg, 2013.

[10] M. Chrobak and D. Eppstein, "Planar orientations with low out-degree and compaction of adjacency matrices," *Theor. Comput. Sci.*, vol. 86, no. 2, pp. 243–266, 1991.

[11] Y. Asahiro, E. Miyano, H. Ono, and K. Zenmyo, "Graph orientation algorithms to minimize the maximum outdegree," in *Proceedings of the 12th Computing: The Australasian Theroy Symposium - Volume 51*, CATS '06, (Darlinghurst, Australia, Australia), pp. 11–20, Australian Computer Society, Inc., 2006.

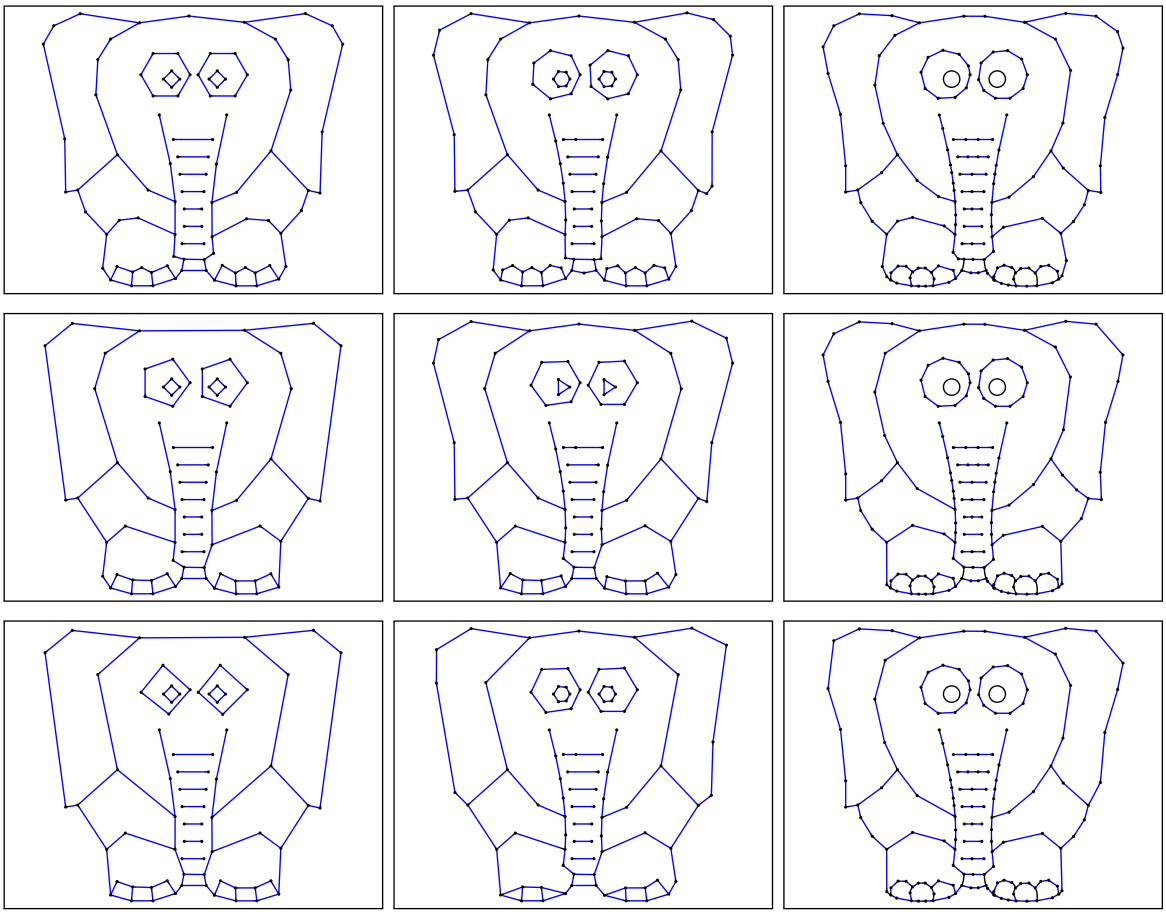
[12] H. Imai and M. Iri, "Polygonal approximations of a curve—formulations and algorithms," 1988.

[13] P. K. Agarwal and J. Erickson, "Geometric range searching and its relatives," in *Advances in Discrete and Computational Geometry*, 1999.

[14] B. Chazelle, L. J. Guibas, and D. T. Lee, "The power of geometric duality," *BIT*, vol. 25, pp. 76–90, June 1985.

[15] M. Overmars, *The Design of Dynamic Data Structures*. Lecture Notes in Computer Science, Springer, 1983.

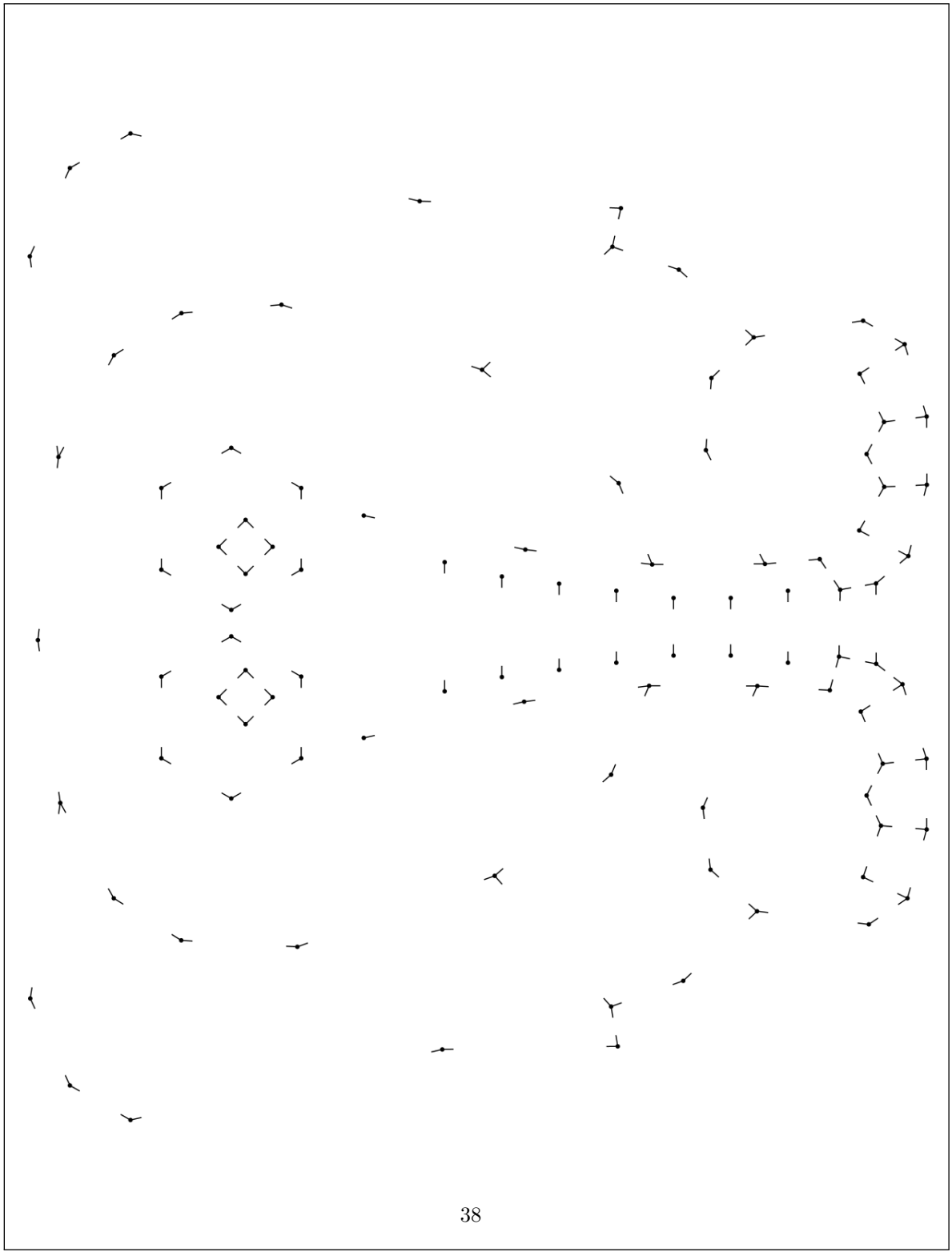
A Impact of input parameters



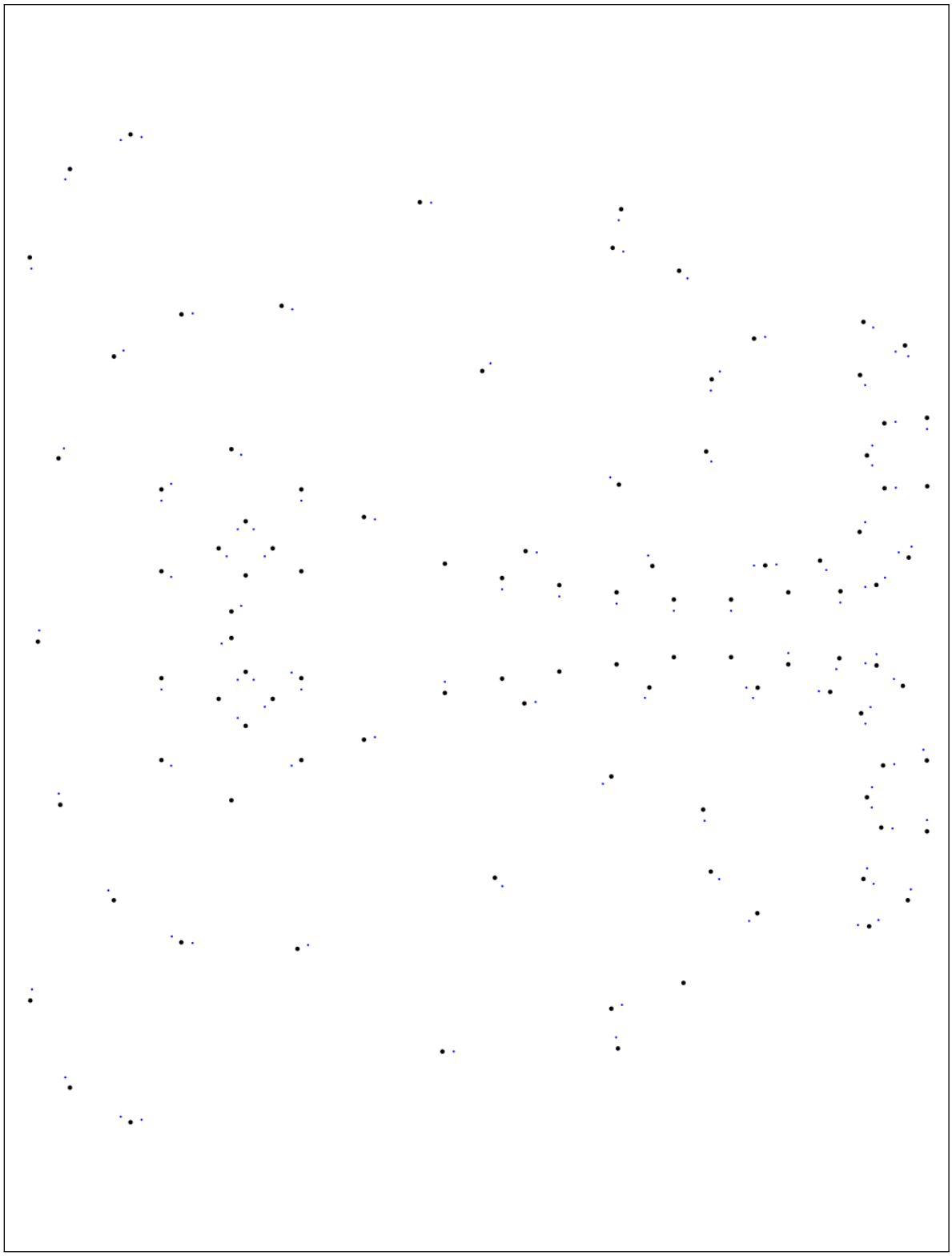
Left to right α and top to bottom ϵ

B Visual variations

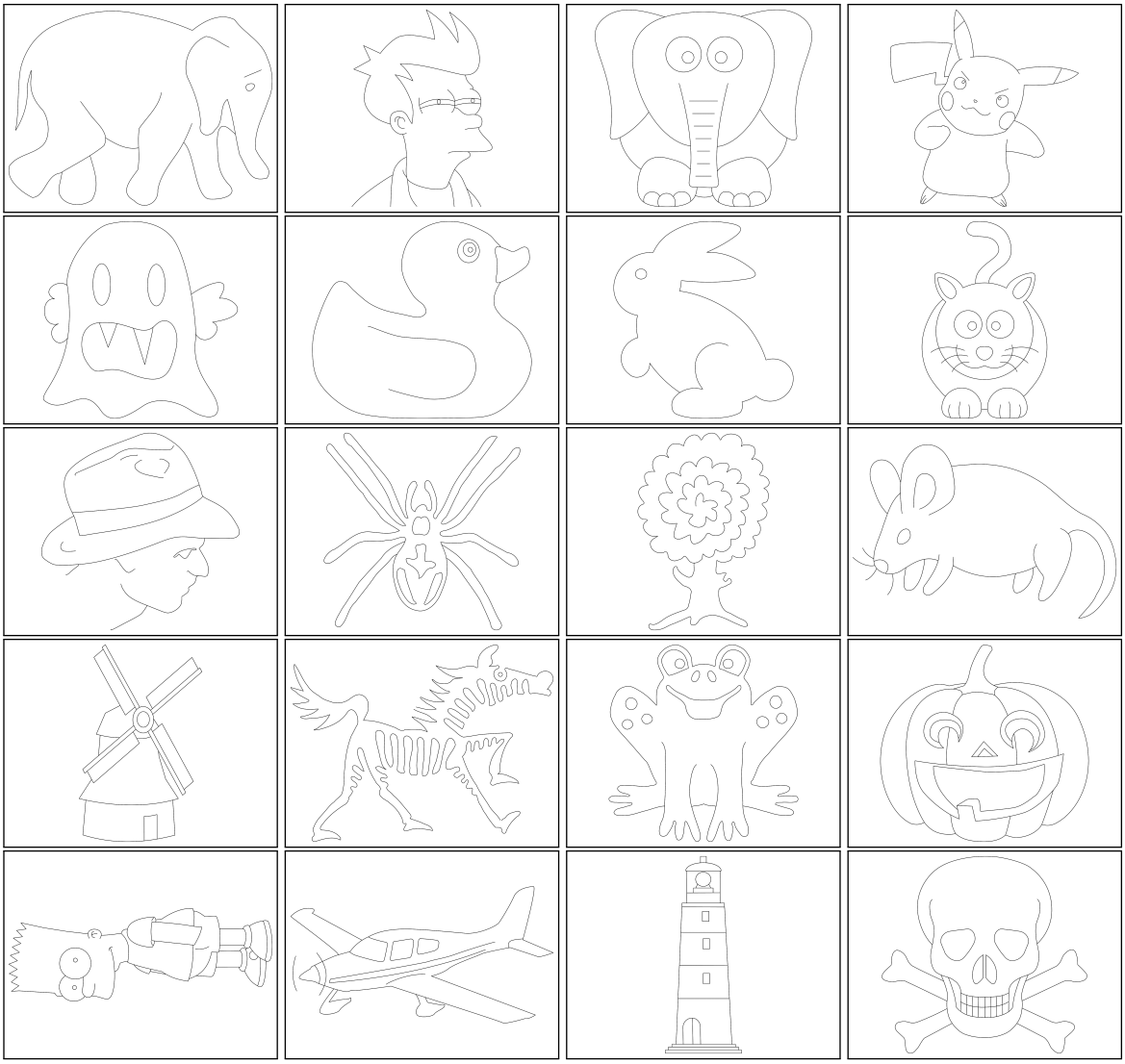
Symmetric



Pointwise



C Data set



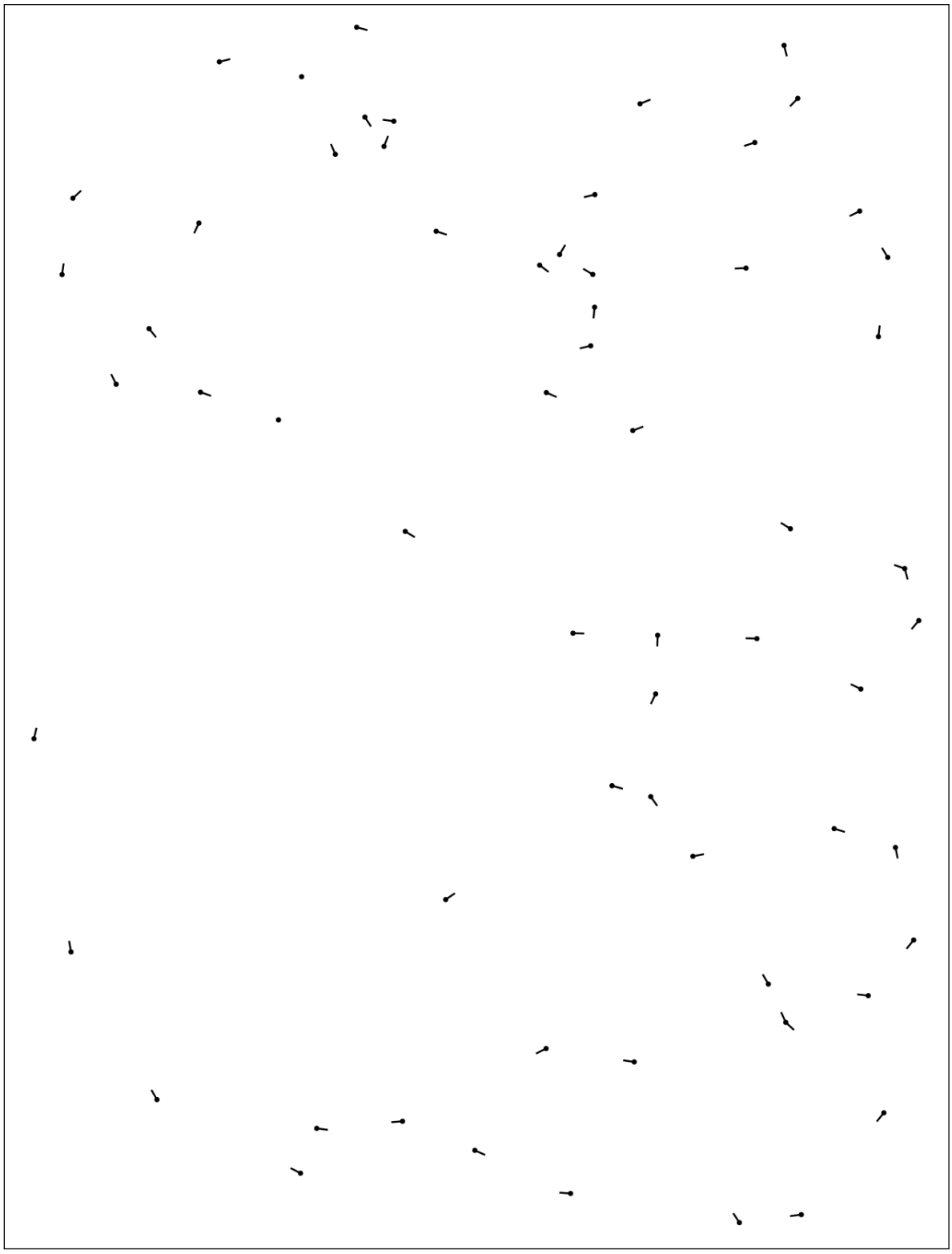
Input drawings

D Generated puzzles

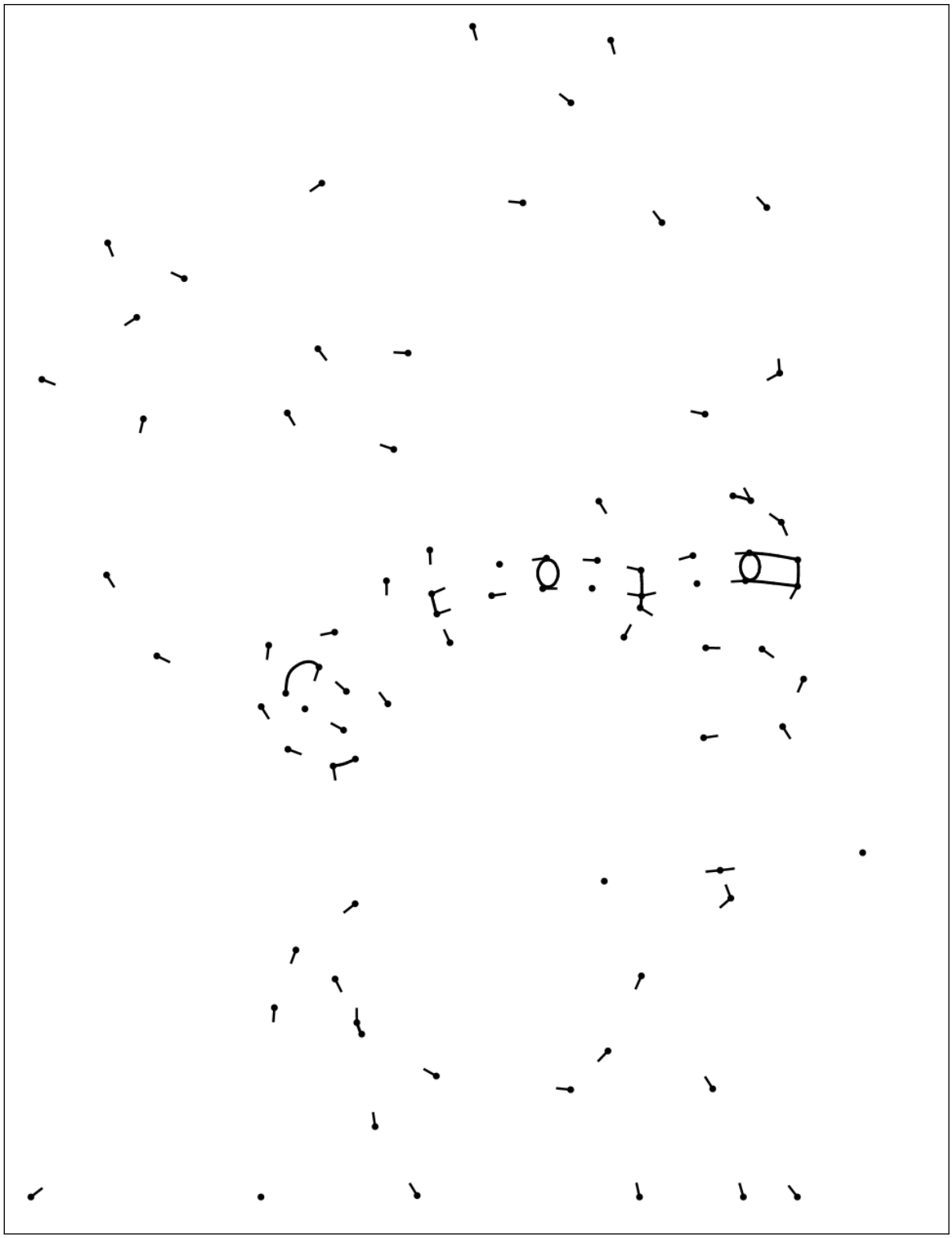
Input				Generated puzzle, default α(10) and ε(15)				
Drawing number	Points	Length	Degree	Puzzle points	Detail points	Degree	Length	Predrawn length
1	1798	10507.2892	3	65	0	2	10331.371	0
2	1576	8399.2462	4	81	137	2	8254.4534	451.7616
3	2398	13024.1583	3	103	0	2	12756.0977	0
4	1951	8671.5475	3	84	362	2	8465.613	298.8972
5	1599	9011.9212	3	59	0	2	8796.6859	0
6	1398	7468.4429	3	45	32	2	7328.0893	91.4347
7	1123	5956.1985	3	42	0	2	5776.6106	0
8	2155	10819.0164	4	113	24	2	10522.2396	84.8388
9	1733	8271.5718	3	61	17	2	8070.5268	40.7191
10	2788	14255.3803	3	108	8	2	13774.6797	44.7035
11	2305	9681.3258	3	123	0	2	8860.1345	0
12	1764	9207.452	3	74	15	2	9029.9206	74.7723
13	1191	9444.6653	3	82	61	2	9325.0937	371.9483
14	3733	18197.7939	2	174	24	2	17113.3195	78.5602
15	2664	13031.419	3	126	4	2	12495.7883	18.764
16	2852	14947.2466	3	111	19	2	14626.8052	102.4524
17	2072	10232.4094	3	108	248	2	9721.6736	558.8031
18	2048	11504.2221	4	82	8	2	11313.4308	24.1098
19	963	7807.7873	3	79	64	2	7653.1226	396.5878
20	2072	12291.4777	4	97	2	2	12037.4406	17.6676

Input properties and output at default values

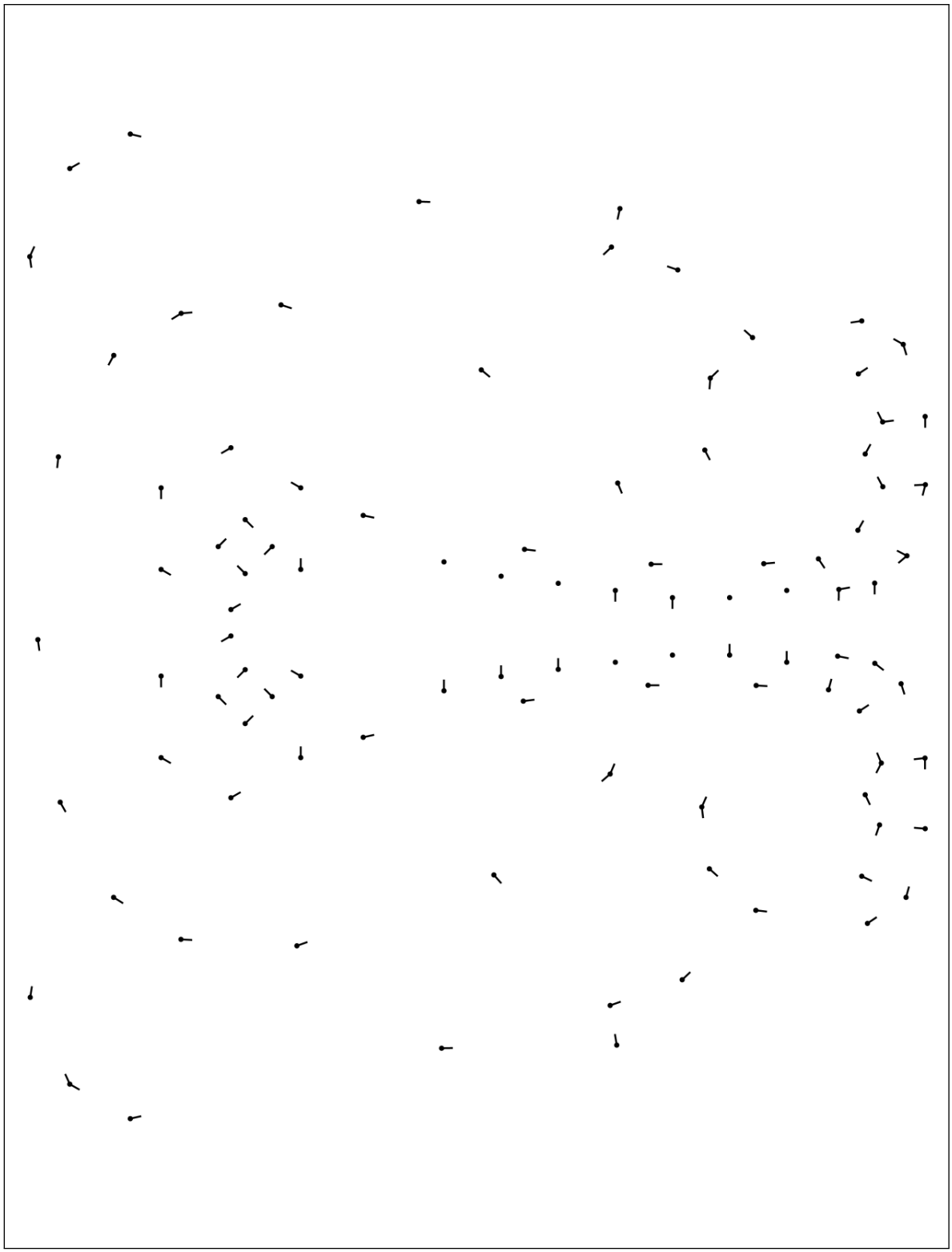
Puzzle 1, $\epsilon = 15$, $\alpha = 10$



Puzzle 2, $\epsilon = 15$, $\alpha = 15$

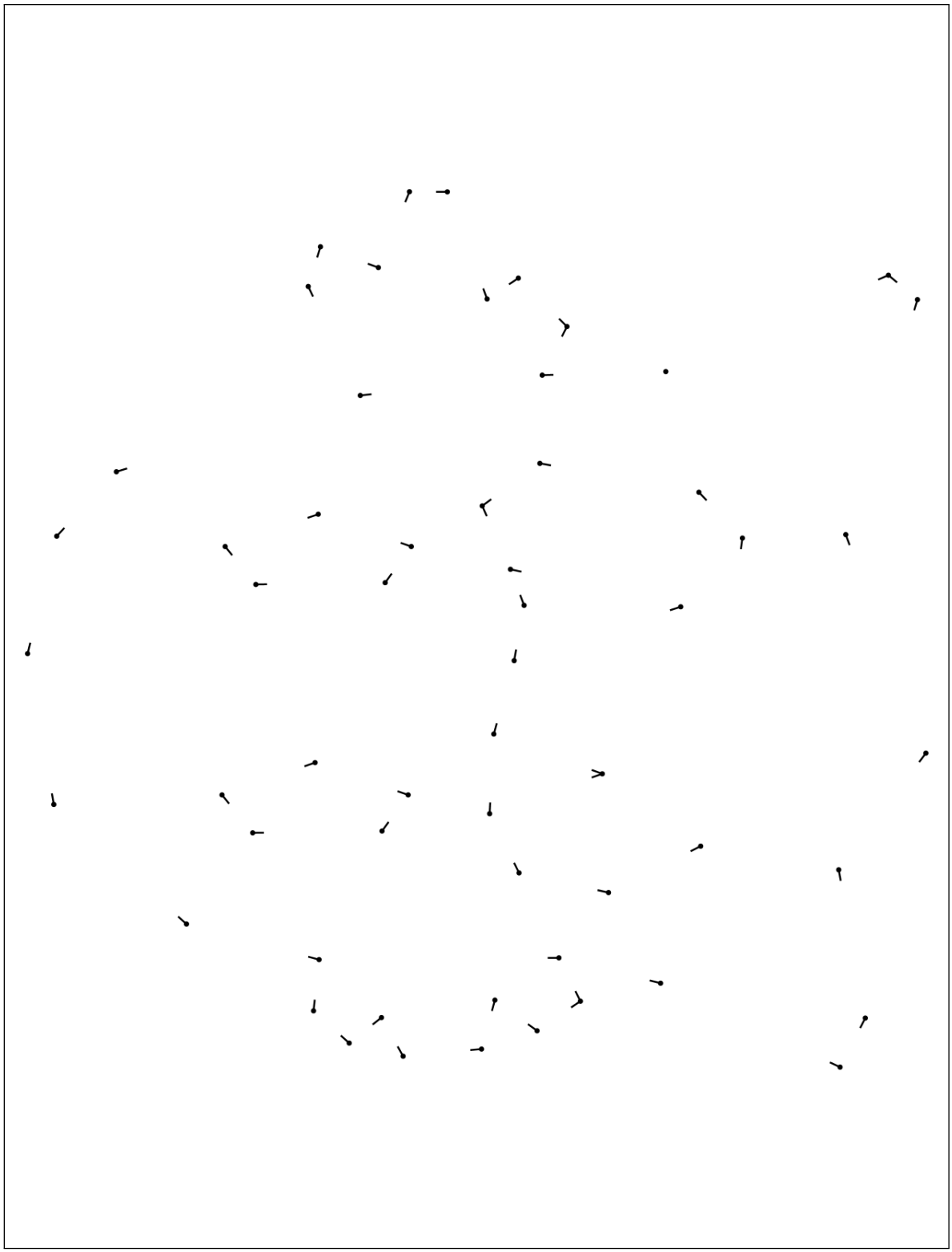


Puzzle 3, $\epsilon = 15$, $\alpha = 10$

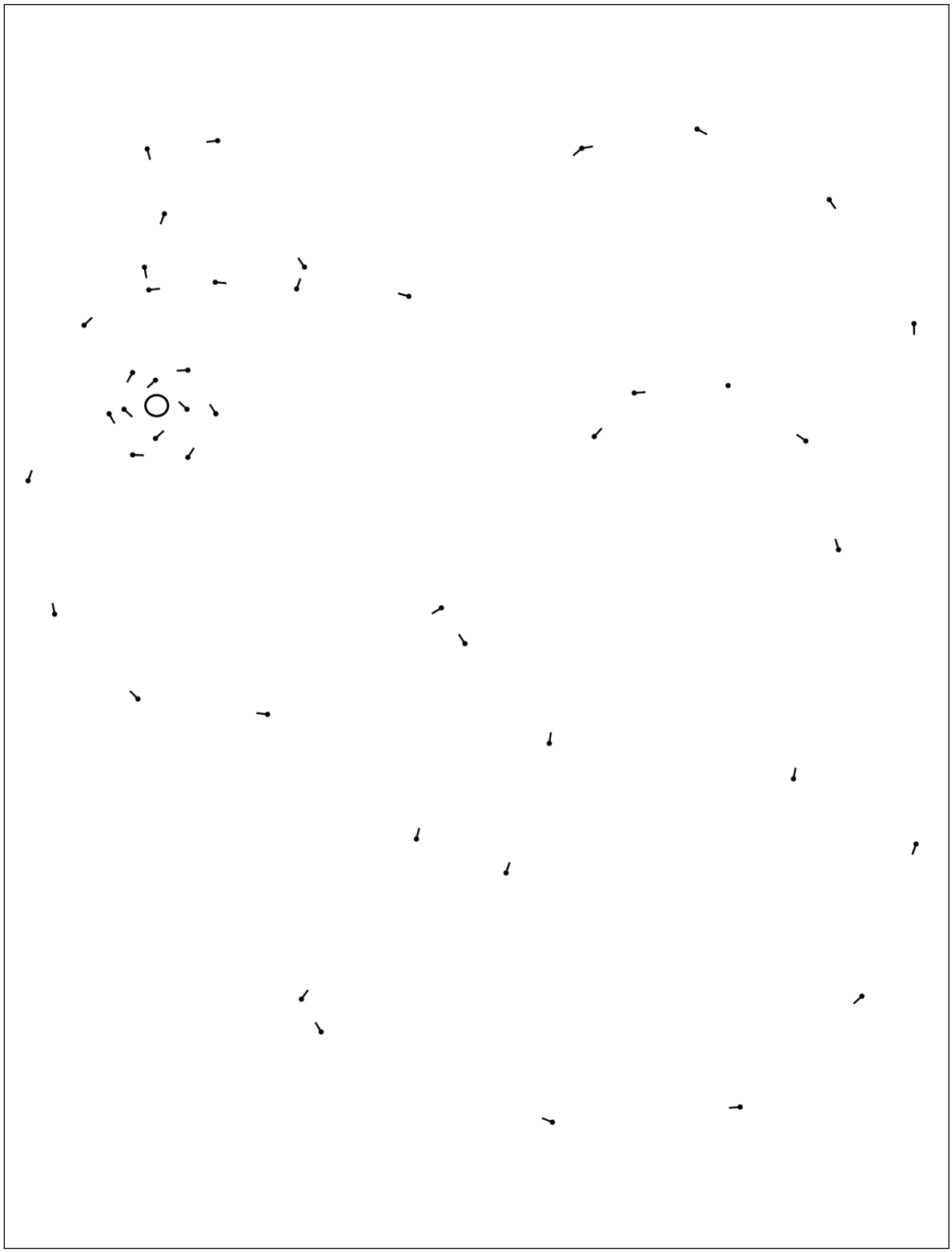


Puzzle 4, $\epsilon = 15$, $\alpha = 25$

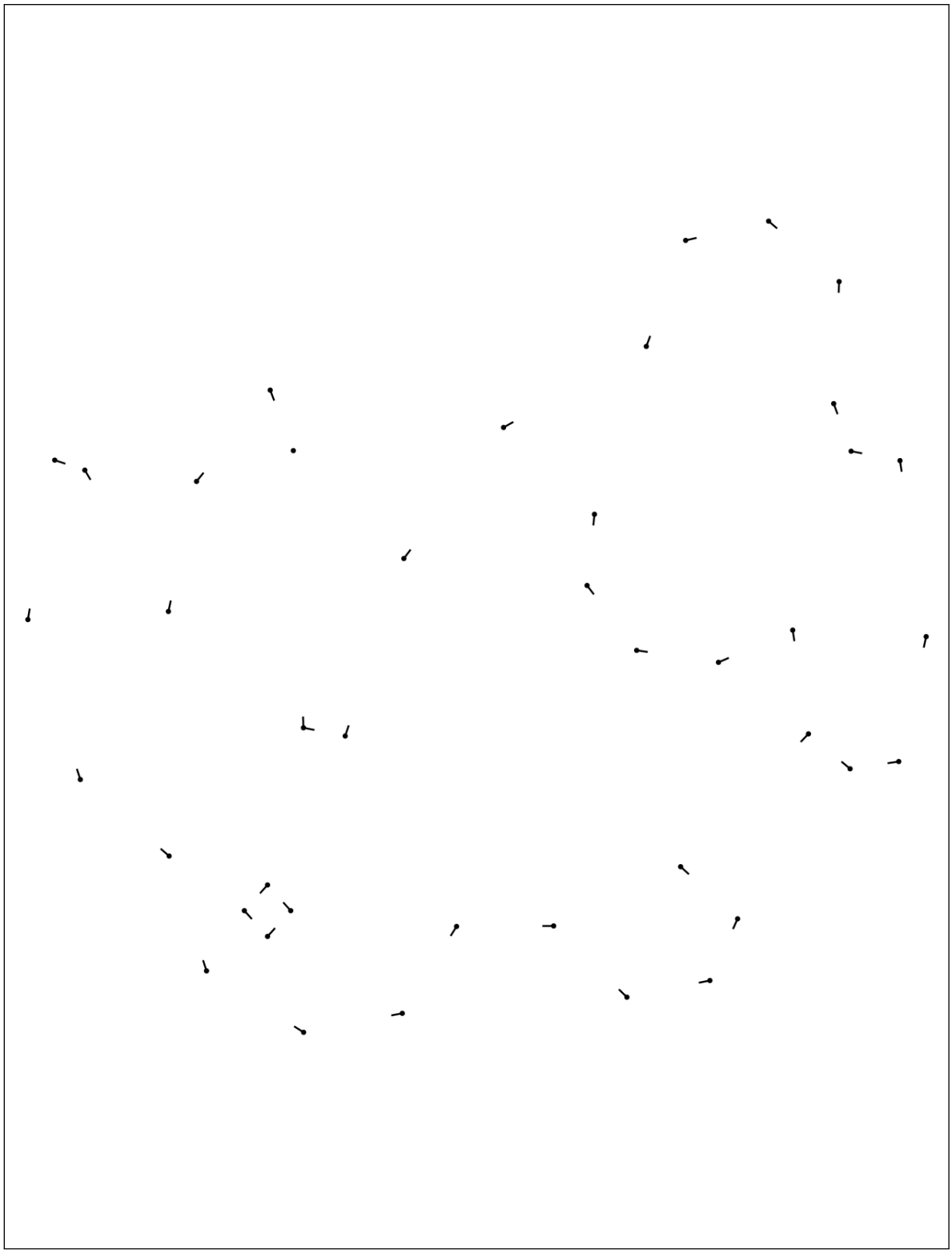
Puzzle 5, $\epsilon = 15$, $\alpha = 10$



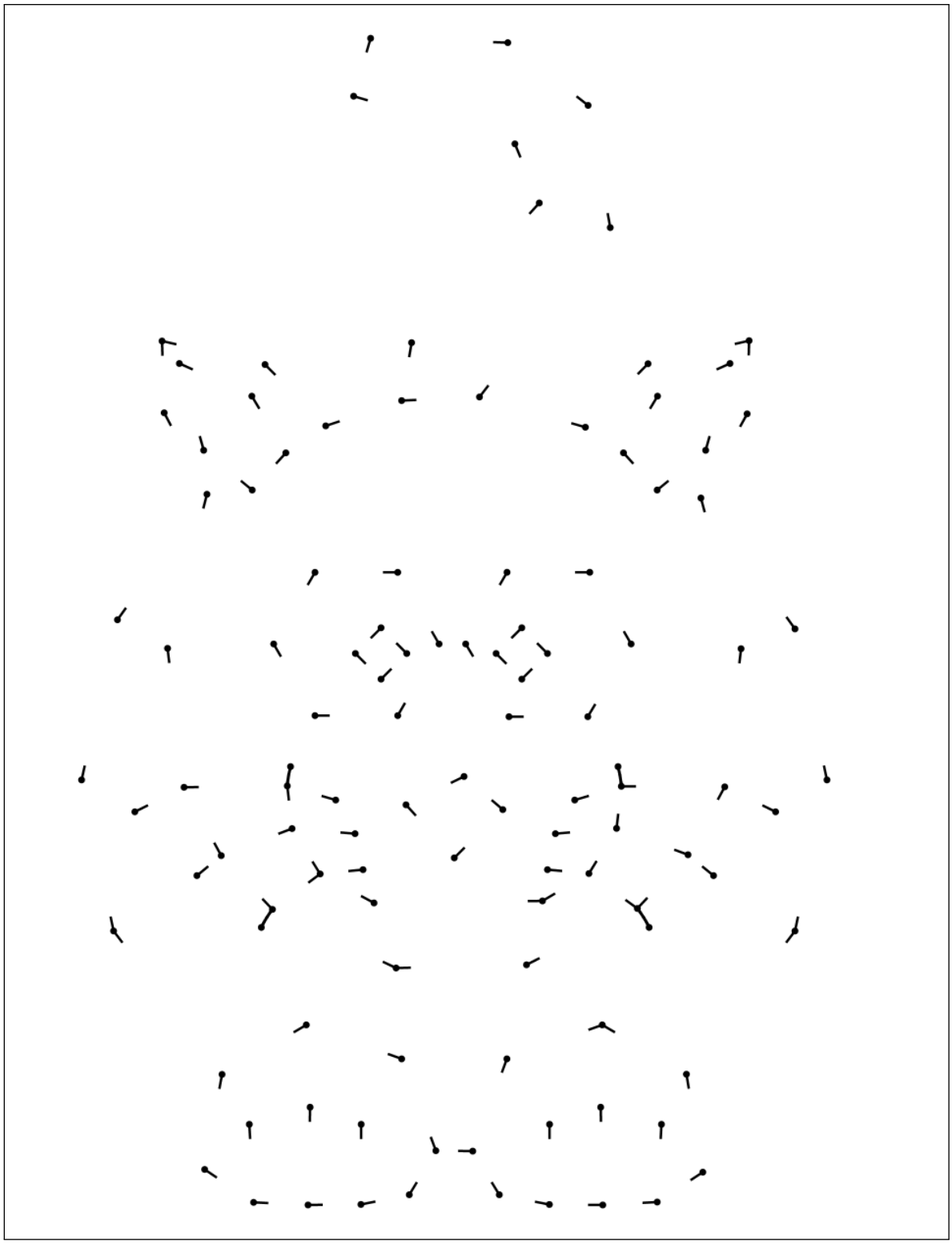
Puzzle 6, $\epsilon = 15$, $\alpha = 10$



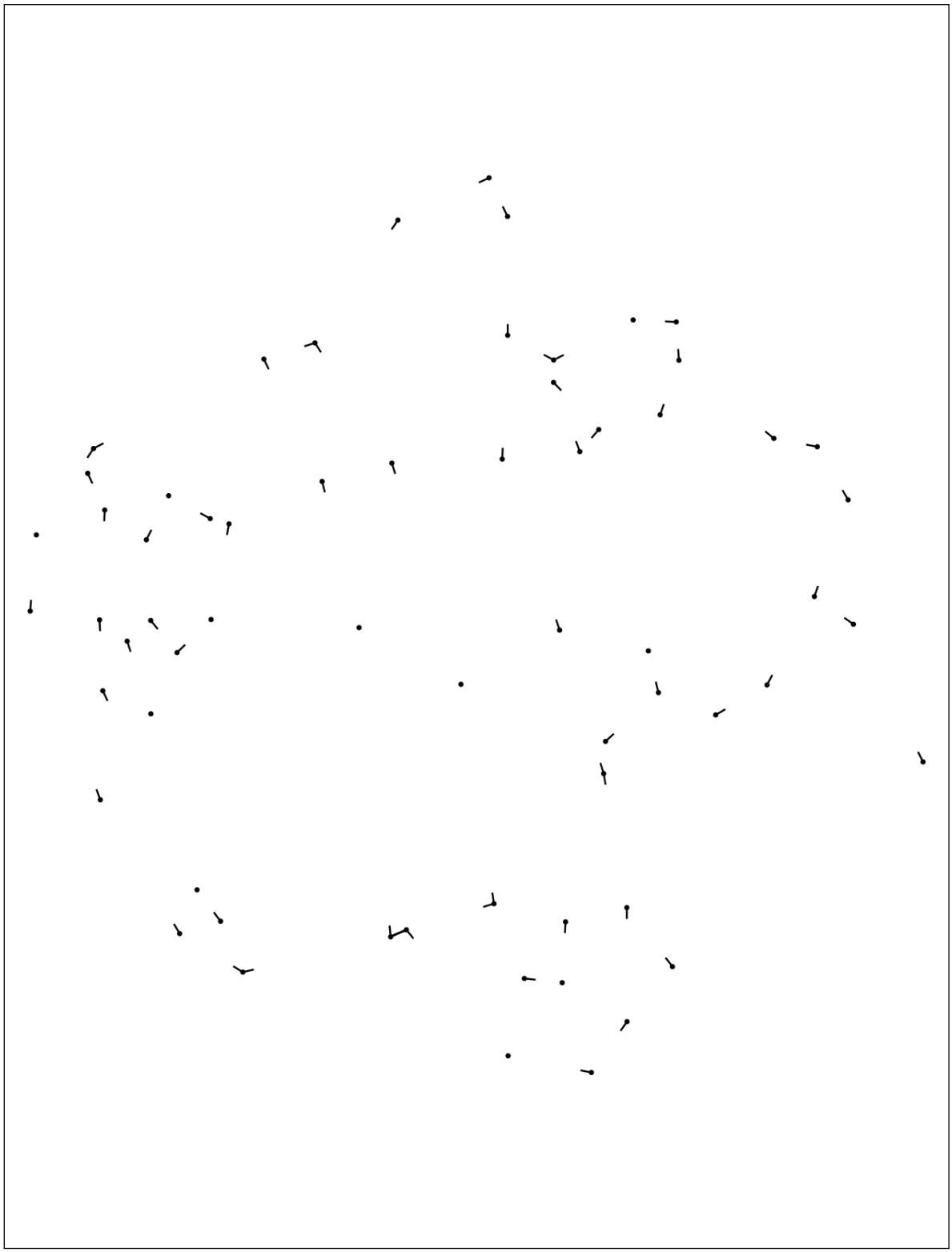
Puzzle 7, $\epsilon = 15$, $\alpha = 10$



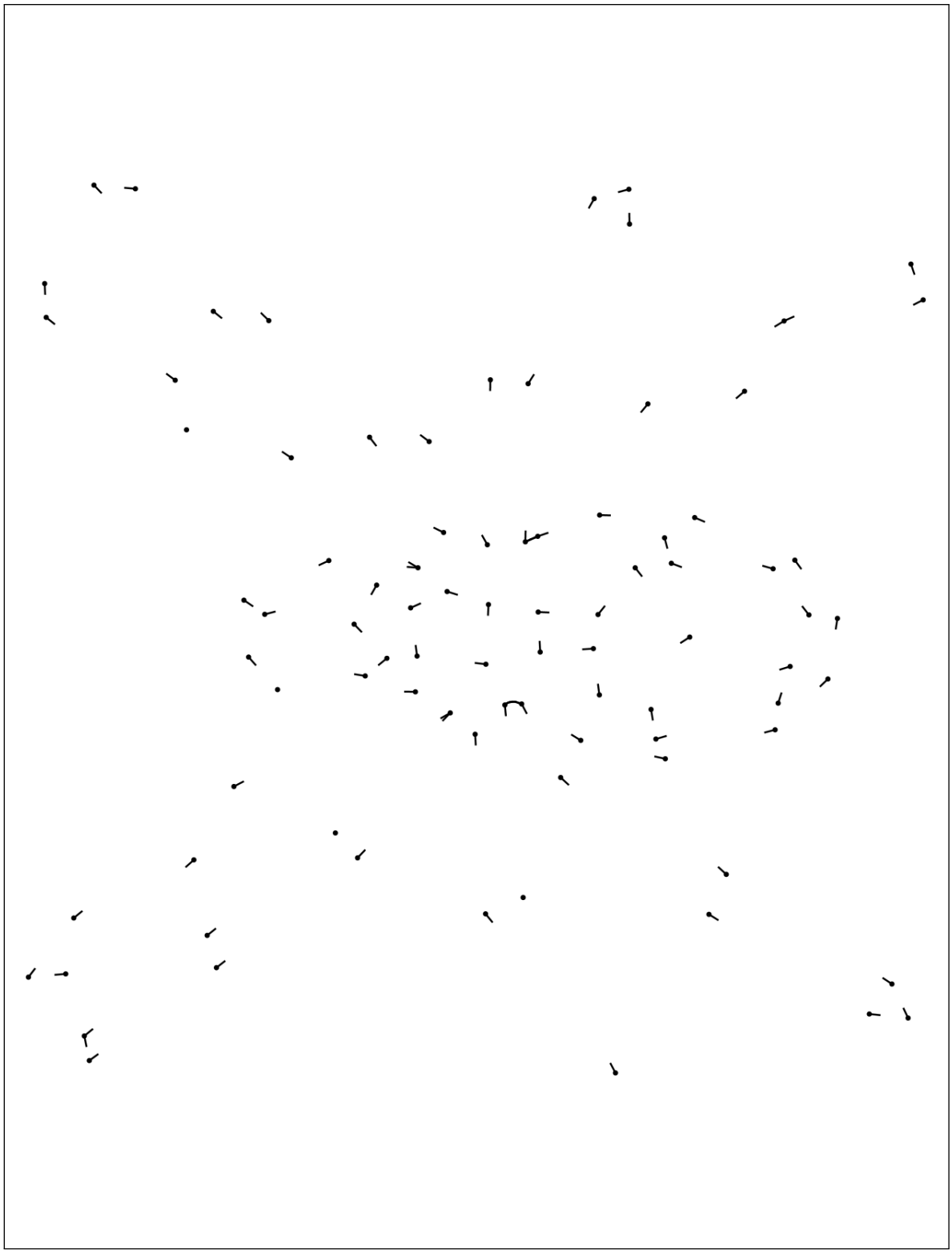
Puzzle 8, $\epsilon = 15$, $\alpha = 10$



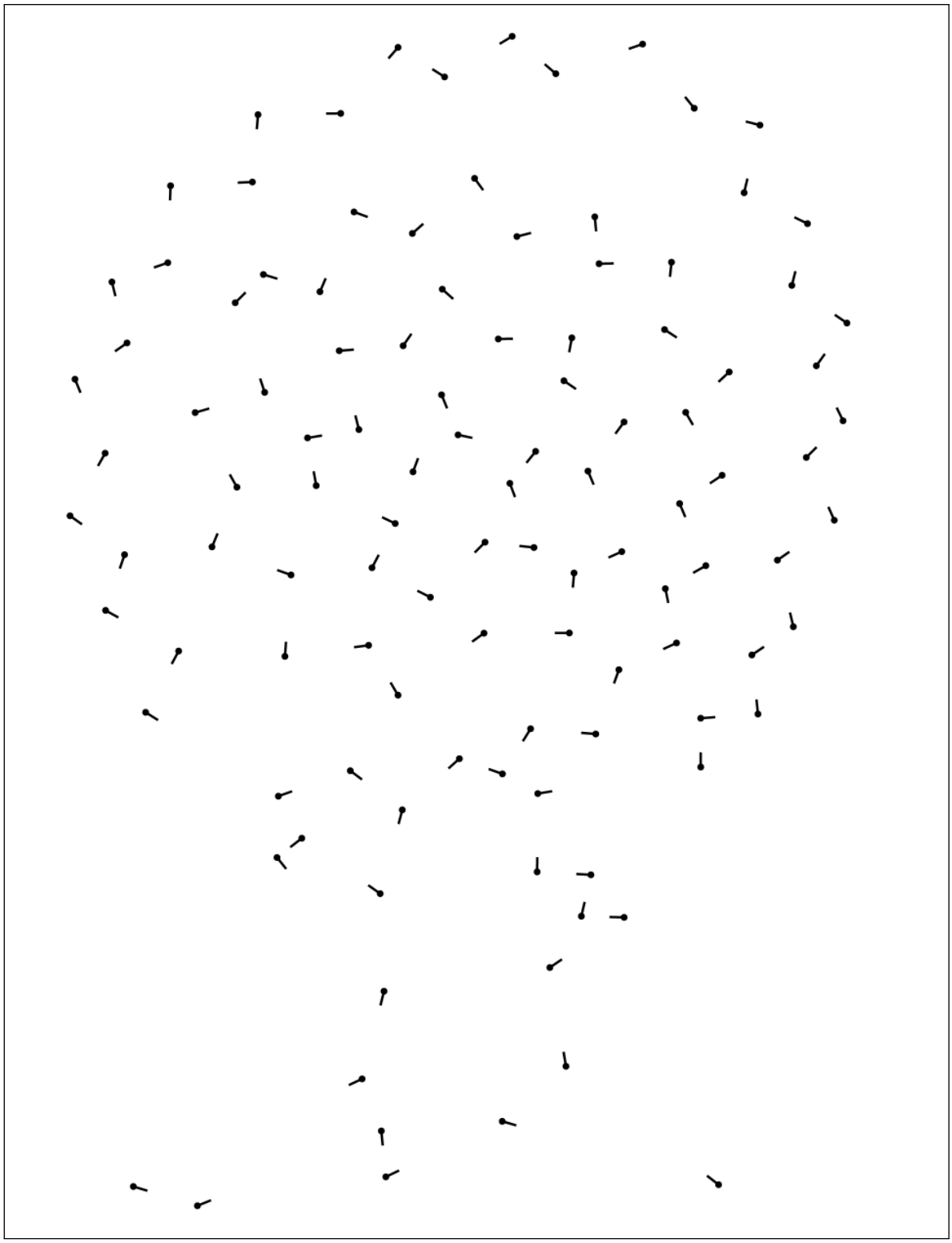
Puzzle 9, $\epsilon = 15$, $\alpha = 20$



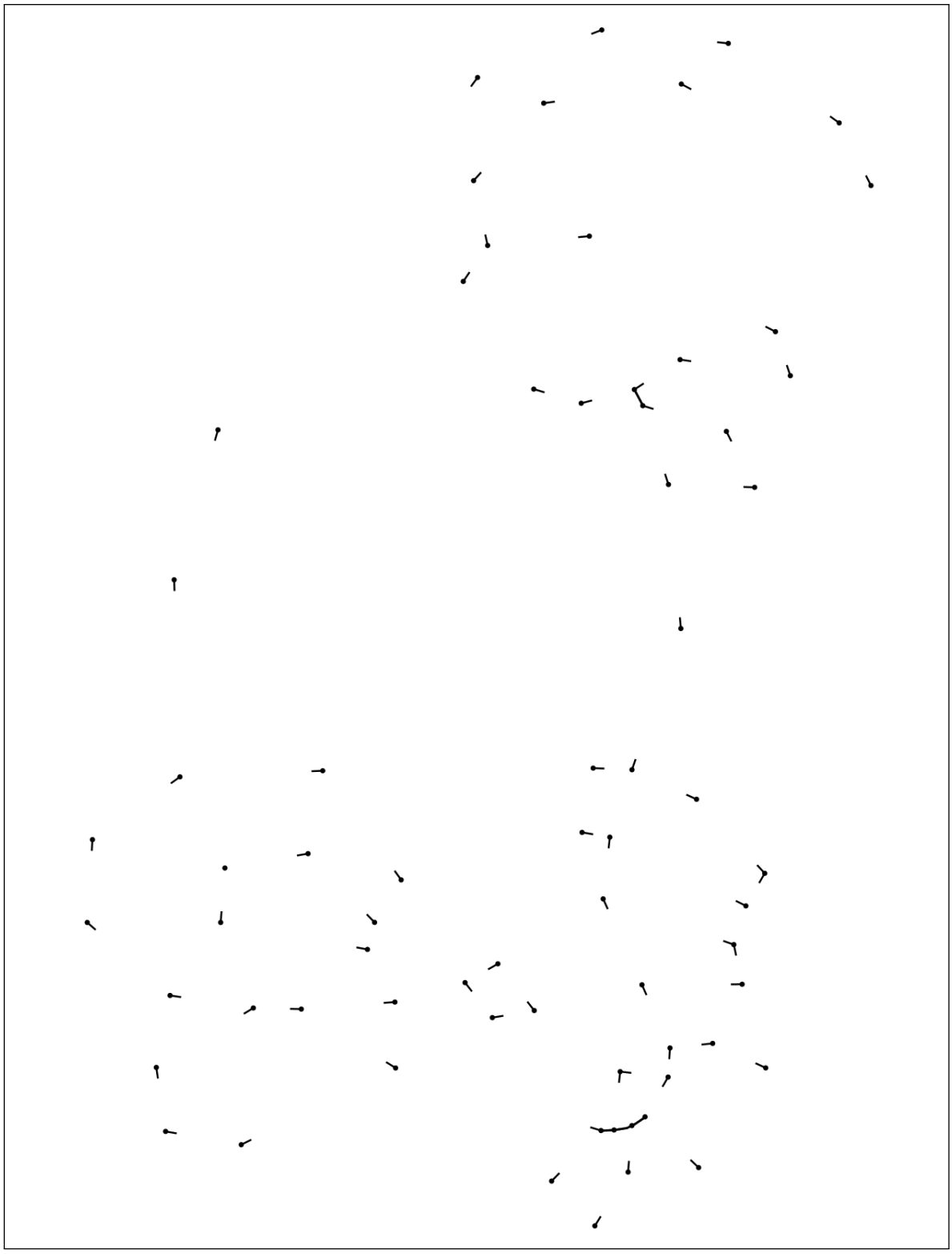
Puzzle 10, $\epsilon = 15$, $\alpha = 10$



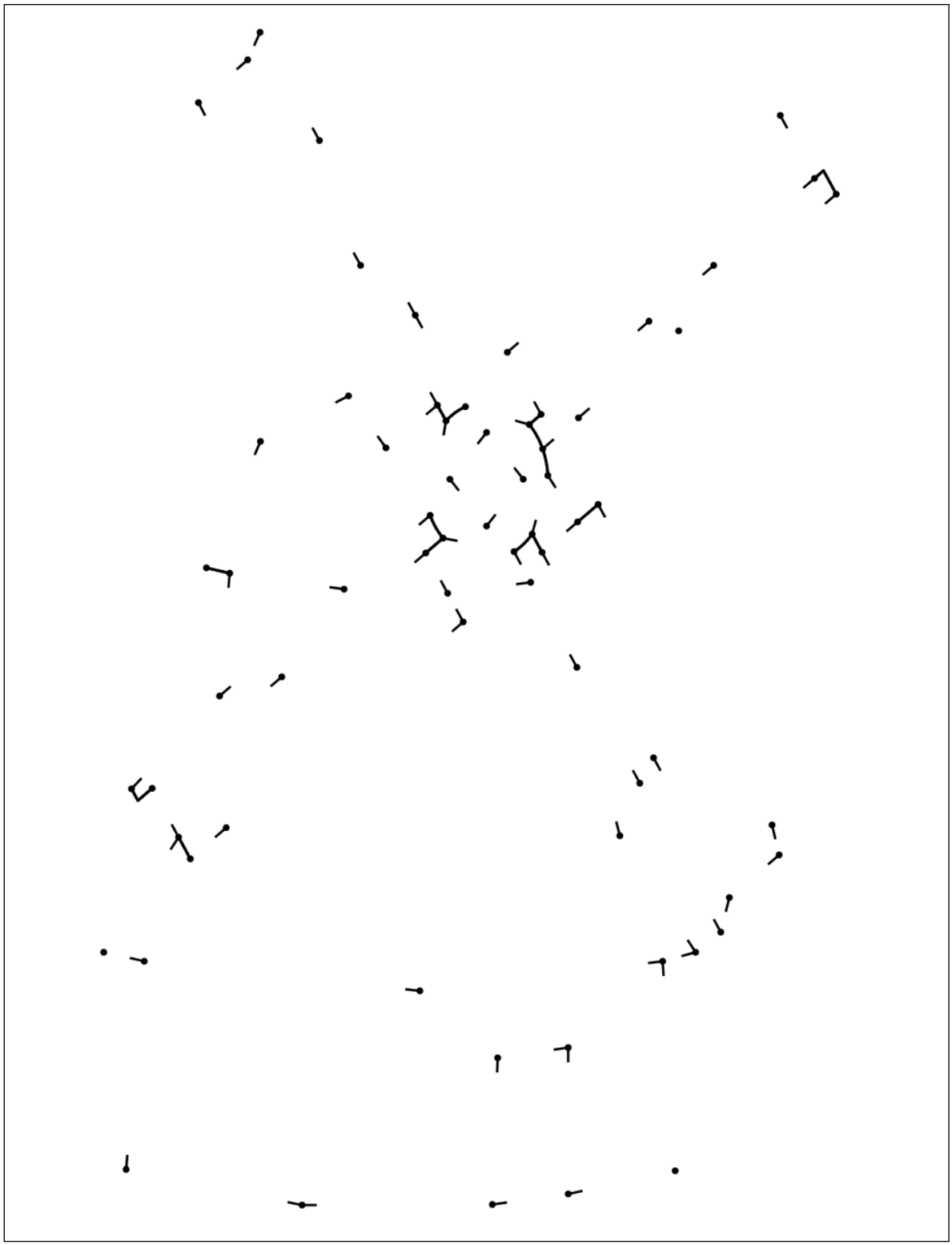
Puzzle 11, $\epsilon = 15$, $\alpha = 10$



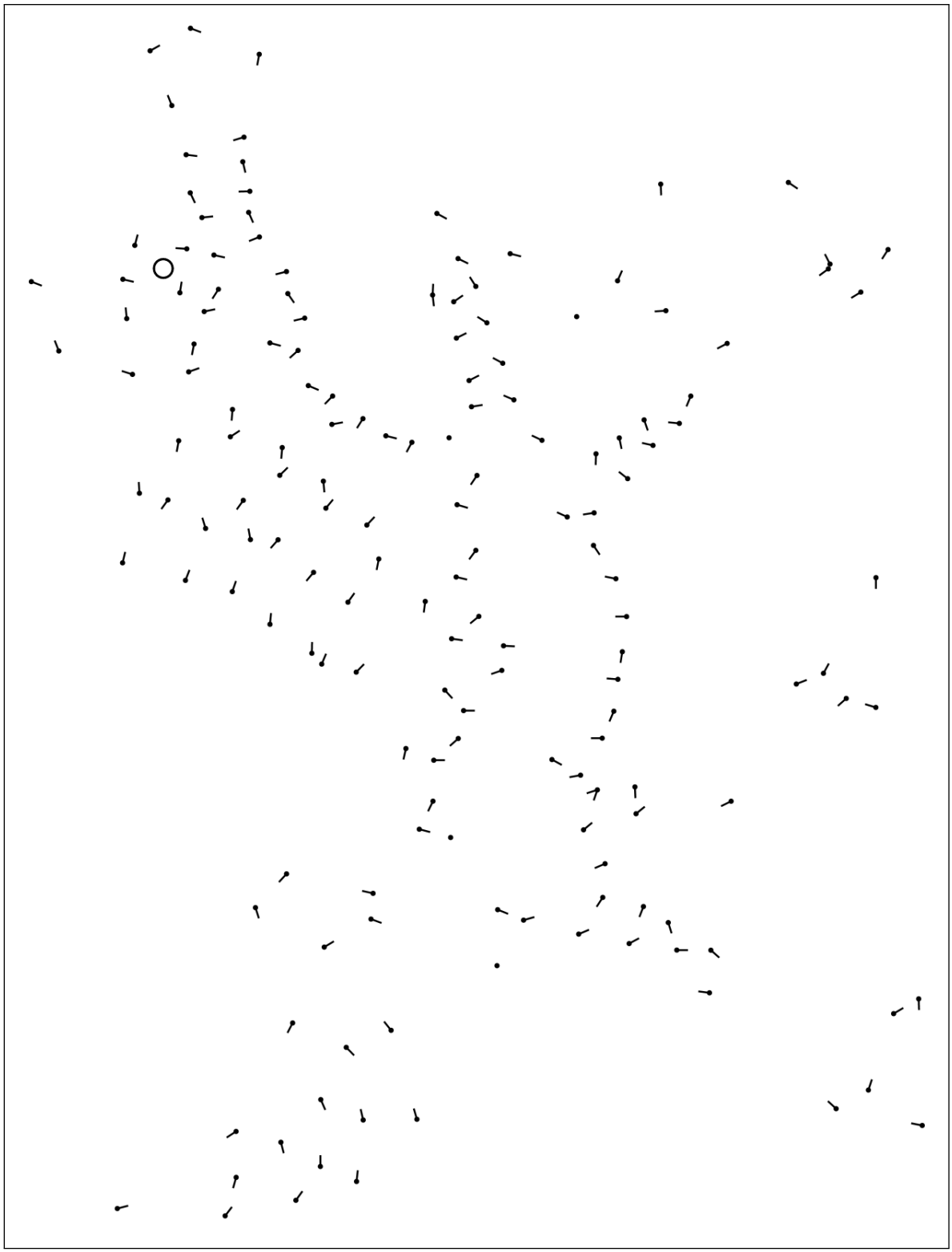
Puzzle 12, $\epsilon = 15$, $\alpha = 10$



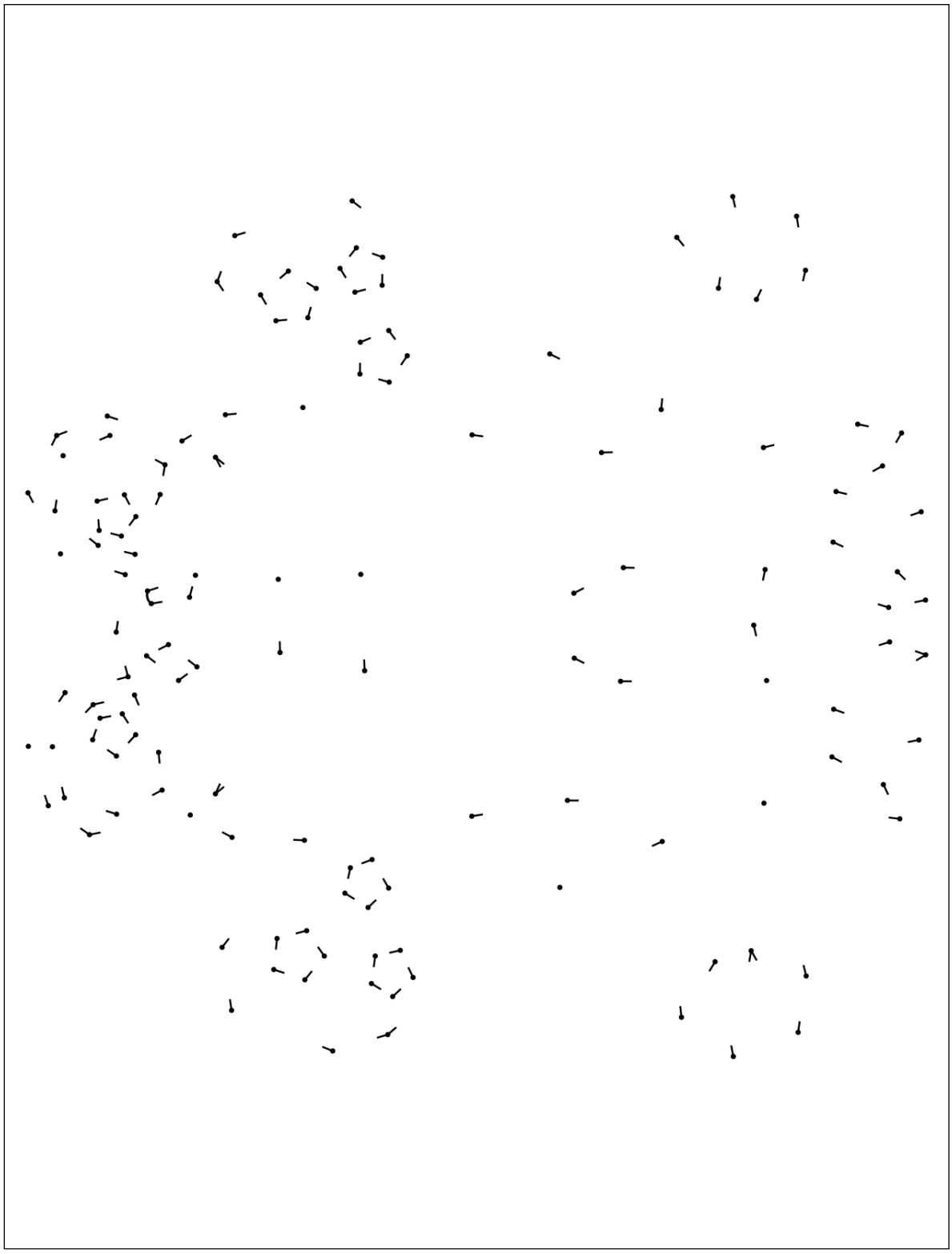
Puzzle 13, $\epsilon = 15$, $\alpha = 10$



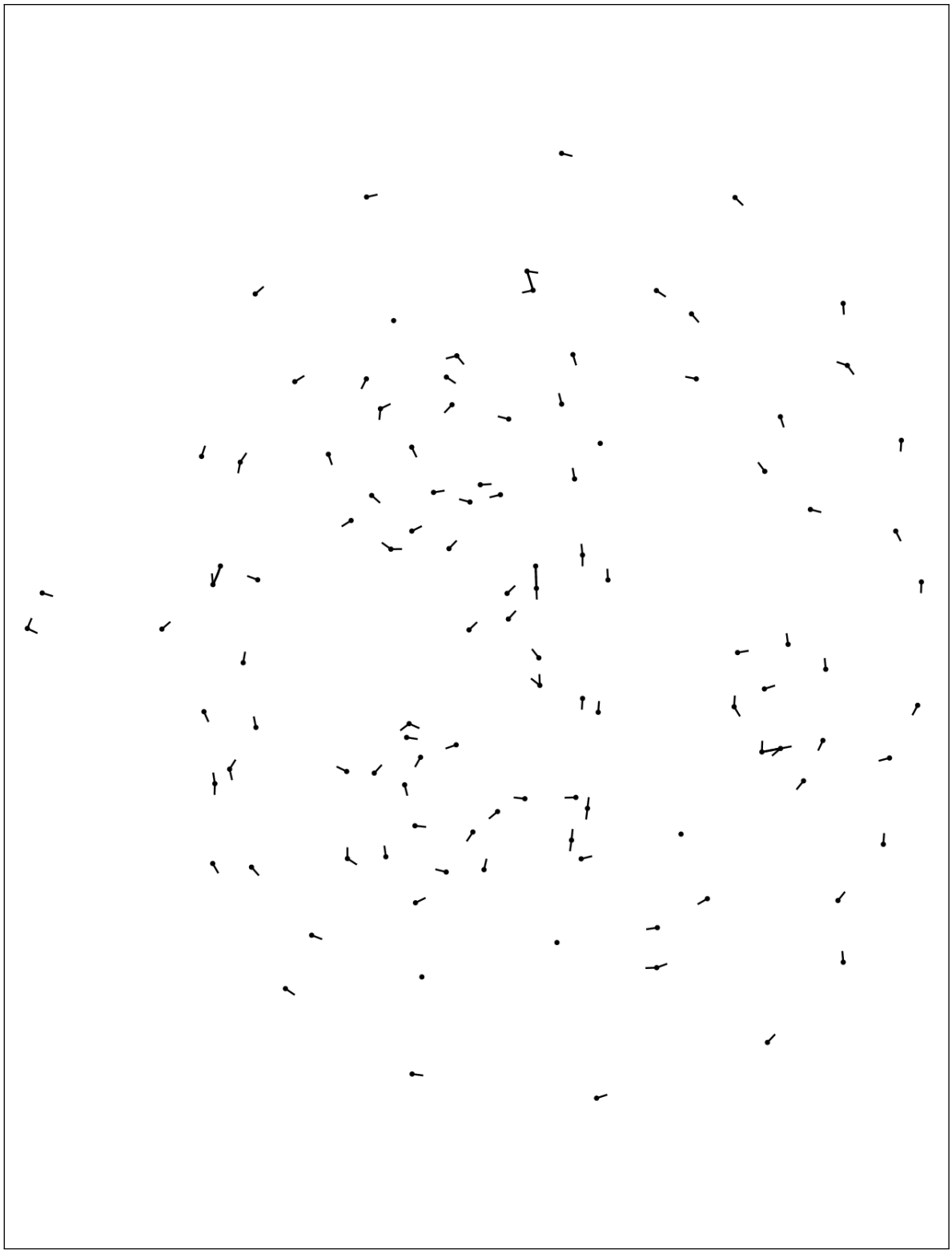
Puzzle 14, $\epsilon = 15$, $\alpha = 10$



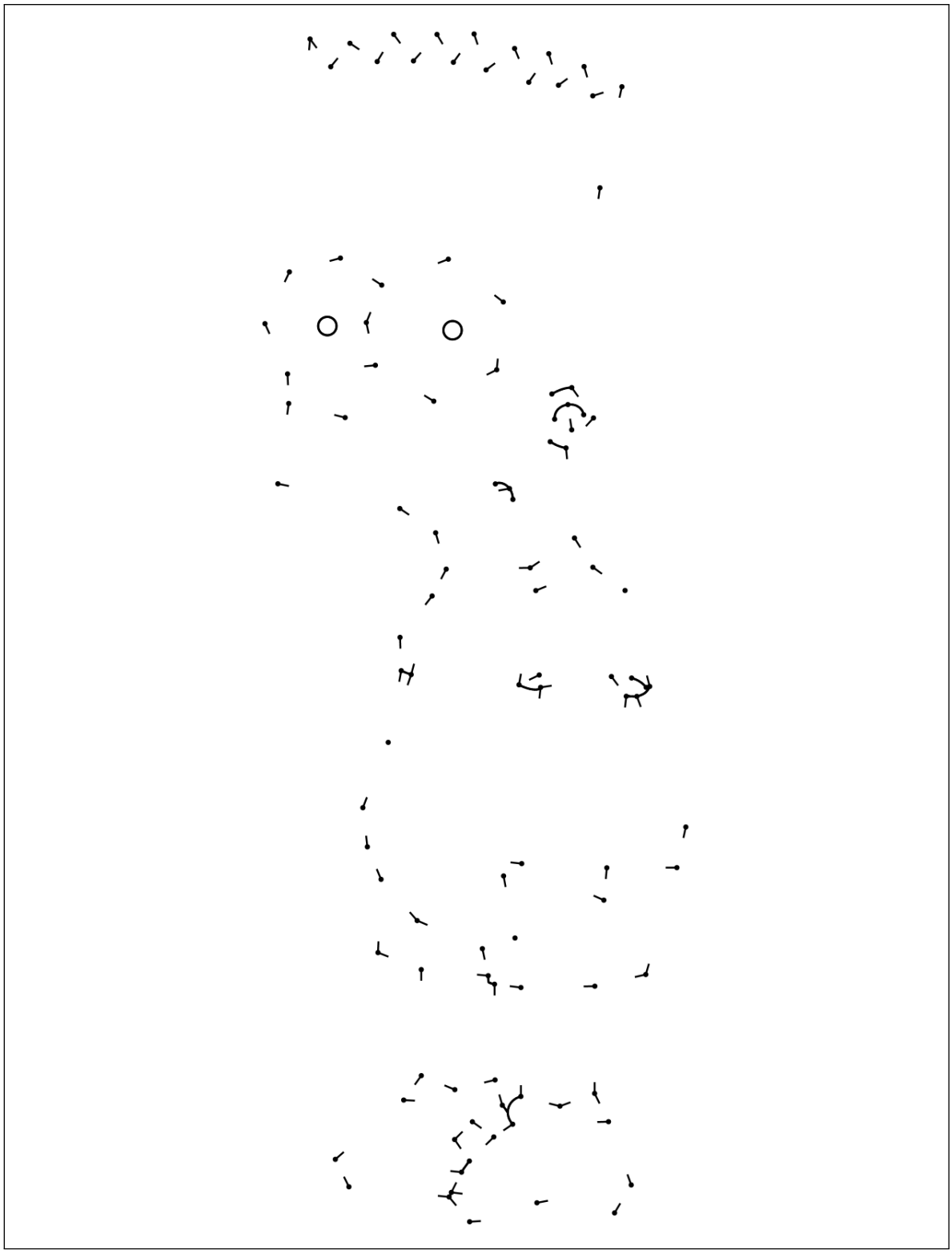
Puzzle 15, $\epsilon = 15$, $\alpha = 20$



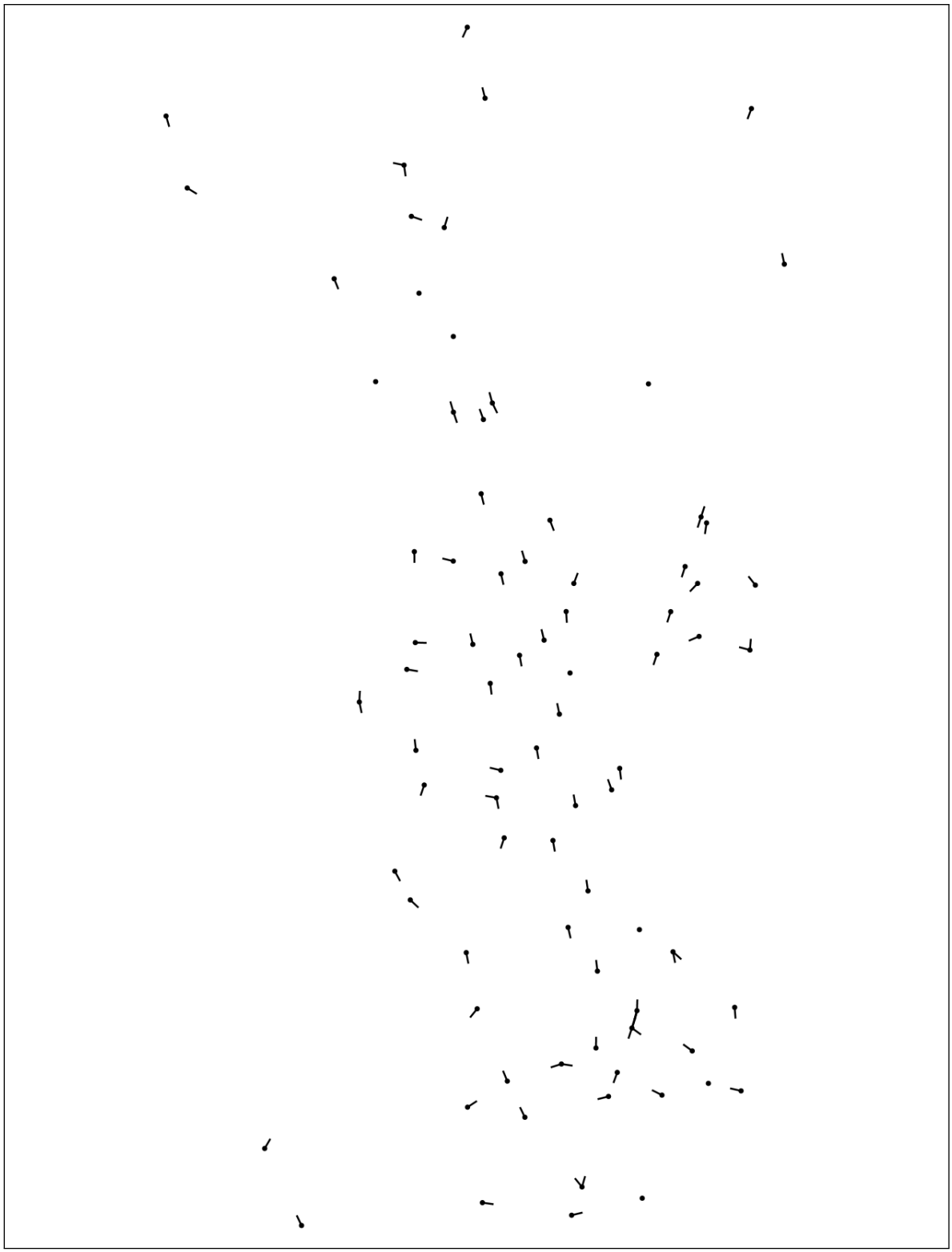
Puzzle 16, $\epsilon = 15$, $\alpha = 15$



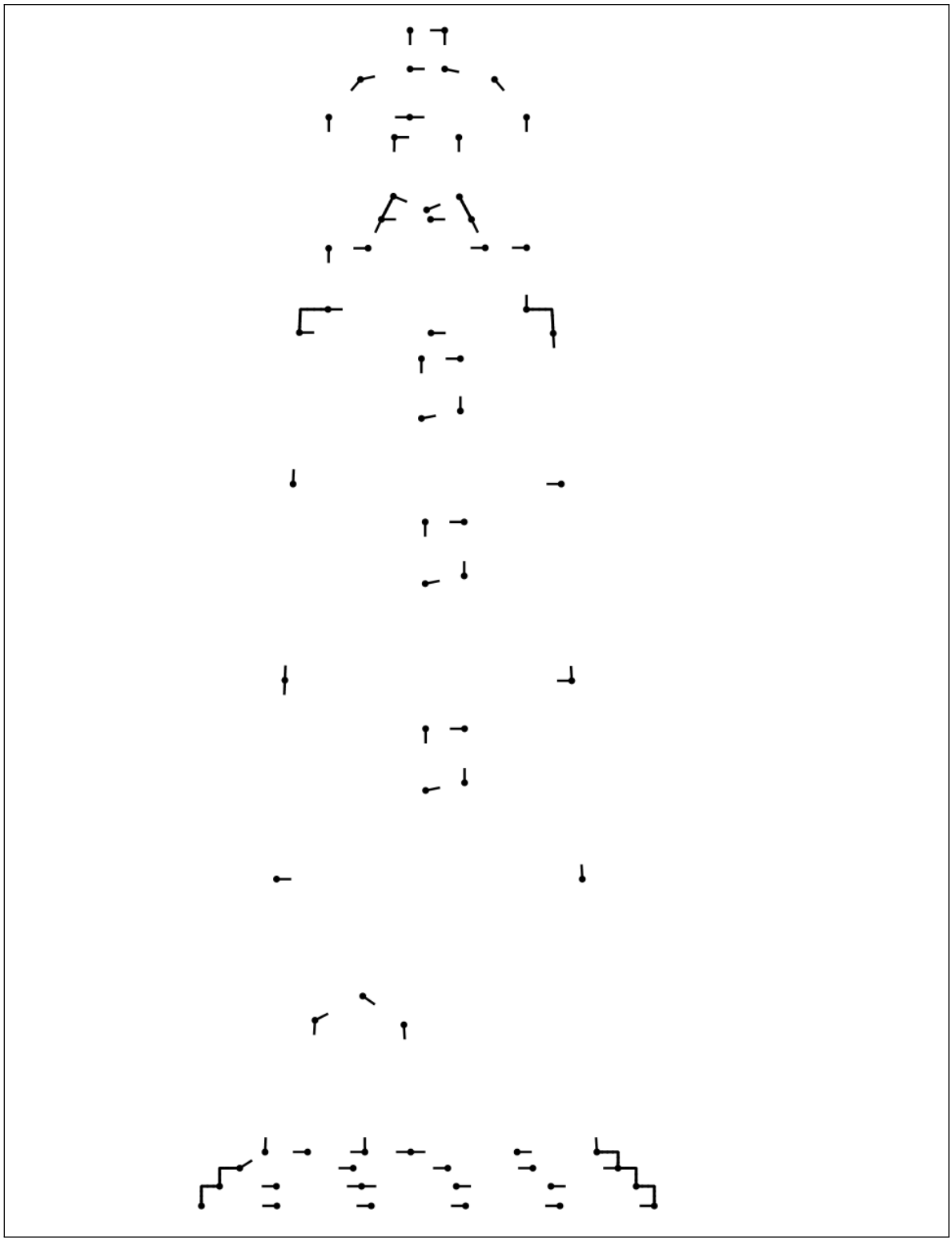
Puzzle 17, $\epsilon = 15$, $\alpha = 10$



Puzzle 18, $\epsilon = 15$, $\alpha = 15$



Puzzle 19, $\epsilon = 15$, $\alpha = 10$



Puzzle 20, $\epsilon = 15$, $\alpha = 10$