

# A Force-Directed Algorithm that Preserves Edge Crossing Properties

François Bertault

Department of Computer Science and Software Engineering  
University of Newcastle  
Callaghan 2308 NSW Australia  
`francois@cs.newcastle.edu.au`

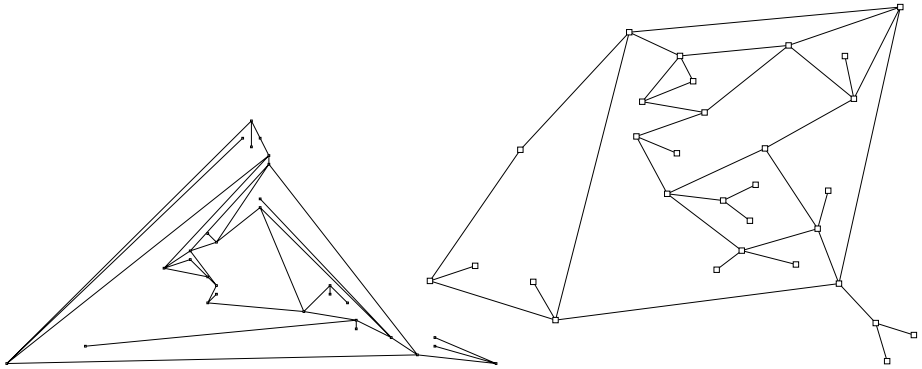
**Abstract.** We present an iterative drawing algorithm for undirected graphs, based on a force-directed approach, that preserves edge crossing properties. This algorithm insures that two edges cross in the final drawing if and only if these edges crossed on the initial layout. So no new edge crossings are introduced. We describe applications of this technique to improve classical algorithms for drawing planar graphs and for interactive graph drawing.

## 1 Introduction

Force-directed algorithms are commonly used for graph drawing because they are easy to implement and often achieve good results. Many force-directed algorithms have been proposed, which differ in the force model or the convergence method used. Usually, the force model is chosen to try to obtain uniform edge lengths, and show symmetries in the graph [2,3,4]. However, these algorithms can introduce a lot of edge crossings, which reduces the readability of the drawing. Algorithms with more complicated force models that try to reduce these edge crossings have also been proposed. These algorithms use genetic methods or simulated annealing, but are slow, difficult to parameterize, and don't insure that the resulting drawing is planar when the graph is planar.

The best classical algorithms for drawing straight-line planar graphs requires the graph to be biconnected [1,5,7]. For drawing general planar graphs, the first step is to modify the graph, by using augmentation techniques [6], that is by adding nodes and edges to the graph. Figure 1 shows an example of a planar graph, drawn using a planar graph and a basic augmentation technique. Note the lack of symmetry and variation in edge length.

The basis of the approach presented in this paper is to combine the useful characteristics of both classical planar graphs and force-directed algorithms. The approach proceeds by first finding a planar drawing using classical algorithms and then iteratively applying a new force directed algorithm, called **PrEd**. The final drawing improves symmetry and uniformity of edge lengths, while preserving initial edge-crossing properties of the graph. This insures that no new edge crossings are introduced. Figure 1 shows an example of planar graph drawn with



**Fig. 1.** Left: planar graph obtained using the LEDA planar graphs drawing algorithm. Right: planar graph obtained by applying the PrEd algorithm to the initial layout on the left. Note that the planar embedding remains the same.

this method. The initial planar layout is shown in Fig. 1. Note that the planar embedding remains the same.

The PrEd algorithm is not limited to planar graphs. The intrinsic property of the algorithm is that two edges cross on the final drawing if and only if these edges crossed on the initial layout. This property is well suited for interactive graph drawing. A user can easily indicate the desired relative position between nodes and edges by simply moving nodes interactively. The PrEd algorithm will then improve the graph symmetry and produce more uniform edge lengths.

## 2 PrEd algorithm

We consider a graph  $G = (V, E)$ , where  $V$  is a set of nodes, and  $E$  a set of undirected edges  $(a, b) = (b, a) \in V \times V$ . The number of nodes is denoted  $|V|$  and the number of edges  $|E|$ . The principle of the PrEd algorithm is the following. At each iteration, for each node  $v$  of the graph, a force  $F(v)$  is computed, that depends on the positions of the nodes and edges of the graph. Each node is then moved in the direction of  $F(v)$ . The main difference with classical force-directed algorithms is that we restrict, for each node, the maximum amplitude of the move such that the edge-crossing properties are preserved. This restriction depends on the direction of the force.

### 2.1 Computation of forces

The force model is very similar to other force-directed algorithms. Three kinds of forces between nodes and edges are considered: the attraction forces between nodes linked by an edge; the repulsion forces between each pair of nodes, and the repulsion forces between nodes and edges.

The position of a node  $v$  is denoted  $(x(v), y(v))$ . The Euclidean distance between two nodes  $a$  and  $b$  is denoted  $d(a, b)$  and  $\delta$  is the length of the edges that we would like to obtain. The force applied to a node  $v$  is denoted  $F(v) = (F_x(v), F_y(v))$ . The attraction force  $F^a(u, v)$  and repulsion force  $F^r(u, v)$  between two nodes  $u$  and  $v$  of the graph, are defined by:

$$F_x^a(u, v) = \frac{d(u, v)}{\delta} (x(v) - x(u)) \quad F_x^r(u, v) = \frac{-\delta^2}{d(u, v)^2} (x(v) - x(u))$$

This has been shown to be effective [3].

For the computation of the repulsing force  $F^e(v, (a, b))$  between a node  $v$  and an edge  $(a, b)$ , we consider a virtual node  $i_v$ , defined by the projection of the node  $v$  on the vector formed by the vertices of the edge. A force is applied to nodes  $v$ ,  $a$  and  $b$  only if the virtual node  $i_v$  is located on the edge, and if the distance between  $v$  and  $i$  is smaller than a parameter  $\gamma$ . We ignore forces if  $v = a$  or  $v = b$ .

$$F_x^e(v, (a, b)) = \begin{cases} -\frac{(\gamma - d(v, i_v))^2}{d(v, i_v)} (x(i_v) - x(v)) & \text{if } i_v \in (a, b), d(v, i_v) < \gamma \\ & v \neq a, v \neq b \\ 0 & \text{otherwise} \end{cases}$$

The overall force applied to a node  $v$  is obtained by summing the attraction and repulsion forces.

$$F_x(v) = \sum_{(u,v) \in E} F_x^a(u, v) + \sum_{u \in V} F_x^r(u, v) + \sum_{(a,b) \in E} F_x^e(v, (a, b)) - \sum_{\substack{u \in V, w \in V \\ (v,w) \in E}} F_x^e(u, (v, w))$$

We have a similar formula for the  $F_y(v)$  component.

## 2.2 Computation of the amplitude of moves

The node-edge repulsion force has been used in other algorithms in order to avoid overlapping between nodes and edges. However, since we consider discrete moves of the nodes during the steps of the algorithm, this does not guarantee that a node will not cross an edge and create a new crossing. For preserving the crossing properties between edges, a *zone*  $Z(v)$  is associated to each node  $v$ .  $Z(v)$  indicates where the node  $v$  is allowed to move. A zone is defined by eight arcs  $Z_1(v), \dots, Z_8(v)$ . The zone of a node  $v$  can be represented with only eight values  $R_1(v), \dots, R_8(v)$ , corresponding to the radius of the arcs. The zone of a node is not the maximal area in which a node is allowed to move without changing the crossing properties. The choice of representing zones by eight arcs is a good compromise between efficiency and liberty of move of the nodes.

The amplitude of move of a node  $v$ , with a force  $F(v)$  applied to it, is bounded by the value of the arc that contains  $F(v)$ . For example, in Fig. 2, the node  $v$  moves in direction of the force  $F(v)$ , with an amplitude bounded by  $R_8(v)$ . An arc with an infinite radius is represented in white on the figure.

The computation of the zones is made such that we avoid the creation of new crossings, while considering only one node and one edge at a time. For each pair of node  $v$  and edge  $(a, b)$ , the idea is to consider two cases, depending on the position of the virtual node  $i_v$  that we defined for the computation of repulsive forces between nodes and edges. In both cases, we restrict the move of the nodes  $v$ ,  $a$  and  $b$  for avoiding the creation of crossing. If the virtual node  $i_v$  is located on the edge  $(a, b)$ , we also allow the node  $v$  to “escape” by increasing its distance to the edge  $(a, b)$ . If the virtual node  $i_v$  is not located on the edge  $(a, b)$ , we allow the node  $v$  to “turn around” the edge  $(a, b)$ .

To compute the zones of each node of the graph, the eight values of each zone are first initialized to infinity. Then, for each pair of node  $v$  and edge  $(a, b)$ , we consider the position of the virtual node  $i_v$ :

Case 1 If  $i_v$  is on the edge  $(a, b)$  (Fig. 3). We consider the segment  $[v, i_v]$ , and we search which arc  $s$  of  $Z(v)$  intersects the segment. We update the values of the zones as follow:

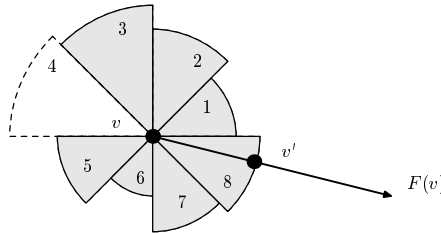
$$\begin{aligned} R_j(v) &= \min(R_j(v), d(v, i_v)/3), & j &= r(s-2), \dots, r(s+2) \\ R_j(a) &= \min(R_j(a), d(v, i_v)/3), & j &= r(s+2), \dots, r(s+6) \\ R_j(b) &= \min(R_j(b), d(v, i_v)/3), & j &= r(s+2), \dots, r(s+6) \\ &\text{where } r(j) = 1 + (j \bmod 8) \end{aligned}$$

Case 2 If  $i_v$  is not on the edge  $(a, b)$  (Fig. 4). We update the values of the zones as follow:

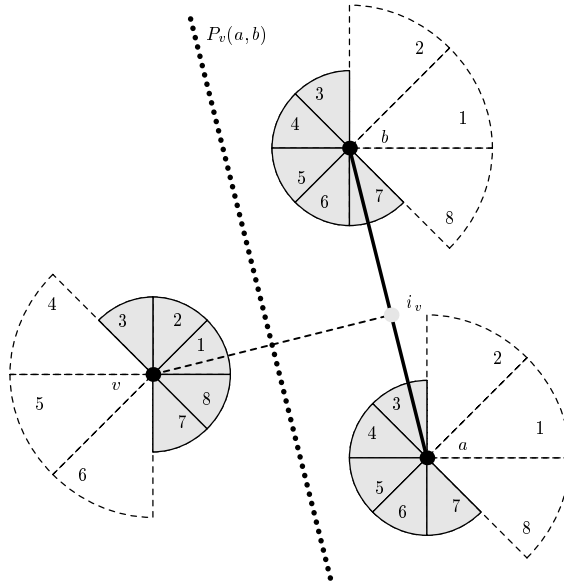
$$\begin{aligned} R_j(v) &= \min(R_j(v), \min(d(a, v), d(b, v))/3), & j &= 1, \dots, 8 \\ R_j(a) &= \min(R_j(a), d(a, v)/3), & j &= 1, \dots, 8 \\ R_j(b) &= \min(R_j(b), d(b, v)/3), & j &= 1, \dots, 8 \end{aligned}$$

In summary, the steps of the **PrEd** algorithm are, for one iteration:

- Step 1 Computation of the forces applied to each node:  $O(|V|^2 + |V||E|)$ .
- Step 2 Computation the values of the zone of each node:  $O(|V||E|)$ .
- Step 3 Move of each node, with the amplitude of the move bounded by its zone:  $O(|V|)$ .



**Fig. 2.** Move of a node  $v$  according to its zone  $Z(v)$ .



**Fig. 3.** Computation of the zones for a node  $v$  and an edge  $(a, b)$  (case 1).

The overall complexity of one step is  $O(|V|^2 + |V||E|)$ . In practice, the algorithm works well for small or sparse graphs. For example, the figure 1 was obtained in 100 iterations and 3 seconds. In some cases, the distance between nodes can be small.

### 2.3 Correctness

**Theorem 1.** *The PrEd algorithm preserves edge-crossing properties of a graph.*

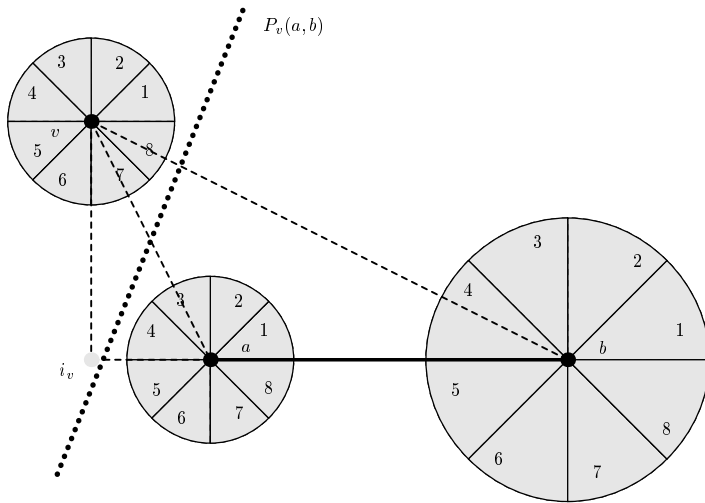
*Proof.* We first show that one step of the algorithm preserves non crossing edges. For this, we consider two non crossing edges  $(u, v)$  and  $(a, b)$ . The idea of the proof is to show that we can construct two polytopes (i.e. an intersection of half planes),  $P(u, v)$  and  $P(a, b)$ , such that we have:

$$P(u, v) \cap P(a, b) = \emptyset, Z(u) \cup Z(v) \subset P(u, v) \text{ and } Z(a) \cup Z(b) \subset P(a, b) \quad (2.1)$$

Since a polytope is convex, and the ends of edges can only move inside their zone, the edges remains in their polytope after the moves of the ends.

The key point of the proof is to show, for a node  $v$  and an edge  $(a, b)$ , that we can define a polytope  $P_v(a, b)$ , containing  $Z(a)$  and  $Z(b)$  and not  $Z(v)$ . We consider the two following cases, according to the position of the virtual node  $i_v$ . The other cases are deduced by renaming:

Case 1 If  $i_v \in (a, b)$ . We define  $P_v(a, b)$  by the half plane with a boundary orthogonal to the segment  $[v, i_v]$ , that passes throw the middle of this segment as shown in Fig. 3.



**Fig. 4.** Computation of the zones for a node  $v$  and an edge  $(a, b)$  (case 2).

Case 2 If  $i_v \notin (a, b)$ , with  $d(i_v, a) < d(i_v, b)$ . We define  $P_v(a, b)$  by the half plane with a boundary forming an angle of  $\pi/4$  with the segment  $[a, v]$ , as shown in Fig. 4. This plane contains  $Z(a)$  if the radius of  $Z(a)$  is smaller than  $d(a, v)/(2\sqrt{2})$ , and we chose to define the radius of the zones by  $d(a, v)/3$ . We also verify easily that this half plane contains also  $Z(b)$ , since the radius of  $Z(b)$  is given by  $d(b, v)/3$ .

The final step of the proof is to verify, according to the positions of the virtual nodes, that we can define the polytopes  $P(u, v)$  and  $P(a, b)$  that verify property (2.1).

Case 1 If  $i_u \in (a, b)$  and  $i_v \in (a, b)$ : we define  $P(a, b) = P_u(a, b) \cap P_v(a, b)$ , and  $P(u, v) = \mathbb{R}^2 \setminus P(a, b)$ .

Case 2 If  $i_u \in (a, b)$ ,  $i_v \notin (a, b)$ :

- (a) If  $i_a \in (u, v)$ ,  $i_b \notin (a, b)$ , and  $d(u, i_u) < d(a, i_a)$ : we define  $P(a, b) = P_u(a, b)$ , and  $P(u, v) = \mathbb{R}^2 \setminus P(a, b)$ .
- (b) If  $i_a \notin (u, v)$  and  $i_b \notin (a, b)$ : we define  $P(a, b) = P_u(a, b)$ , and  $P(u, v) = \mathbb{R}^2 \setminus P(a, b)$ .

Case 3 If  $i_u \notin (a, b)$ ,  $i_v \notin (a, b)$ ,  $i_a \notin (u, v)$  and  $i_b \notin (a, b)$ , with  $d(a, u) \leq \min(d(a, v), d(b, u), d(b, v))$ : we define  $P(a, b) = P_u(a, b)$ , and  $P(u, v) = \mathbb{R}^2 \setminus P(a, b)$ .

For verifying that one step of the algorithm preserves crossing edges, we use similar techniques. If two edges  $(u, v)$  and  $(a, b)$  cross, we remark that we can remove the crossing by moving the nodes only if we can obtain a crossing between the virtual edges  $(u, a)$  and  $(v, b)$ , or between  $(u, b)$  and  $(v, a)$ .

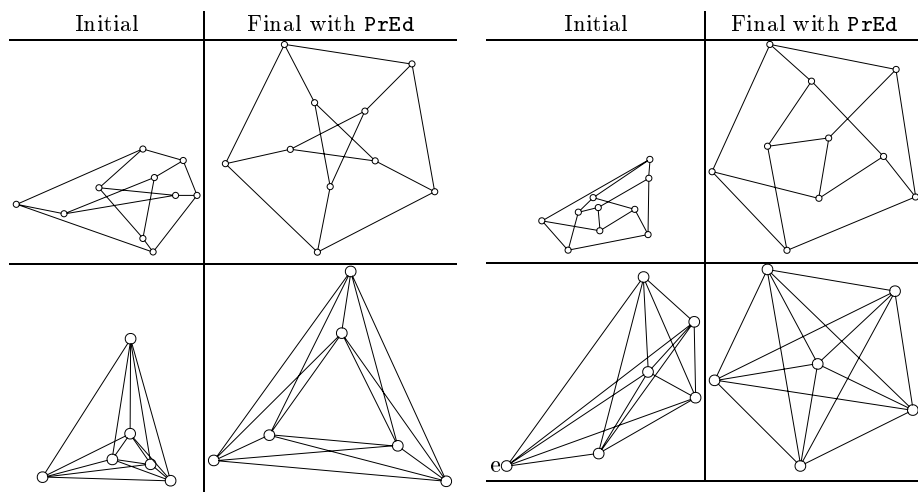
### 3 Further work

The efficiency of the algorithm could be greatly improved. We could, instead of considering all edges of the graph for defining the zone of a node, consider only a restricted set of edges that “surround” that node. The set of edges that we need to consider could be determined in a preprocessing step of the algorithm.

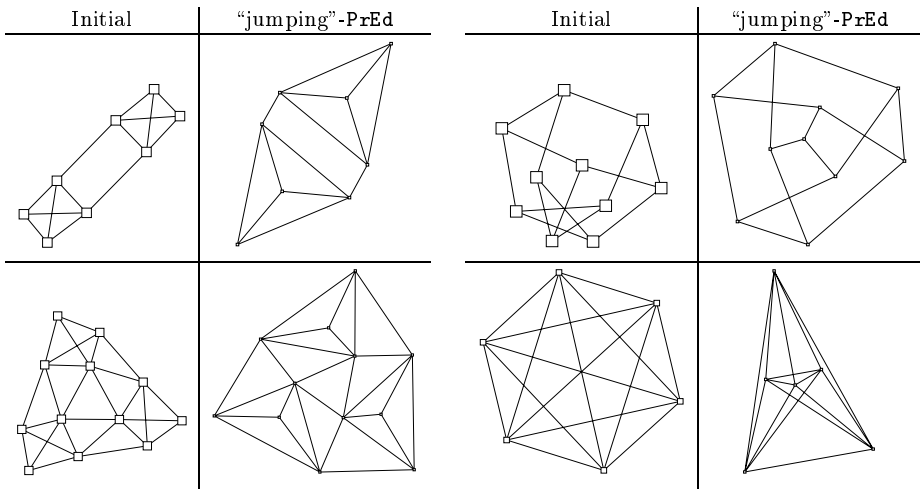
The **PrEd** algorithm can also be combined with a simple “jumping” heuristic, for reducing edge-crossings in general graph drawing. The initial positions of nodes are obtained using a classical force-directed algorithm. Then, for each pair of edge crossings, we test, during the iterations of the **PrEd** algorithm, if the fact to remove this crossing, by placing one end near the intersection, reduces the total number of crossings in the graph. This naive approach can give good results on simple graphs, but is very expensive. Figure 6 demonstrates this algorithm to remove edge crossings. Further work could be to look for better heuristics based on this idea.

### 4 Conclusion

The **PrEd** algorithm takes  $O(|V|^2 + |V||E|)$  time to complete one iteration. It preserves crossing and non-crossing edges during the iteration steps. This property is well suited for improving the drawing of straight-line planar graphs, where the initial position of nodes is obtained from a classical algorithm.



**Fig. 5.** Examples of interactive drawing with the **PrEd** algorithm. The initial position of nodes is specified by the user, and the **PrEd** algorithm improves symmetry and edge length criteria without changing the relative position of nodes.



**Fig. 6.** Combination of a the PrEd algorithm with a “jumping” operation for reducing edge-crossings.

The PrEd algorithm can also be applied for interactive drawing of general graphs. The user can modify interactively the position of the nodes, and the algorithm improves the drawing criteria such as symmetry and uniform edges lengths. Relative positions of nodes are preserved by the algorithm. Small modifications of the graph induces small modifications on the new representation of the graph. Examples of drawings obtained this way are shown in Fig. 5.

## References

1. H. de Fraysse, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10:41–51, 1990.
2. P. D. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.
3. T. Fruchterman and E. Reingold. Graph drawing by force-directed placement. *Software-Practice and Experience*, 21(11):1129–1164, 1991.
4. T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31:7–15, 1989.
5. G. Kant. Drawing planar graphs using the *lmc*-ordering. *Proc. IEEE Symp. on Foundation of Computer Science*, pages 101–110, 1992.
6. Goos Kant and Hans L. Bodlaender. Triangulating planar graphs while minimizing the maximum degree. *Information and Computation*, 135(1):1–14, 1997.
7. P. Mutzel. A fast linear time embedding algorithm based on the Hopcroft-Tarjan planarity test. Technical report, Institut für Informatik, universität zu Köln, 1992.