Crossing Reduction in Circular Layouts[⋆]

Minor Revisions (June 26, 2008)

Michael Baur[1] and Ulrik Brandes[2]

[1] Department of Computer Science, Universität Karlsruhe (TH), Germany.
`baur@informatik.uni-karlsruhe.de`

[2] Department of Computer & Information Science, University of Konstanz, Germany.
`Ulrik.Brandes@uni-konstanz.de`

Abstract. We propose a two-phase heuristic for crossing reduction in circular layouts. While the first algorithm uses a greedy policy to build a good initial layout, an adaptation of the sifting heuristic for crossing reduction in layered layouts is used for local optimization in the second phase. Both phases are conceptually simpler than previous heuristics, and our extensive experimental results indicate that they also yield fewer crossings. An interesting feature is their straightforward generalization to the weighted case.

1 Introduction

In circular graph layout, the vertices of a graph are constrained to distinct positions along the perimeter of a circle, and an important objective is to minimize the number of edge crossings in such layouts. Since circular crossing minimization is $\mathcal{NP}$-hard [8], several heuristics have been devised [7,3,14]. Moreover there is a factor $\mathcal{O}(\log^2 |V|)$ approximation algorithm [13].

We propose a two-phase approach for obtaining circular layouts with few crossings. In the first phase, vertices are iteratively added to either end of a linear layout. This leaves three degrees of freedom: the start vertex, the insertion order, and the end at which to append the next vertex. For the different strategies tried, empirical evidence suggests that a particular one outperforms both the others and previous heuristics.

For the second phase, we adapt a local optimization procedure for layered layouts, sifting [9], to the circular case. Note that, similar to 2-layer layouts, the number of crossing is completely determined by the (cyclic) ordering of vertices. The thus related one-sided crossing minimization problem in 2-layer drawings of bipartite graphs is $\mathcal{NP}$-hard as well [5], but significantly better understood. It turns out that circular sifting reduces the number of crossings both with respect to our first phase and previous heuristics.

After defining some terminology in Section 2, we describe our greedy append and circular sifting algorithms for the phases in Sections 3 and 4. Both are evaluated experimentally in Section 5.

2 Preliminaries

Throughout this paper, let $G = (V, E)$ be a simple undirected graph with $n = |V|$ vertices and $m = |E|$ edges. Furthermore, let $N(v) = \{u \in V : \{u, v\} \in E\}$ denote the neighborhood of a vertex $v \in V$. A *circular layout* of G is a bijection $\pi : V \to \{0, \dots, n-1\}$, interpreted as a clockwise sequence of distinct positions on the circumference of a circle. By selecting a reference vertex $s \in V$ we obtain linear orders $\prec_s^\pi$ from π by defining

$$u \prec_s^\pi v \iff (\pi(u) - \pi(s) \mod n) < (\pi(v) - \pi(s) \mod n)$$

for all $u, v \in V$, i.e. u is encountered before v in a cyclic traversal starting from s. We say that $u, v \in V$ are *consecutive*, denoted by $u \curvearrowright_\pi v$, if $\pi(v) - \pi(u) \equiv 1 \mod n$. A subset $W \subset V$ is

[⋆] This is the author's version. Research partially supported by DFG under grants Wa 654/13-2 and Br 2158/1-2.

consecutive, if there is an ordering of the vertices of W so that $w_0 \curvearrowright_\pi w_1 \curvearrowright_\pi \ldots \curvearrowright_\pi w_{|W|-1}$, $w_i \in W$.

Let

$$\chi_\pi(\{u_1, v_1\}, \{u_2, v_2\}) = \begin{cases} 1 & \text{if } u_1 \prec^\pi_{u_1} u_2 \prec^\pi_{u_1} v_1 \prec^\pi_{u_1} v_2 \\ 0 & \text{otherwise .} \end{cases} \tag{1}$$

for all $\{u_1, v_1\}, \{u_2, v_2\} \in E$ and w.l.o.g. $\pi(u_i) < \pi(v_i)$. We say that $e_1, e_2 \in E$ *cross* in π, iff $\chi_\pi(e_1, e_2) = 1$, i.e. the endvertices of e_1, e_2 are encountered alternately in a cyclic traversal. The *crossing number* of a circular layout π is $\chi(\pi) = \sum_{e_1,e_2 \in E} \chi_\pi(e_1, e_2)$ and $\chi(G) = \min_\pi \chi(\pi)$ is called the *circular crossing number* of G. We will omit π from our notation whenever the layout is clear from context.

Theorem 1 ([8]). *Circular crossing minimization is $\mathcal{NP}$-hard.*

On the other hand, a graph has a circular layout with no crossings, if and only if it is outerplanar. A linear time recognition algorithm for outerplanar graphs [11] is easily extended to yield a crossing-free circular layout [14].

Since, in particular, trees have circular layouts with no crossings, it is possible to consider the biconnected components of a graph separately, and insert their circular layouts into a crossing-free layout of the block-cutpoint-tree without producing additional crossings (see Fig. 1). Hence, only biconnected graphs are used in the experimental evaluation summarized in Section 5.

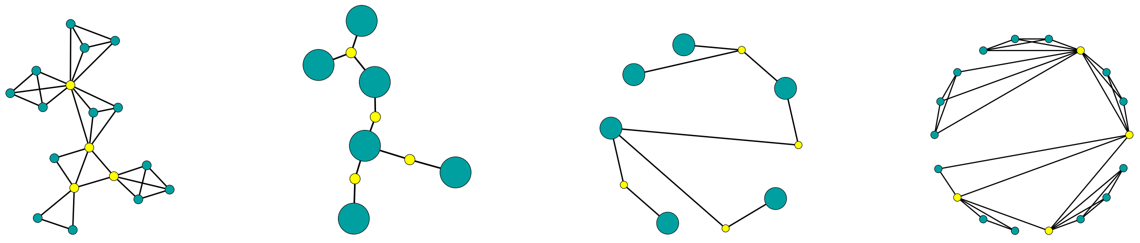

Fig. 1. The circular crossing number of a graph is the sum of those of its biconnected components (cutpoints shown in lighter color)

3 Initial Layout

Our approach for an initial layout is inspired by a heuristic algorithm for the minimum total edge length problem in circular layouts [7]. This problem is somewhat related to crossing minimization, since shorter edges tend to cross few other edges.

The basic idea is simple: start with a layout consisting of a single vertex and place the other vertices, one at a time, at either end of the current (linear) layout (see Algorithm 1). After all vertices are inserted, the final layout is considered to be circular. This method leaves us with three parameters to choose:

- the start vertex s,
- the processing sequence, and
- the end to append the next vertex at.

Note that the processing sequence need not to be fixed in the beginning, but may be determined while the algorithm proceeds. Since, in our experiments, the rules for choosing a start vertex had little influence on the final result, it is chosen at random. In the following we describe instantiations for the other two parameters.

During the algorithm some vertices are already placed while others are not. An edge is called *open*, if it connects a placed vertex with an unplaced one, and *closed*, if both its vertices have been inserted.

Four rules for determining an insertion order are investigated. The rationale behind these heuristics is to keep the number of open edges low, because they tend to result in crossings later on.

1. *Degree.* Vertices are inserted in non-increasing order of their degree.
2. *Inward Connectivity.* At each step, a vertex with the largest number of already placed neighbors is selected, i.e. a vertex which closes the most open edges.
3. *Outward Connectivity.* At each step, a vertex with the least number of unplaced neighbors is selected, i.e. a vertex which opens the fewest new edges.
4. *Connectivity.* At each step, a vertex with the largest number of already placed neighbors is selected, where ties are broken in favor of vertices with fewer unplaced neighbors.

The other degree of freedom left is the selection of an end of the current layout at which to append the next vertex. Again, four rules of choice are investigated.

1. *Random.* Select the end at which to append randomly each time.
2. *Fixed.* Always append to the same end.
3. *Length.* Append each vertex to the end that yields the smaller increase in total edge length.
4. *Crossings.* Append each vertex to the end that yields fewer crossing of edges being closed with open edges. In Fig. 2, there are eight such crossings for the left end and only six for the right end. Note that crossings with closed edges not incident to the currently inserted vertex need not be considered because they are the same for both sides. It should also be noted that crossings with open edges are independent of the positions at which the unplaced vertex will eventually be placed.

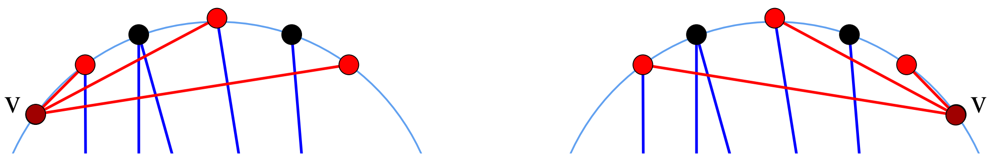

Fig. 2. Incident edges of v cross open edges

The experiments outlined in Section 5 show that the combination of the *Connectivity* insertion order with *Crossings* outperforms all other combinations, and it can be implemented efficiently.

Theorem 2. *The Greedy-Append heuristic with* Connectivity *insertion order and end-to-append selection based on* Crossings *can be implemented to run in* $\mathcal{O}((n+m)\log n)$ *time.*

Algorithm 1: Greedy-Append Heuristic

```
place start vertex $s \in V$ arbitrarily;
$V \leftarrow V \setminus \{s\}$;
while $V \neq \emptyset$ do
    greedily choose $v \in V$;
    append $v$ at either end of the current layout;
    $V \leftarrow V \setminus \{v\}$;
```

Proof. The insertion sequence can be realized by storing all unplaced vertices in a two-dimensional priority queue, in which the first key gives the number of already placed neighbors and the second the number of unplaced neighbors. With an efficient implementation, update and extract operations require $\mathcal{O}(\log n)$ time. Since each vertex is extracted once, and each edge triggers exactly one update, the total running time for determining the insertion order is $\mathcal{O}((n+m)\log n)$.

The number of crossings with open edges can be determined from prefix and suffix sums over vertices already in the layout. These can be maintained efficiently using a balanced binary tree storing in its leaves the number of open edges incident to a placed vertex, and in its inner nodes the sum of the values of its two children. The prefix sum at a vertex is the sum of all values in left children of nodes on the path from the corresponding leaf to the root. The suffix sum is determined symmetrically. Insertion of a vertex thus requires $\mathcal{O}(\log n)$ time to determine the crossing numbers from prefix and suffix sums and $\mathcal{O}(d(v)\log n)$ for updating the tree. The total is again $\mathcal{O}((n+m)\log n)$. □

Note that the heuristic is easily generalized to weighted graphs. In the next section we show how to further reduce the number of crossings, given an initial layout.

4 Improvement by Circular Sifting

Sifting was originally introduced as a heuristic for vertex minimization in ordered binary decision diagrams [12] and later adapted for the one-sided crossing minimization problem [9]. The idea is to keep track of the objective function while moving a vertex along a fixed ordering of all other vertices. The vertex is then placed in its (locally) optimal position. The method is thus an extension of the greedy-switch heuristic [4].

For crossing reduction the objective function is the number of crossings between the edges incident to the vertex under consideration and all other edges. The efficient computation of crossing numbers in sifting for layered layouts is based on the crossing matrix. Its entries correspond to the number of crossings caused by pairs of vertices in a particular linear ordering and are computed easily in advance. Whenever a vertex is placed in a new position only a smallish number of updates is necessary.

It is not possible to adapt the crossing matrix to the circular case, since two vertices cannot be said to be in a (linear) order generally. Thus we define the crossing number

$$c_{uv}(\pi) = \sum_{x\in N(u)} \sum_{y\in N(v)} \chi_\pi(\{u,x\},\{v,y\}) \tag{2}$$

only for pairs of consecutive vertices $u \curvearrowright v \in V$ and use the following exchange property, which is the basis for sifting and holds nevertheless.

Lemma 1. *Let $u \curvearrowright v \in V$ be consecutive vertices in a circular layout π, and let π' be the layout with their positions swapped, then*

$$\begin{aligned}\chi(\pi') &= \chi(\pi) - c_{uv}(\pi) + c_{vu}(\pi')\\ &= \chi(\pi) - \sum_{x\in N(u)} |\{y\in N(v) : y \prec_x^\pi u\}| + \sum_{y\in N(v)} \left|\{x\in N(u) : x \prec_y^{\pi'} v\}\right|\end{aligned}$$

Proof. Since u and v are consecutive, edges incident to neither u nor v do not change their crossing status. The first equality follows immediately. For the second equality, observe that the sums are obtained from (2) by inserting (1). See Fig. 3 for an illustration. □

Based on the above lemma, the locally optimal position of a single vertex can be found by iteratively swapping the vertex with its neighbor and recording the change in crossing count, which is computed by considering only edges incident to one of these two vertices. After the vertex has been moved past every other vertex, it is placed where the intermediary crossing counts reached their minimum. Repositioning each vertex once in this way is called a *round of circular sifting.*

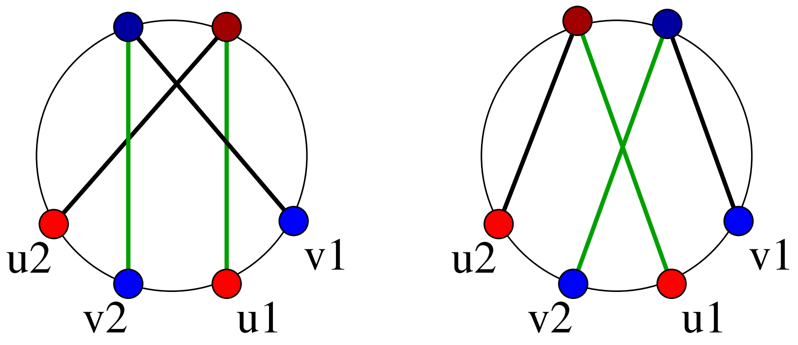

Fig. 3. After swapping consecutive vertices $u \curvearrowright v$, exactly those pairs of edges cross that did not before

Algorithm 2: Circular sifting

```
for ($u \in V$) do
    let $v_0 = u \prec_u v_1 \prec_u \ldots \prec_u v_{n-1}$ denote the current layout;
    for ($v \in V$) do
        sort adjacency list of $v$ according to the current layout;
    $\chi \leftarrow 0$;  $\chi^* \leftarrow 0$;  $v^* \leftarrow v_{n-1}$;
    for ($k \leftarrow 1, \ldots, n-1$) do
        let $x_0 \prec_{v_k} \ldots \prec_{v_k} x_{r-1}$ denote the adjacency list of $u$ without $v_k$;
        let $y_0 \prec_{v_k} \ldots \prec_{v_k} y_{s-1}$ denote the adjacency list of $v_k$ without $u$;
        $c \leftarrow 0$;  $i \leftarrow 0$;  $j \leftarrow 0$;
        while ($i < r$ and $j < s$) do
            if ($x_i \prec_{v_k} y_j$) then
                $c \leftarrow c - (s-j)$;  $i \leftarrow i+1$;
            else if ($y_j \prec_{v_k} x_i$) then
                $c \leftarrow c + (r-i)$;  $j \leftarrow j+1$;
            else
                $c \leftarrow c - (s-j) + (r-i)$;  $i \leftarrow i+1$;  $j \leftarrow j+1$;
        $\chi \leftarrow \chi + c$;
        if ($\chi < \chi^*$) then $\chi^* \leftarrow \chi$;  $v^* \leftarrow v_k$;
    move $u$ so that $v^* \curvearrowright u$;
```

If adjacency lists are ordered according to the current layout, the sums in Lemma 1 are over suffix lengths in these lists. Updating the crossing count therefore corresponds to merging the adjacency lists, where the length of the remaining suffix is added or subtracted.

Theorem 3. *One round of circular sifting takes $\mathcal{O}(nm)$ time.*

Proof. Sorting the adjacency lists according to the vertex order is easily done in $\mathcal{O}(m)$ time (traverse the vertices in order, and add each to the adjacency lists of its neighbors). If adjacency lists are stored cyclically, a head pointer yields $\prec_v$ for arbitrary v, i.e. the adjacency lists need not be reordered before a swap. The final relocation of u takes time $\mathcal{O}(1)$.

When swapping u with neighbor v_k the adjacency lists are traversed in time $\mathcal{O}(d_G(u)+d_G(v_k))$. Since

$$\sum_{u\in V}\sum_{v\in V}\big(d_G(u)+d_G(v)\big) = \sum_{u\in V}\sum_{v\in V} d_G(u) + \sum_{u\in V}\sum_{v\in V} d_G(v) = 2\cdot n\cdot 2m$$

the total running time is in $\mathcal{O}(nm)$. □

At the end of the outer loop each vertex is placed at its locally optimal position, so that circular sifting can only decrease the number of crossings. Our experiments outlined in the next section suggest that a few rounds of sifting suffice to reach a local minimum.

Note that in edge-weighted graphs we can define the *weighted crossing number* by counting each crossing with the product of the two edge weights involved. If suffix cardinalities are replaced by suffix sums of weights, Lemma 1 generalizes to the weighted case. Modifying the algorithm accordingly is straightforward.

5 Experimental Evaluation

We performed extensive experiments to determine the relative behavior of the different variants of our heuristics. As a base reference we use CIRCULAR [14], the currently most effective heuristic for circular crossing minimization. CIRCULAR consists of two phases as well: an initial placement (CIRCULAR 1) derived from a recognition algorithm for outerplanar graphs [11], and a subsequent improvement phase (CIRCULAR 2) that probes alternative positions for each vertex and relocates if the number of crossings is reduced. While the second phase appears to be similar to circular sifting, it differs in that a vertex is moved to fewer candidate positions and may thus miss good positions. Note also that CIRCULAR 2 actually counts crossings (rather than just changes) so that its running time depends on the number of crossings. When restricting replacements to a subset of positions, circular sifting simulates CIRCULAR 2 with an improved worst-case performance, but in our experiments we rather implemented an improved method for counting crossings, since realistic graphs have relatively few crossings anyway.

All algorithms have been implemented by the same person in C++ using LEDA [10]. Our experiments were carried out on a standard desktop computer with 1.5 GHz and 512 MB running Linux. Each data point is the average of 10 runs with different internal initializations (in particular, permuted adjacency lists).

The experiments were run on three families of undirected, biconnected graphs (recall from Section 2 that crossings between edges in different biconnected components can be avoided altogether):

– *Rome graphs.* A set of 10 541 biconnected components with 10 to 80 vertices used in [2]. These are sparse real-world graphs with $m \approx 1.3n$.
– *Fixed average degree.* Three sets of random graphs with 10 to 200 vertices and variable edge probability of $\frac{3}{n-1}$, $\frac{5}{n-1}$, and $\frac{10}{n-1}$, resulting in graphs with expected average degree of 3, 5, and 10.
– *Fixed density.* Three sets of random graphs with 10 to 200 vertices and fixed edge probability of 0.02, 0.05, and 0.1, resulting in graphs with expected density of 2, 5, and 10 percent.

A selection of results is given in the appendix. For a comprehensive list of figures see [1]. We here summarize our conclusions.

5.1 Initialization using Greedy Append

The performance of various combinations of insertion orders for greedy append is shown in Fig. 4 relative to CIRCULAR 1. While for some rules of choice the results depend on number of edges in the graph, the *Connectivity* variant consistently outperforms all others, including CIRCULAR 1. The results in Fig. 5 indicate that appropriate placement is indeed important, but has a much smaller effect than the insertion order. On random graphs, the combination of *Connectivity* insertion with *Length* or *Crossings* perform almost equally well, with a slight advantage for *Crossings.*

The two best combinations, *Connectivity* with *Length* or *Crossings*, compare favorably with CIRCULAR 1 in terms of the resulting number of crossings (see Figs. 7). Note that the running time of the initialization methods is negligible, especially when compared to the improvement strategies.

5.2 Subsequent Improvement using Circular Sifting

Circular sifting reaches a local minimum in few rounds. As can be expected, the improvement is larger in early rounds, and the number of rounds required depends on the initial configuration (see Fig. 6). It can be concluded that the improvement algorithms (circular sifting and CIRCULAR 2) should not be used by themselves, but only in combination with a good initialization method.

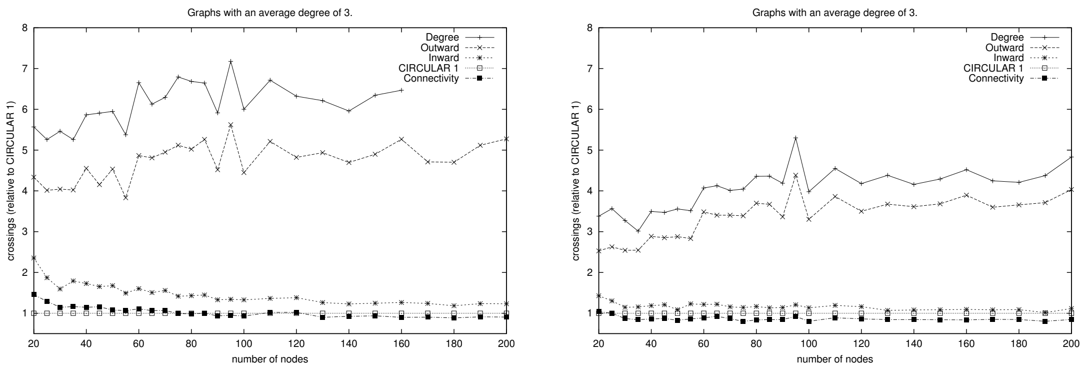

Fig. 4. Greedy append: insertion orders combined with *Fixed* (left) and *Crossings* (right) placement rules

.

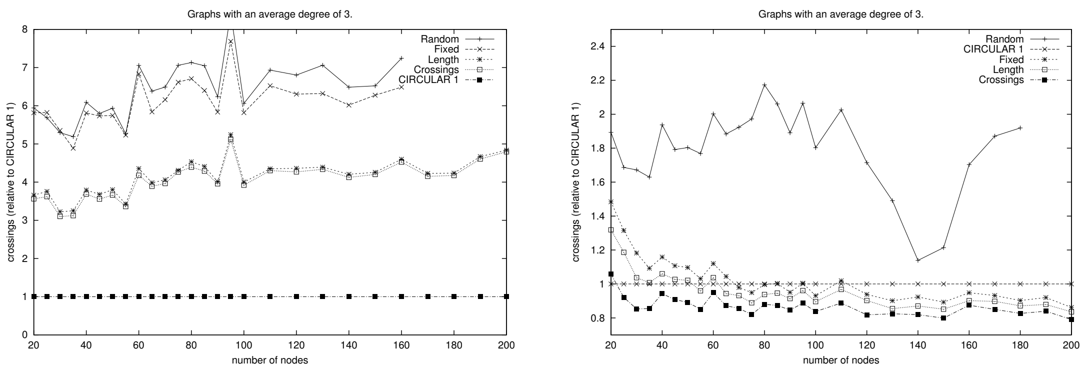

Fig. 5. Greedy append: placement rules combined with *Degree* (left) and *Connectivity* (right) insertion orders

With any of the good initialization strategies identified in the previous subsection, circular sifting is able to further reduce the number of crossings produced by CIRCULAR 2 as can be seen in Figs. 7 and 8 and is also confirmed by an independent study of He and Sýkora [6]. This suggests that the additional positions considered for relocation indeed pay off. However, there is a slight runtime penalty if sifting is run until there is no further improvement.

Conclusion

We have presented an approach for circular graph layout with few crossings. It consists of two phases: in the first phase, we greedily append vertices to either end of a partial (linear) layout according to some criteria, and in the second we further reduce the number of crossings by repeatedly sifting each vertex to a locally optimal position.

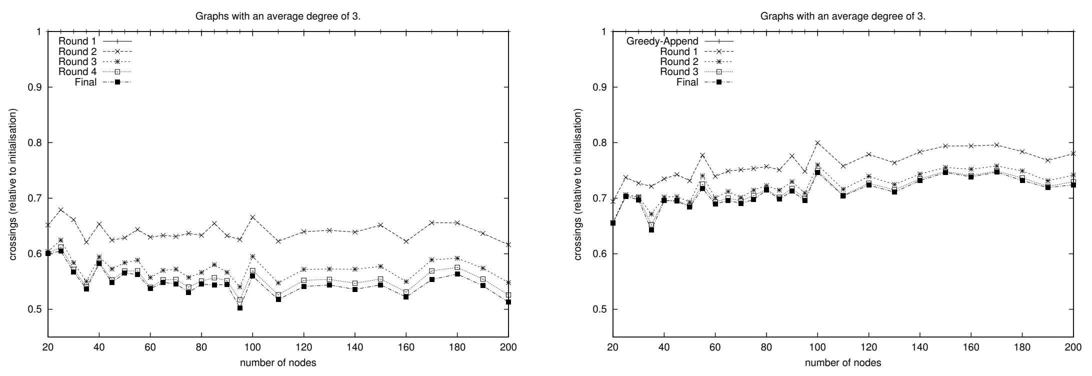

Fig. 6. Circular sifting: improvement after various rounds without initialization (left) and with greedy append (right)

Our experimental evaluation clearly shows that the method of choice is to initialize circular sifting with a greedy-append approach using the *Connectivity* insertion order with the *Crossings* placement rule and that this combination consistently outperforms previous heuristics. They also show that both phases are necessary. While circular sifting yields a substantial improvement over the initial layouts, a good initialization significantly reduces the number of rounds required and thus the overall running time at essentially no extra cost.

References

1. M. Baur and U. Brandes. Crossing Reduction in Circular Layouts. Technical Report 2004-14, ITI Wagner, Faculty of Informatics, Universität Karlsruhe (TH), 2004. 6
2. G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An Experimental Comparison of Four Graph Drawing Algorithms. *Computational Geometry: Theory and Applications*, 7(5-6):303–325, April 1997. 6
3. U. Dogrusöz, B. Madden, and P. Madden. Circular Layout in the Graph Layout Toolkit. In *Proceedings of the 4th International Symposium on Graph Drawing (GD'96)*, volume 1090 of *Lecture Notes in Computer Science*, pages 92–100. Springer, January 1997. 1
4. P. Eades and D. Kelly. Heuristics for Reducing Crossings in 2-Layered Networks. *Ars Combinatoria*, 21(A):89–98, 1986. 4
5. P. Eades and N. C. Wormald. Edge Crossings in Drawings of Bipartite Graphs. *Algorithmica*, 11(4), April 1994. 1
6. H. He and O. Sýkora. New Circular Drawing Algorithms, 2004. Unpublished manuscript. 7
7. E. Mäkinen. On Circular Layouts. *International Journal of Computer Mathematics*, 1988. 1, 2
8. S. Masuda, T. Kashiwabara, K. Nakajima, and T. Fujisawa. On the $\mathcal{NP}$-completeness of a computer network layout problem. In *Proceedings of the 20th IEEE International Symposium on Circuits and Systems 1987*, pages 292–295. IEEE Computer Society, 1987. 1, 2
9. C. Matuszewski, R. Schönfeld, and P. Molitor. Using Sifting for k -Layer Straightline Crossing Minimization. In *Proceedings of the 7th International Symposium on Graph Drawing (GD'99)*, volume 1731 of *Lecture Notes in Computer Science*, pages 217–224. Springer, January 2000. 1, 4
10. K. Mehlhorn and S. Näher. *LEDA, A platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999. 6
11. S. L. Mitchell. Linear Algorithms to Recognize Outerplanar and Maximal Outerplanar Graphs. *Information Processing Letters*, 9(5):229–232, December 1979. 2, 6
12. R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design*, pages 42–47. IEEE Computer Society, 1993. 4
13. F. Shahrokhi, O. Sýkora, L. A. Székely, and I. Vrto. Book Embeddings and Crossing Numbers. In *Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'93)*, volume 903 of *Lecture Notes in Computer Science*, pages 256–268. Springer, 1994. 1

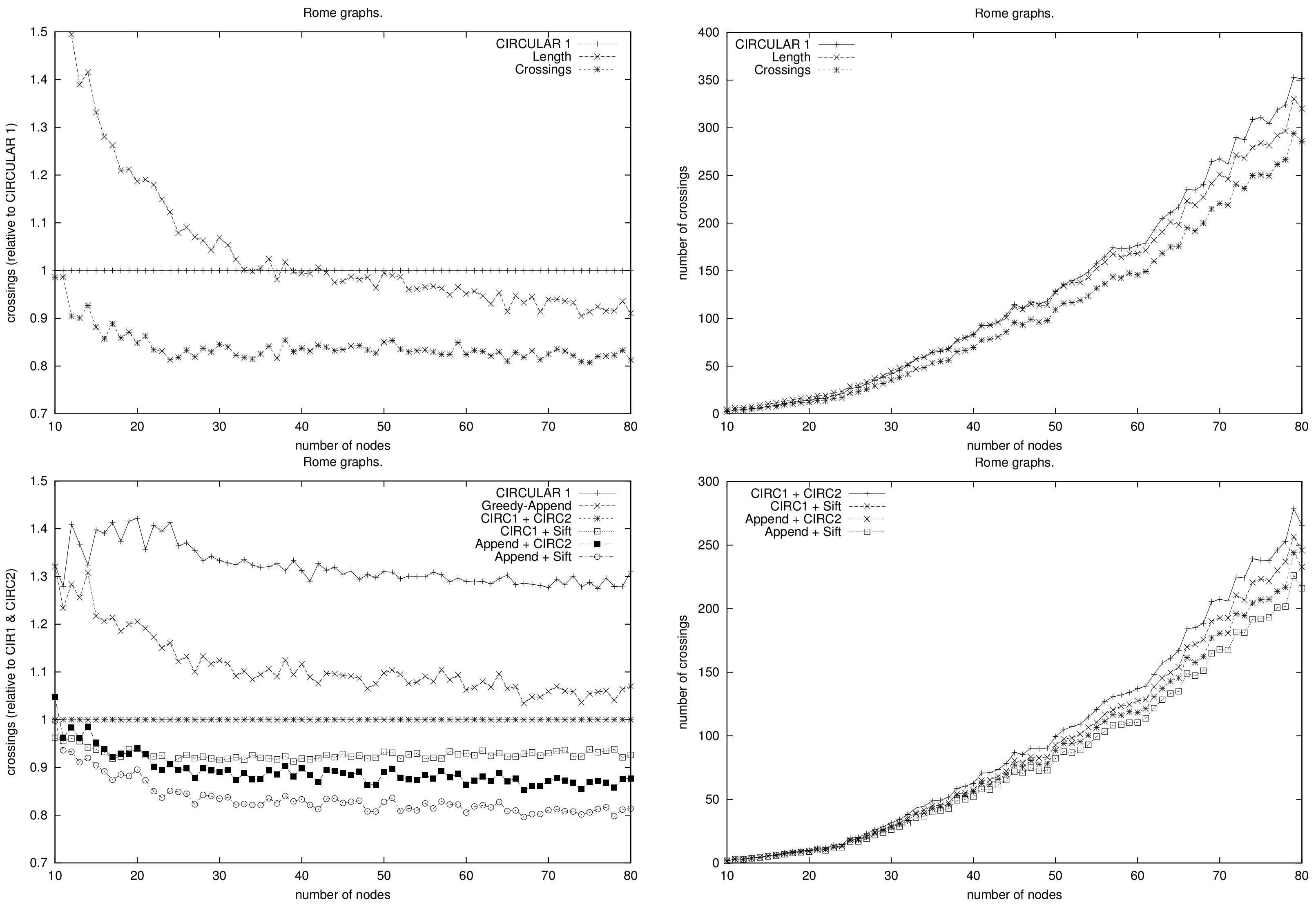

Fig. 7. Results on the "Rome graphs", a commonly used benchmark data set

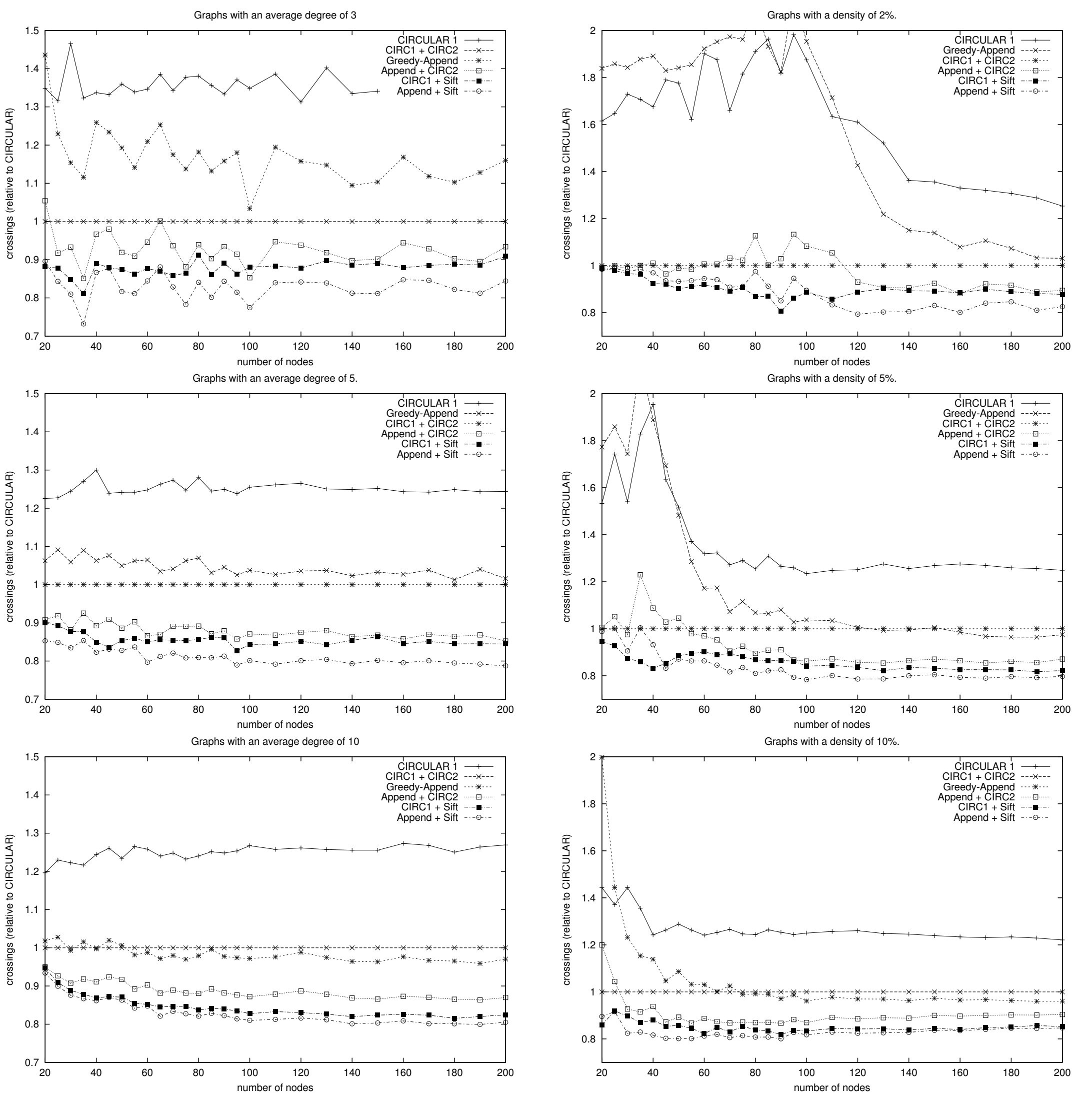

Fig. 8. Results on random graphs relative to CIRCULAR

14. J. M. Six and I. G. Tollis. Circular Drawings of Biconnected Graphs. In *Selected Papers from the 1st International Workshop on Algorithm Engineering and Experimentation (ALENEX'99)*, volume 1619 of *Lecture Notes in Computer Science*, pages 57–73. Springer, 1999. 1, 2, 6