Graph Drawing Contest 2020

Kunhao ZHENG Yuyang ZHAO

January 2020

1 Problem Description

A *straight-line upward grid drawing* is a 2D drawing in which the following conditions are satisfied:
(1) For each edge, the target vertex should have a strictly higher y-coordinate than the source vertex.
(2) For each edge, neither the target vertex nor the source vertex could be on the line segment of another edge.
(3) The graph is embedded on the input grid: the x-coordinate and y-coordinate of the vertices must be integers on a grid of size [0..width]×[0..height].

Given the input graph in JSON format, which contains a list of vertices and a list of oriented edges, we try to firstly check the validity of a drawing and compute the number of drawing; and then construct a valid drawing by moving the nodes in the grid; finally minimize the number of crossing based on a valid drawing.

2 Algorithm

2.1 Checking the validity of a drawing

Among the three conditions, the most difficult one to verify is to check whether the target vertex or the source vertex are on the line segment of another edge, so we introduce the lemme below:
Lemme: P(x,y) is on the segment of endpoints $(x_1, y_1), (x_2, y_2)$ *(endpoints included), if and only if*

- $(x - x_1) \times (y - y_2) = (x - x_2) \times (y - y_1)$
- $(x - x_1) \times (x - x_2) \leq 0$
- $(y - y_1) \times (y - y_2) \leq 0$

And then we are supposed to calculate the number of crossing. We introduce the lemme below to judge whether the two edges have a crossing.
Lemme: the edge P_1P_2 *and the edge* Q_1Q_2 *have a crossing, if and only if*

- $(\vec{P_1P_2} \times \vec{P_1Q_1}) \cdot (\vec{P_1P_2} \times \vec{P_1Q_2}) \leq 0$
- $(\vec{Q_1Q_2} \times \vec{Q_1P_1}) \cdot (\vec{Q_1Q_2} \times \vec{Q_1P_2}) \leq 0$

Algorithm 1 Check the validity of a drawing

```
for each $e \in G$ do
  verify $y_{source}<y_{target}$;
  verify neither source vertex nor target vertex is on the line segment of another edge;
  verify the vertices are in the grid;
  if at least one condition is not satisfied then
    return false
  end if
end for
return true
```

Algorithm 2 Compute the number of crossing

```
$counter \Leftarrow 0$
for each $e_1 \in G$ do
  for each $e_2 \in G$ do
    if $e_1 = e_2$ then
      continue
    else
      if $e_1$ and $e_2$ have a crossing then
        $counter \Leftarrow counter + 1$
      end if
    end if
  end for
end for
return counter
```

2.2 Compute a valid drawing

To compute a valid drawing, firstly we compute a *topological ordering* of the graph with a DFS traversal, and then we decide the position of each node according to this ordering. Recall the definition of *topological ordering* as follows.
Definition (topological ordering):
Topological Sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge, the source vertex comes before the target vertex in the ordering.

After sorting the vertices in *topological order*, we are going to decide the position of each node according to this order. In this way, for each node, we place all of its predecessors before deciding its position so that the condition about the relationship of y-coordinates can be satisfied easily.

The principal idea is that for each node, we compute the maximum y-coordinate of its predecessors, note y_{max} and we start to try to place the node on the position $[0, y_{max} + 1]$, and to check the validity of the present drawing. If not, we place the node on $[1, y_{max} + 1]$ and repeat the process until find a valid position. (If all of points with $y_{max} + 1$ can't satisfy the conditions, we start to try $y_{max} + 2$).

Algorithm 3 Topological Order

```
Input:
    Directed Acyclic Graph G=(V,E);
Output:
    List of vertices in topological ordering;
1: Create a temporary stack and a boolean array named as visited[ ].
2: Mark all the vertices as not visited i.e. initialize visited[ ] with 'false' value.
3: Call the recursive helper function topologicalSortUtil() to store Topological Sort start-
   ing from all vertices one by one.
4: Define the function topologicalSortUtil():
       • Mark the current node as visited.
       • Recall for all the successors of this vertex.
       • Push current vertex to stack which stores result.
5: Print contents of stack.
```

Algorithm 4 Compute a valid drawing

```
Input:
  A given drawing G in JSON format;
  List of vertices in topological ordering L;
Output:
  A valid drawing $G'$;
  if G is a valid drawing then
    return G
  end if
  Creat a blank drawing $G'$
  for Vertices $v \in G$ do
    $y_{max} \Leftarrow$ maximum y-coordinate among predecessors of $v$
    $x \Leftarrow 0$
    $y \Leftarrow y_{max} + 1$
    while the present drawing isn't valid do
      if $x < Width - 1$ then
        $x \Leftarrow x + 1$
      else
        $x \Leftarrow 0$
        $y \Leftarrow y + 1$
      end if
      place $v$ on the position $[x, y]$
    end while
  end for
  return $G'$
```

2.3 Minimize the number of crossing

In order to minimize the number of crossing, we have two different method: force-directed layouts and local search heuristic.

Method 1: Force-directed layouts
In this model, vertices can be seen as particles of a physical system that evolves under the action of forces exerced on the vertices. We define three kinds of forces as follows:

- For any pair of vertices (u, v), there exists a repulsive force.

$$F = \frac{-CK^2(\mathbf{x}(v) - \mathbf{x}(u))}{[\|\mathbf{x}(v) - \mathbf{x}(u)\|]^2}$$

- Between the adjacent vertices (u, v), there exists an attractive force.

$$F = \frac{[\|\mathbf{x}(v) - \mathbf{x}(u)\|]}{K}(\mathbf{x}(v) - \mathbf{x}(u))$$

- For any pair of edges of the same source vertex $(\vec{AB}, \vec{AC})$,with an intersection angle θ, there exists a repulsive force in the horizontal direction.

$$F = (\alpha(\tan^{-1}(\frac{[\|\vec{AB}\|]}{\beta}) + \tan^{-1}(\frac{[\|\vec{AC}\|]}{\beta})) + \gamma\cot(\frac{\theta}{2}))$$

In order to remain a valid drawing, after determining the new position of the present vertex. If the new position is higher (have a bigger y-coordinate) than the old one, we augment all of vertices higher than it in the old drawing by the same length. If the new position is lower (have a smaller y-coordinate) than the old one, we diminish all of vertices lower than it in the old drawing by the same length. Then we check the validity of the present drawing, if it is valid, we remain this change.

Method 2: Local search heuristic
Local search is a heuristic method for solving computationally hard optimization problems. Local search algorithms move from solution to solution in the space of candidate solutions (the search space) by applying local changes, until a solution deemed optimal is found or a time bound is elapsed.

In this problem, each time in the iteration we find the edge who has the biggest crossing numbers by using a priority queue. Then we do a local research by moving the source vertex and the target vertex of this edge, to determine the best position (diminish largely the number of crossing). In each iteration we will reduce both the search range and the search step by half. Thus we need two constant described the initial search range and the initial step length in the pseudo-code C_{step} and C_{range}. For some large inputs, we set $C_{step} = 10$ and $C_{range} = 5$. On the contrary, if the size of the graph is small, we set $C_{step} = height$ and $C_{range} = 1$ by doing a global research to find the best new position. Finally we do the iteration for the new drawing.

Algorithm 5 Local research heuristic

```
Input:
  Height, Weight: criterion of type of research;
  A 2D valid drawing G of size $G_{height} \times G_{width}$;
Output:
  A valid drawing $G'$;
  Find the edge which most contributes to the number of crossing
  $counter \Leftarrow$ the number of crossing of $G$
  if $G_{height}$ <Height and $G_{width} < Width$ then
    $step \Leftarrow Height/C_{step}$,
    $Xrange = Width, Yrange = Height$,
    $(Xnew, Ynew) = (Width/C_{range}, Height/C_{range})$
    while $step > 0$ do
      Find and update the best position among $(\{Xnew + \mathrm{N}step\} \cap [Xnew - Xrange/2, Xnew + Xrange/2], \{Ynew + \mathrm{N}step\} \cap [Ynew - Yrange/2, Ynew + Yrange/2])$
      $step \Leftarrow step/2, Xrange \Leftarrow Xrange/2, Yrange \Leftarrow Yrange/2$
    end while
  else
    Find the best position among the neighboors of endpoints
  end if
  Check the validity of the new drawing
  return $G'$
```

3 Result

3.1 Result presenting

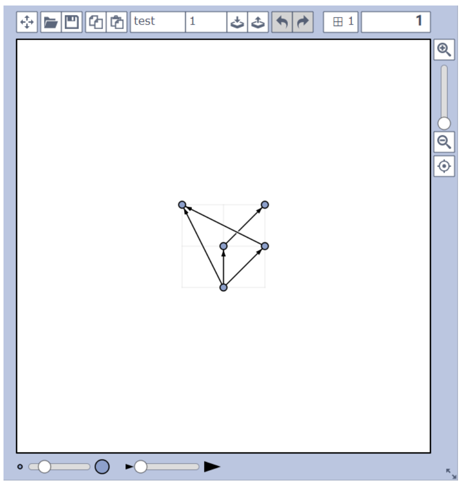
The **Figure 1** and **Figure 2** show the comparison between initial layout and localSearch output in small input, with the crossing indicated in the upper right.

In these graphs except graph-6 we set $C_{step} = height$ and $C_{range} = 1$ conducting a global search. While in order to reduce the running time we set $C_{step} = 5$ and $C_{range} = 2$ for graph-6.

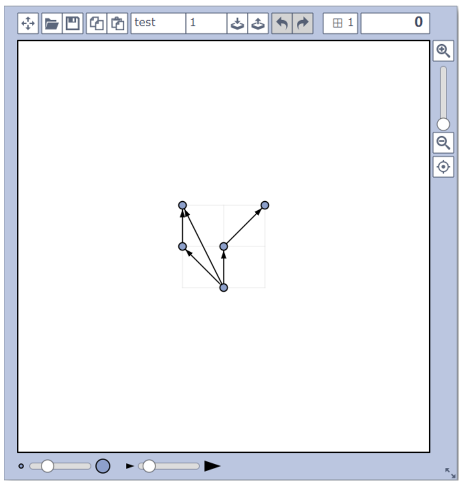
The **Figure 3** shows the comparison between initial layout and localSearch output in large input. For the reason that the initial layout is not given for large output, we have computed it by using **Algorithm 4** computInitialLayout(), which gives a compact layout of the graph.

The **Figure 4** shows the comparison between initial layout and forceDirected output in large input. The output is automatically valid after conducting Algorithm. Nevertheless it doesn't reduce too much the crossing number and the overall form of the graph.

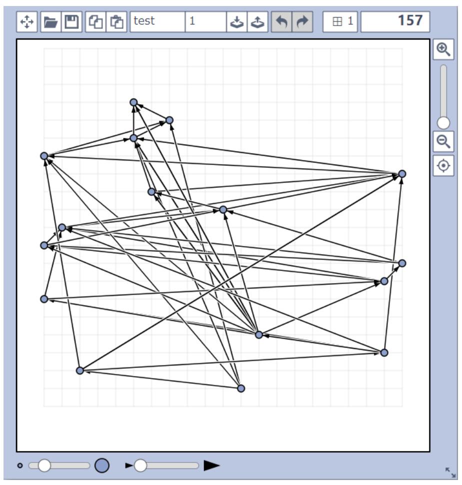
The **Figure 5** gives a whole view of the comparison among initial layout, forceDirected output and localSearch output in large input. We set the parameters of localSearch $C_{step} = 10$ and $C_{range} = 2$ From auto-7 to auto-9, while for auto-10 we set the parameters as $C_{step} = 5$ and $C_{range} = 2$ by using a relatively bigger step length to reduce the number of candidate points, in purpose of reducing the running time.



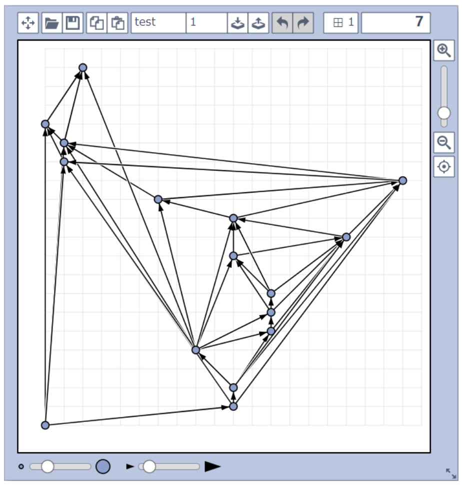
(a) graph-1



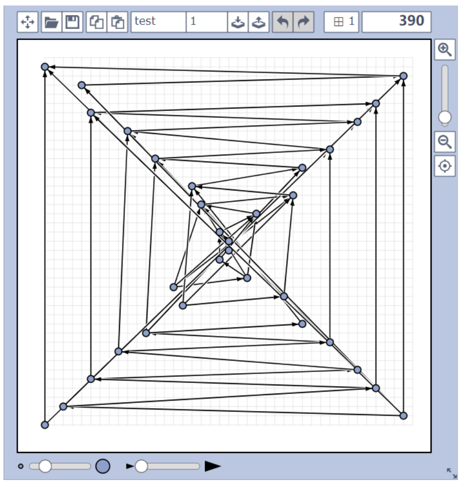
(b) localSearch graph-1



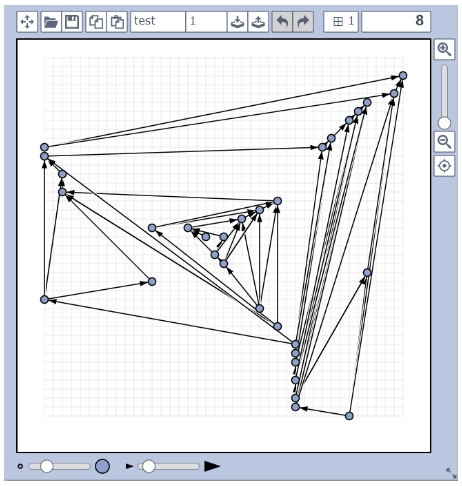
(c) graph-2



(d) localSearch graph-2

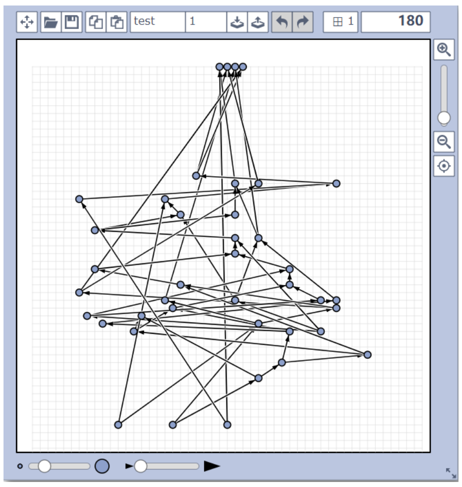


(e) graph-3

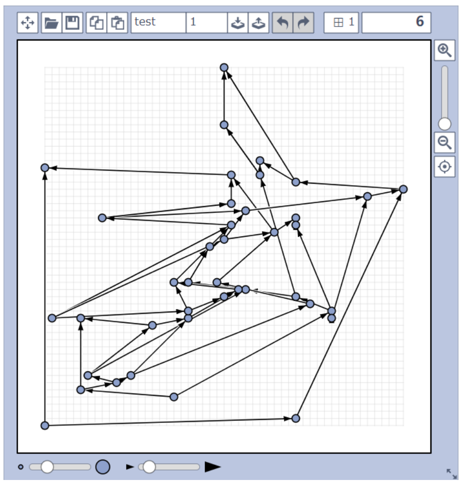


(f) localSearch graph-3

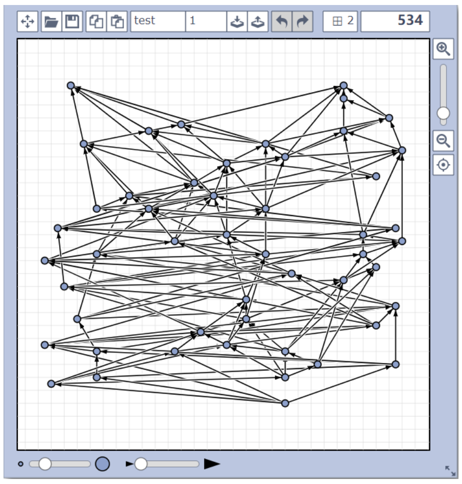
Figure 1: Comparison of initial layout and localSearch output in small input



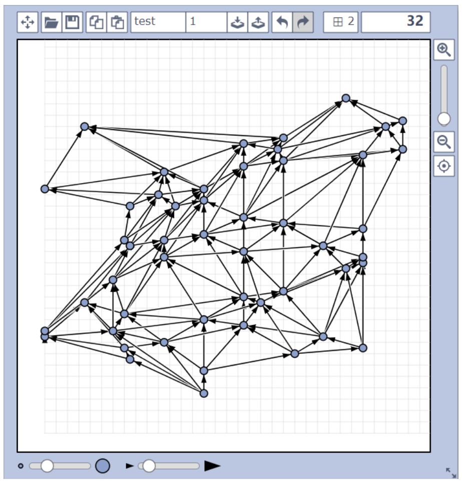
(a) graph-4



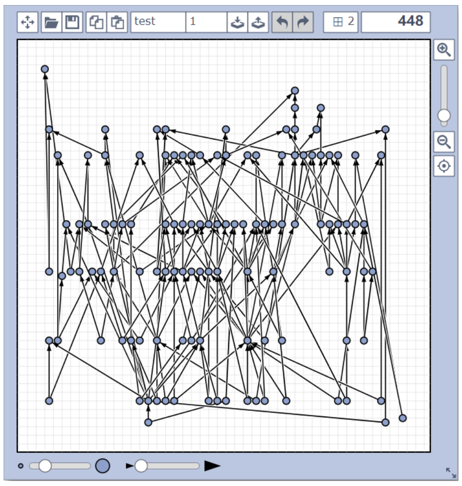
(b) localSearch graph-4



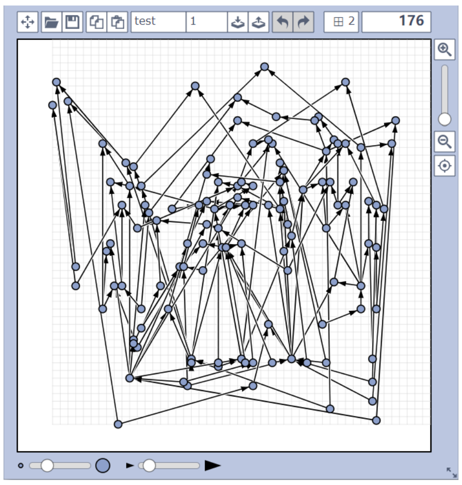
(c) graph-5



(d) localSearch graph-5

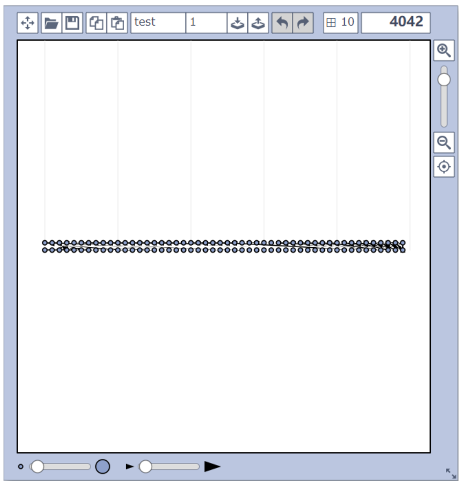


(e) graph-6

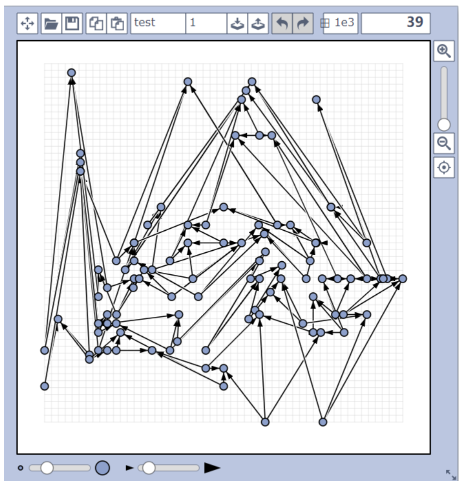


(f) localSearch graph-6

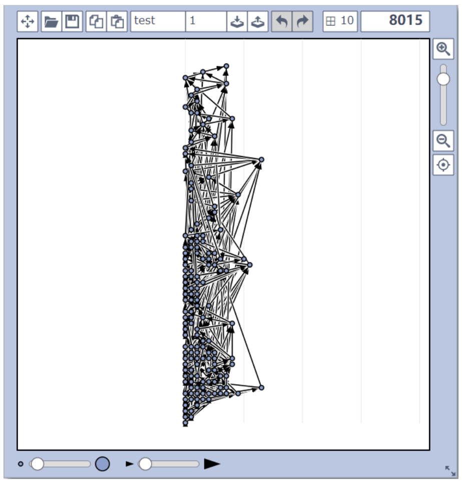
Figure 2: Comparison of initial layout and localSearch output in small input



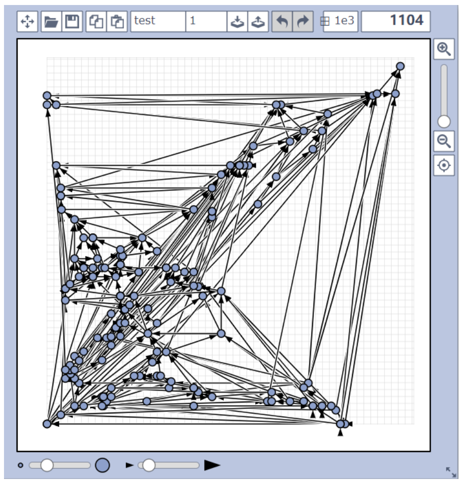
(a) auto-7



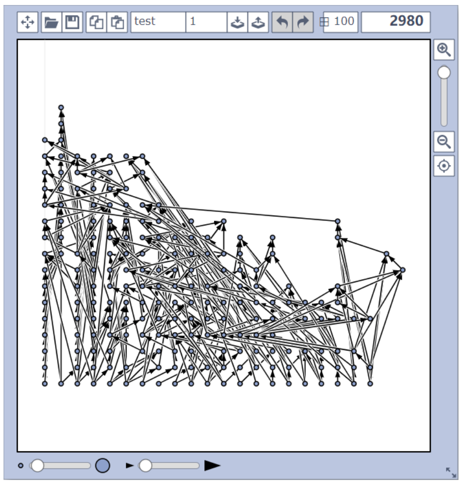
(b) localSearch auto-7



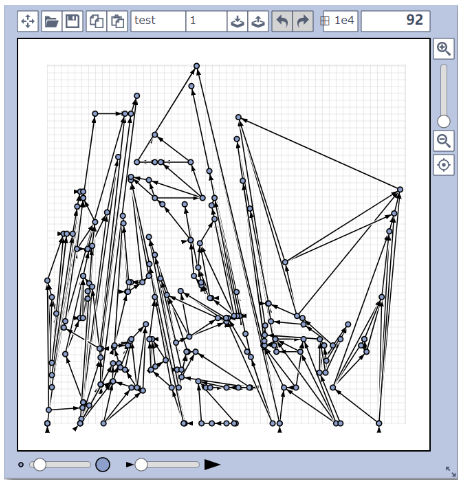
(c) auto-8



(d) localSearch auto-8

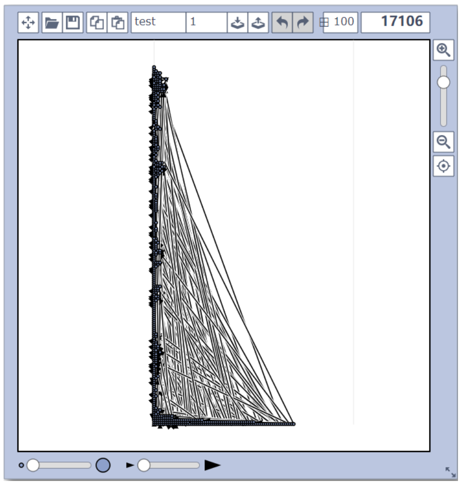


(e) auto-9

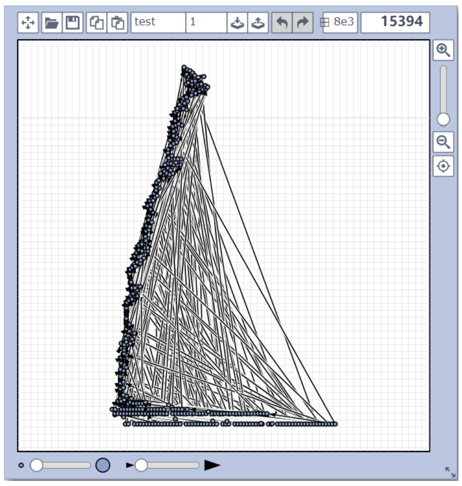


(f) localSearch auto-9

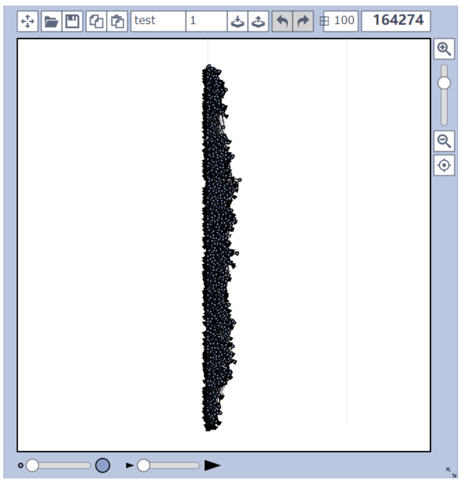
Figure 3: Comparison of initial layout and localSearch output in large input



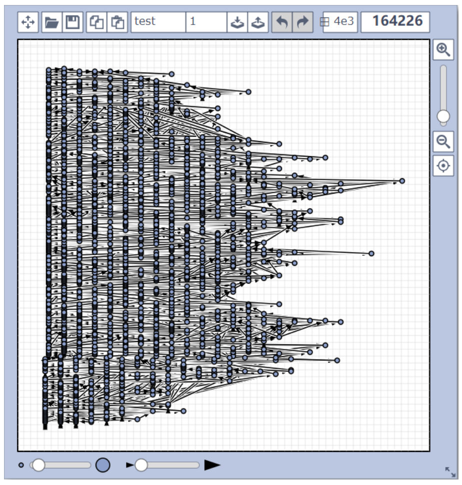
(a) auto-10



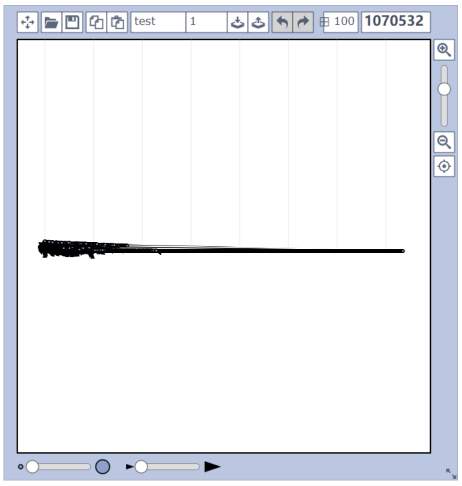
(b) forceDirected auto-10



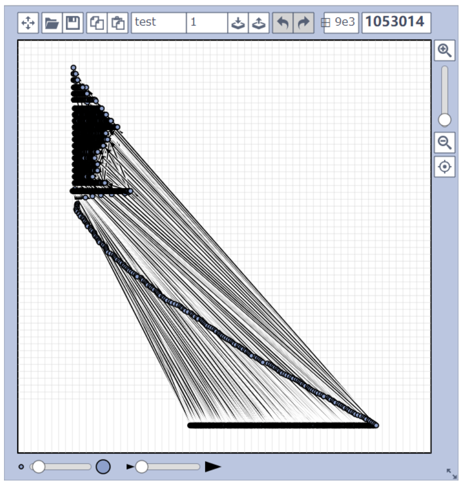
(c) auto-11



(d) forceDirected auto-11

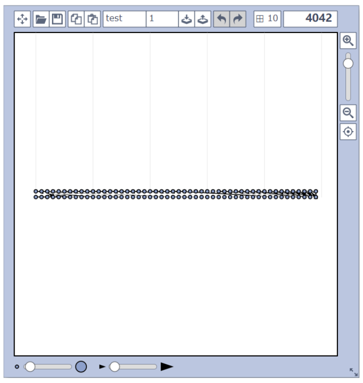


(e) auto-12

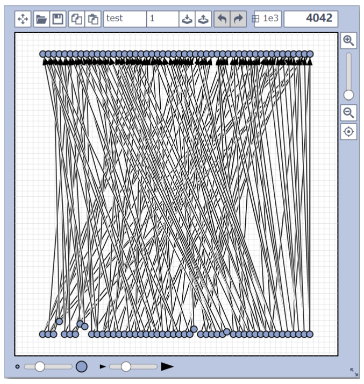


(f) forceDirected auto-12

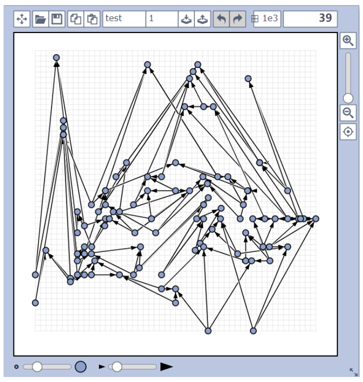
Figure 4: Comparison of initial layout and forceDirected output in large input



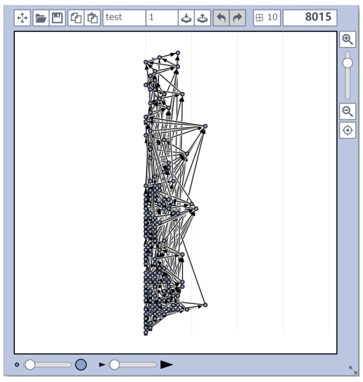
(a) Initial auto-7



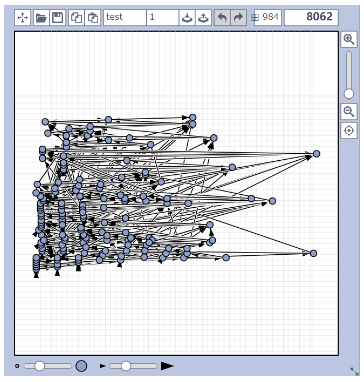
(b) forceDirected auto-7



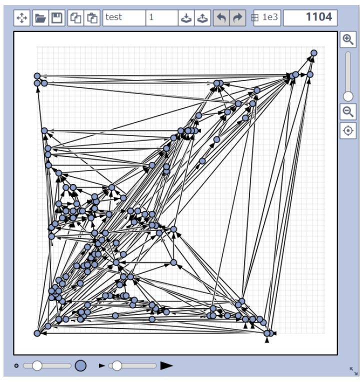
(c) localSearch auto-7



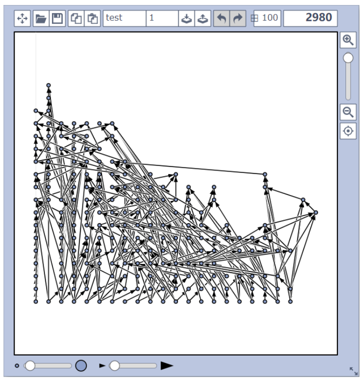
(d) Initial auto-8



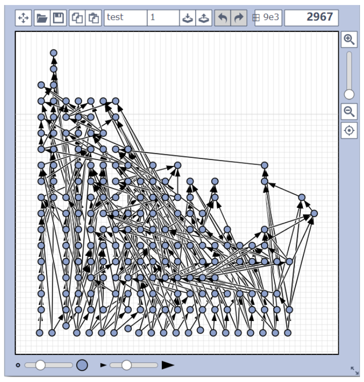
(e) forceDirected auto-8



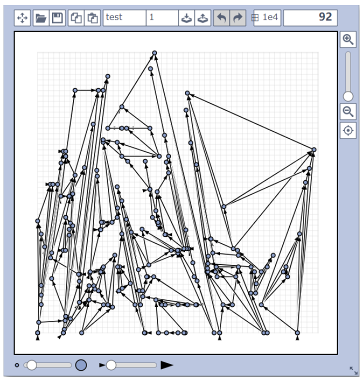
(f) localSearch auto-8



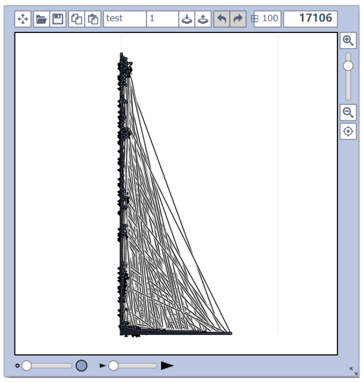
(g) Initial auto-9



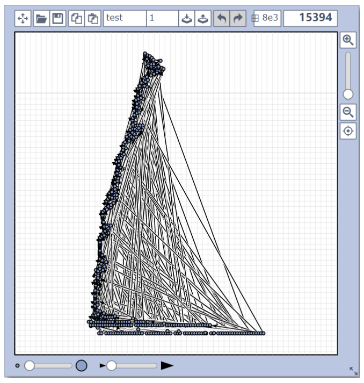
(h) forceDirected auto-9



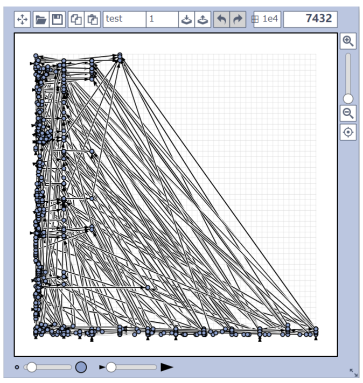
(i) localSearch auto-9



(j) Initial auto-10



(k) forceDirected auto-10



(l) localSearch auto-10

Figure 5: Comparison of initial layout, forceDirected output and localSearch output in large input

Table 1: Performance comparison.

Original crossing number	Force-directed layouts			Local search heuristic		
	Crossing number	Reduction ratio	Runtime(seconds)	Crossing Number	Reduction ratio	Runtime(seconds)
1	\	\	\	0	100%	0.01
157	\	\	\	7	95.5%	0.49
390	\	\	\	8	97.9%	4.47
180	\	\	\	6	96.7%	2.80
534	\	\	\	32	94.0%	4.83
448	\	\	\	176	60.7%	4.30
4042	4042	0.00%	0.0767551	39	99.0%	77.6
8015	8062	-0.58%	0.1362603	1104	80.1%	205
2980	2967	0.44%	0.3202732	92	96.9%	646
17106	15394	10.0%	1.0021531	7432	56.6%	900
164274	164226	0.03%	6.3856798	42083	25.6%	$> 10^7$
1070532	1053014	1.63%	12.2658151	\	\	\

3.2 Result Analysis

For large input, by adding the edge-repulsive force in the force-directed heuristic algorithm, our algorithm works very well presenting a fast output of valid drawing automatically without a check of validation. However the crossing reduction depends largely on the structure of the graph itself, for some graphs close to complete graphs the result may not be so satisfying.

On the contrary, this algorithm works not so satisfactorily in small input. especially those whose space of valid drawing is close to the space of the whole canvas.

The local search heuristic algorithm presents a highly robust quality both in small input and large input. By setting properly the parameters C_{step} and C_{range}, we can obtain a result quite satisfying. However, if the two parameters are not properly chosen, the output may ends at first iteration and returns the original initial output, or it may take a long time to converge towards our final output. Especially in large input, the running time increases rapidly with the input size.