A Fast and Simple Heuristic for Constrained Two-Level Crossing Reduction

Michael Forster

University of Passau, 94030 Passau, Germany
forster@fmi.uni-passau.de

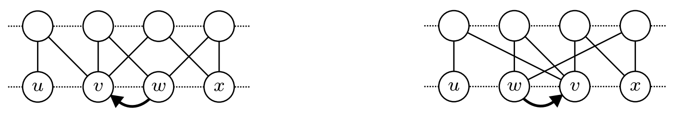
Abstract. The one-sided two-level crossing reduction problem is an important problem in hierarchical graph drawing. Because of its NP-hardness there are many heuristics, such as the well-known barycenter and median heuristics. We consider the constrained one-sided two-level crossing reduction problem, where the relative position of certain vertex pairs on the second level is fixed. Based on the barycenter heuristic, we present a new algorithm that runs in quadratic time and generates fewer crossings than existing simple extensions. It is significantly faster than an advanced algorithm by Schreiber [12] and Finnocchi [1, 2, 6], while it compares well in terms of crossing number. It is also easy to implement.

1 Introduction

The most common algorithm for drawing directed acyclic graphs is the algorithm of Sugiyama, Tagawa, and Toda [13]. The vertex set is partitioned into parallel horizontal levels such that all edges point downwards. For every intersection between an edge and a level line, a dummy vertex is introduced that may later become an edge bend. In a second phase, a permutation of the vertices on each level is computed that minimizes the number of edge crossings. Finally, horizontal coordinates are computed, retaining the vertex order on each level.

A small number of crossings is very important for a drawing to be understandable. Thus, the crossing reduction problem is well studied. The minimization of crossings is NP-hard [4, 8], and many heuristics exist for crossing reduction. Most of them reduce the problem to a sequence of one-sided two-level crossing reduction problems. Starting with an arbitrary permutation of the first level, a permutation of the second level is computed that induces a small number of edge crossings between the first two levels. Then the permutation of the second level is fixed and the third level is reordered. This is repeated for all levels, alternately top down and bottom up, until some termination criterion is met.

A simple and efficient heuristic for the one-sided two-level crossing reduction problem is the barycenter heuristic. For every vertex v on the second level, its barycenter value $b(v)$ is defined as the arithmetic mean of the relative positions of its neighbors $N(v)$ on the first level $b(v) = \frac{1}{|N(v)|} \sum_{v \in N(v)} \mathrm{pos}(v)$. The vertices on the second level are then sorted by their barycenter value. In practice this strategy gives good results, while keeping the running time low. An alternative



(a) The constraint is violated (b) The constraint is satisfied

Fig. 1. The constrained crossing reduction problem

is the median heuristic, which works similar but uses median values instead of the barycenter. The median heuristic can be proven [3, 5] to miss the minimum number of crossings by a factor of at most three. However, in experimental results [9, 10] it is outperformed by the barycenter heuristic.

As a variant of the crossing reduction problem we consider the constrained one-sided two-level crossing reduction problem. In addition to the permutation of the first level, some pairs of vertices on the second level have a fixed relative position. Figure 1 shows a two-level graph with one *constraint* $c = (w, v)$, visualized by the bold arrow. The constraint means that its *source vertex* w must be positioned on the left of its *target vertex* v. In Fig. 1(a), the constraint is violated, and in Fig. 1(b) it is satisfied. Obviously, constraints may increase the number of crossings, in this case from two to five.

Formally, an instance of the constrained one-sided two-level crossing reduction problem consists of a two-level graph $G = (V_1, V_2, E)$, $E \subseteq V_1 \times V_2$ with a fixed permutation of the first level V_1 and a set $C \subseteq V_2 \times V_2$ of constraints. It is our objective to find a permutation of the vertices on the second level V_2 with few edge crossings and all constraints satisfied. Clearly, this problem is NP-hard as well. A solution only exists if the *constraint graph* $G_C = (V_2, C)$ is acyclic.

While the constrained crossing reduction problem has many direct practical applications, it also appears as a subproblem in other graph drawing problems. An example is the application of the Sugiyama algorithm to graphs with vertices of arbitrary size [12] or to clustered graphs [7]. When vertices or clusters span multiple levels, constraints can be used to prevent overlap. Another application is preserving the mental map when visualizing a sequence of related graphs.

This paper is organized as follows: We survey existing approaches for the constrained two-level crossing reduction problem in the next section. In Sect. 3 we present our heuristic and prove its correctness in Sect. 4. Section 5 gives experimental results that compare our heuristic to the existing algorithms. We close with a short summary in Sect. 6.

2 Previous Work

The constrained crossing reduction problem has been considered several times. Sander [11] proposes a simple strategy to extend iterative two-level crossing reduction algorithms to handle constraints. Starting with an arbitrary admissible vertex permutation, updates are only executed if they do not violate a constraint. Together with the barycenter heuristic a modified sorting algorithm is used: The positions of two vertices are only swapped, if no constraint is violated. Waddle [14] presents a similar algorithm. After the calculation of the barycenter values it is checked for each constraint whether its target has a lower barycenter value than its source. In that case the constraint would be violated after sorting the vertices by the barycenter values. To avoid this, the barycenter value of the source vertex is changed to the barycenter value of the target vertex plus some small value. The result of both heuristics is a vertex permutation that satisfies all constraints. However, the extensions are rather restrictive and often prevent the algorithm from finding a good permutation. Accordingly, the results are significantly worse than in graphs without constraints.

Schreiber [12] and Finnocchi [1, 2, 6] have independently presented an advanced algorithm that considers constraints and crossing minimization simultaneously. Their main idea is to reduce the constrained crossing reduction problem to the weighted feedback arc set problem, which is also NP-hard [3]. First the so-called *penalty graph* is constructed. Its vertices are the vertices of the second level. For each pair (u, v) of vertices the number of crossings in the two relative orders of u and v is compared. For this, only edges incident to u or v are considered. If the number of crossings c_{uv} in the relative order $\ldots, u, \ldots, v, \ldots$ is less than the number of crossings c_{vu} in the reverse order $\ldots, v, \ldots, u, \ldots$, then an edge $e = (u, v)$ with weight $w(e) = c_{vu} - c_{uv}$ is inserted. Constraints are added as edges with infinite (or very large) weight. Figure 2 shows the penalty graph of the two-level graph in Fig. 1.

Then a heuristic for the weighted feedback arc set problem is applied to the penalty graph. It is important that the used heuristic guarantees that the edges with infinite weight are not reversed, or constraints may be violated. Finally, the vertices of the now acyclic penalty graph are sorted topologically, and the resulting permutation defines the order of the second level. If no edges had to be reversed, the number of crossings meets the obvious lower bound $c_{\min} = \sum_{u,v \in V} \min\{c_{uv}, c_{vu}\}$. Each reversed edge e increments the number of crossings by its weight. This implies that an optimal solution of the weighted feedback arc set problem is also optimal for the constrained crossing reduction problem.

Comparing the approaches of Sander [11] and Waddle [14] with those of Schreiber [12] and Finnocchi [1, 2, 6] shows a direct trade-off between quality and execution time. Schreiber presents detailed experimental results which show that the penalty graph approach generates significantly less crossings than the barycenter heuristic extensions. This is especially evident, if there are many constraints. The running times, however, are considerably higher. This is not very surprising due to the $O(|V_2|^4 + |E|^2)$ time complexity.

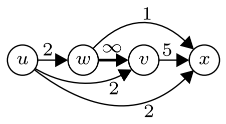


Fig. 2. The penalty graph of Fig. 1

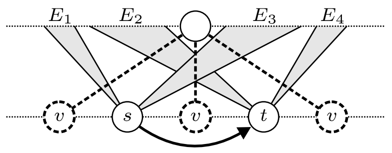
3 A modified barycenter heuristic

The goal of our research is to build an algorithm that is as fast as the existing barycenter extensions while delivering a quality comparable to the penalty graph approach. To achieve this we use a new extension of the barycenter heuristic. We could have used the median heuristic as well, but we did not, because it is experimentally worse and in our algorithm median values are more difficult to handle.

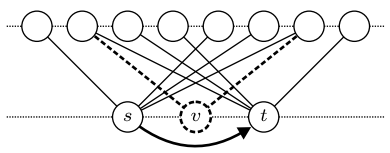
We start by computing the barycenter values of all vertices. As long as the source of each constraint has a lower barycenter value than the target, all constraints are satisfied automatically. In the reverse case the permutation has to be corrected. In this context, we call a constraint $c = (s, t)$ *satisfied* if $b(s) < b(t)$ and *violated* otherwise.

Our algorithm is based on a simple assumption: If a constraint is violated as in Fig. 3(a), the greater barycenter value of the source vertex indicates more edges "to the right" than "to the left", $|E_3| > |E_1|$. The inverse is true for the target vertex, $|E_4| < |E_2|$. In this situation we assume that in the corrected permutation no other vertices should be positioned in between. This seems plausible, because between s and t larger subsets of adjacent edges have to be crossed than outside. Using median values it can be proven that for a vertex with only one incident edge there is always an optimal position beyond any violated constraint. This is not generally true, however, for vertices of higher degree or for the barycenter heuristic as Fig. 3(b) shows. The optimal position for vertex v is in the middle, where its edges generate 6 crossings as opposed to 8 crossings at the other two positions. Nevertheless, adopting the assumption is justified by good experimental results presented in Sect. 5.

Our heuristic, shown in Algorithm 1, partitions the vertex set V_2 into totally ordered vertex lists. Initially there is one singleton list $L(v) = \langle v \rangle$ per vertex v. In the course of the algorithm these lists are pairwise concatenated into longer lists according to violated constraints. Concatenated lists are represented by new dummy vertices and associated barycenter values. As long as there are violated constraints, each violated constraint $c = (s, t)$ is removed one by one and the lists containing s and t are concatenated in the required order. They are then treated as a cluster of vertices. This guarantees that the constraint is satisfied



(a) Vertices with a single edge should not be positioned between the vertices of a violated constraint ($b(s) > b(t)$).



(b) In general, the optimal position for a vertex may be between the vertices of a violated constraint.

Fig. 3. The Basic Assumption of Our Algorithm

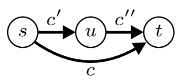
Algorithm 1: CONSTRAINED-CROSSING-REDUCTION

```
Input: A two-level graph $G = (V_1, V_2, E)$ and acyclic constraints $C \subseteq V_2 \times V_2$
Output: A permutation of $V_2$
begin
1      foreach $v \in V_2$ do
2          $b(v) \leftarrow \sum_{u \in N(v)} \mathrm{pos}(u) / \deg(v)$          //barycenter of $v$
3          $L(v) \leftarrow \langle v \rangle$                                   //new singleton list
4      $V \leftarrow \{\, s, t \mid (s,t) \in C \,\}$                            //constrained vertices
5      $V' \leftarrow V_2 - V$                                                   //unconstrained vertices
6      while $(s,t) \leftarrow$ FIND-VIOLATED-CONSTRAINT$(V, C) \neq \perp$ do
7          create new vertex $v_c$
8          $\deg(v_c) \leftarrow \deg(s) + \deg(t)$                              //update barycenter value
9          $b(v_c) \leftarrow \big(b(s) \cdot \deg(s) + b(t) \cdot \deg(t)\big) / \deg(v_c)$
10         $L(v_c) \leftarrow L(s) \circ L(t)$                                   //concatenate vertex lists
11         forall $c \in C$ do
12             if $c$ is incident to $s$ or $t$ then
13                 make $c$ incident to $v_c$ instead of $s$ or $t$
14         $C \leftarrow C - \{(v_c, v_c)\}$                                     //remove self loops
15         $V \leftarrow V - \{s, t\}$
16         if $v_c$ has incident constraints then $V \leftarrow V \cup \{v_c\}$
17         else $V' \leftarrow V' \cup \{v_c\}$
18     $V'' \leftarrow V \cup V'$
19     sort $V''$ by $b()$
20     $L \leftarrow \langle\rangle$                                            //concatenate vertex lists
21     foreach $v \in V''$ do
22         $L \leftarrow L \circ L(v)$
23     return $L$
end
```

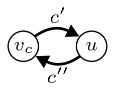
but prevents other vertices from being placed between s and t. Following our assumption, this does no harm. A new vertex v_c replaces s and t to represent the concatenated list $L(v_c) = L(s) \circ L(t)$. The barycenter value of v_c is computed as if all edges that are incident to a vertex in $L(v_c)$ were incident to v_c. This can be done in constant time as demonstrated in lines 8 and 9 of the algorithm. Note that this is not doable for the median value.

When no violated constraints are left, the remaining vertices and vertex lists are sorted by their barycenter value as in the standard barycenter heuristic. The concatenation of all vertex lists results in a vertex permutation that satisfies all constraints. We claim that it has few crossings as well.

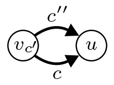
For the correctness of the algorithm it is important to consider the violated constraints in the right order. In Fig. 4 the constraints are considered in the



(a) Before the merge all constraints are satisfiable by the given order. Let c be violated.



(b) After merging s and t the generated constraint cycle makes it impossible to satisfy all constraints.



(c) Starting with c' leads to a correct result.

Fig. 4. Considering constraints in the wrong order

wrong order and c is processed first. This leads to a cycle in the resulting constraint graph which makes it impossible to satisfy all remaining constraints, although the original constraint graph was acyclic. If c is violated, at least one of the other constraints is also violated. Processing this constraint first leads to a correct result.

Thus, we must avoid generating constraint cycles. We use a modified topological sorting algorithm on the constraint graph. The constraints are considered sorted lexicographically by the topsort numbers of the target and source vertices in ascending and descending order, respectively. Using Algorithm 2 this traversal can be implemented in $O(|C|)$ time. The vertices are traversed in topological order. The incoming constraints of a vertex t are stored in an ordered list $I(t)$ that is sorted by the reverse traversal order of the source vertices. If a traversed vertex has incoming violated constraints, the topological sorting is cancelled and the first of them is returned. Note that the processing of a violated constraint can lead to newly violated constraints. Thus, the traversal must be restarted for every violated constraint.

4 Theoretical Analysis

In this section we analyse the correctness and running time of our algorithm. For the correctness we have to show that the vertex permutation computed by our algorithm satisfies all constraints. We start by analyzing Algorithm 2:

Lemma 1. *Let $c = (s, t)$ be a constraint returned by Algorithm 2. Then merging of s and t does not introduce a constraint cycle of two or more constraints.*

Proof. Assume that merging of s and t generates a cycle of at least two constraints. Because there was no cycle before, the cycle corresponds to a path p in G_C from s to t with a length of at least two. Because of the specified constraint traversal order, any constraint in p has already been considered, and thus is satisfied. This implies that $b(t) > b(s)$, and therefore contradicts the assumption. □

Algorithm 2: FIND-VIOLATED-CONSTRAINT

```
Input: An acyclic constraint graph $G_C = (V, C)$ without isolated vertices
Output: A violated constraint $c$, or $\perp$ if none exists

begin
1       $S \leftarrow \emptyset$                                //active vertices
2       foreach $v \in V$ do
3           $I(v) \leftarrow \langle\rangle$                    //empty list of incoming constraints
4           if $\mathrm{indeg}(v) = 0$ then
5               $S \leftarrow S \cup \{v\}$                     //vertices without incoming constraints
6       while $S \neq \emptyset$ do
7           choose $v \in S$
8           $S \leftarrow S - \{v\}$
9           foreach $c = (s, v) \in I(v)$ in list order do
10              if $b(s) \geq b(v)$ then
11                  return c
12          foreach outgoing constraint $c = (v, t)$ do
13              $I(t) \leftarrow \langle c\rangle \circ I(t)$
14              if $|I(t)| = \mathrm{indeg}(t)$ then
15                  $S \leftarrow S \cup \{t\}$
16      return $\perp$
end
```

Theorem 1. *The permutation computed by Algorithm 1 satisfies all constraints.*

Proof. Algorithm 1 maintains the invariant that the constraint graph is acyclic. Because of Lemma 1 no nontrivial cycles are introduced, and self loops are explicitly removed in line 14.

Next we analyse whether the removed self loop constraints are satisfied by the algorithm. Any such self loop c' has been generated by the lines 11–13 from a constraint between s and t. Because of the constraint $c = (s, t)$, the invariant implies that c' was not directed from t to s. Therefore, $c' = (s, t)$ is explicitly satisfied by the list concatenation in line 10.

Each remaining constraint has not been returned by Algorithm 2. Thus, the barycenter value of its source vertex is less than that of its target vertex. Then the constraint is satisfied by line 19. □

The rest of this section analyses the running time of our algorithm. Again, we start with the analysis of Algorithm 2.

Lemma 2. *Algorithm 2 runs in $O(|C|)$ time.*

Proof. The initialization of the algorithm in lines 1–5 runs in $O(|V|)$ time. The while-loop is executed at most $|V|$ times. The nested foreach-loops are both executed at most once per constraint. The sum of these time bounds is $O(|V| + |C|)$. Because the constraint graph does not contain isolated vertices, the overall running time of the algorithm is bounded by $O(|C|)$. $\square$

Theorem 2. *Algorithm 1 runs in* $O(|V_2| \log |V_2| + |E| + |C|^2)$ *time.*

Proof. The initialization of the algorithm in lines 1–3 considers every vertex and edge once and therefore needs $O(|V_2| + |E|)$ time. The while-loop is executed at most once per constraint. It has an overall running time of $O(|C|^2)$ because the running time of one loop execution is bounded by the $O(|C|)$ running time of Algorithm 2. Finally, the sorting in line 19 needs $O(|V_2| \log |V_2|)$ time. The sum of these time bounds is $O(|V_2| \log |V_2| + |E| + |C|^2)$. All other statements of the algorithm do not increase the running time. $\square$

5 Experimental Analysis

To analyse the performance of our heuristic, we have implemented both our algorithm and the penalty graph approach in Java. We have tested the implementations using a total number of 37,500 random graphs: 150 graphs for every combination of the following parameters: $|V_2| \in \{50, 100, 150, 200, 250\}$, $|E|/|V_2| \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$, $|C|/|V_2| \in \{0, 0.25, 0.5, 0.75, 1.0\}$.

Figure 5 displays a direct comparison. The diagrams show, how the results vary, when one of the three parameters is changed. Because the number of crossings grows very fast in the number of edges, we do not compare absolute crossing numbers, but the number of crossings divided by the number of crossings before the crossing reduction. As expected, the penalty graph approach gives strictly better results than our heuristic. But the graphs also show that the difference is very small. For a more detailed comparison, we have also analyzed the quotient of the crossing numbers in Fig. 6. These graphs show that our algorithm is never more than 3% worse than the penalty graph approach. Mostly the difference is below 1%. Only for very sparse graphs there is a significant difference.

This is a very encouraging result, considering the running time difference of both algorithms: Figure 7 compares the running time of the algorithms. As expected, our algorithm is significantly faster than the penalty graph approach. Because of the high running time of the penalty graph approach we have not compared the algorithms on larger graphs, but our algorithm is certainly capable of processing larger graphs. For example, graphs with $|V_2| = 1000$, $|E| = 2000$, and $|C| = 500$ can be processed in less than a second, although our implementation is not highly optimized.

6 Summary

We have presented a new fast and simple heuristic for the constrained one-sided two-level crossing reduction problem. In practice, the algorithm delivers nearly the same quality as existing more complex algorithms, while its running time is significantly better. For further improvement, a traversal of the violated constraints is desired that runs faster than $O(|C|^2)$.

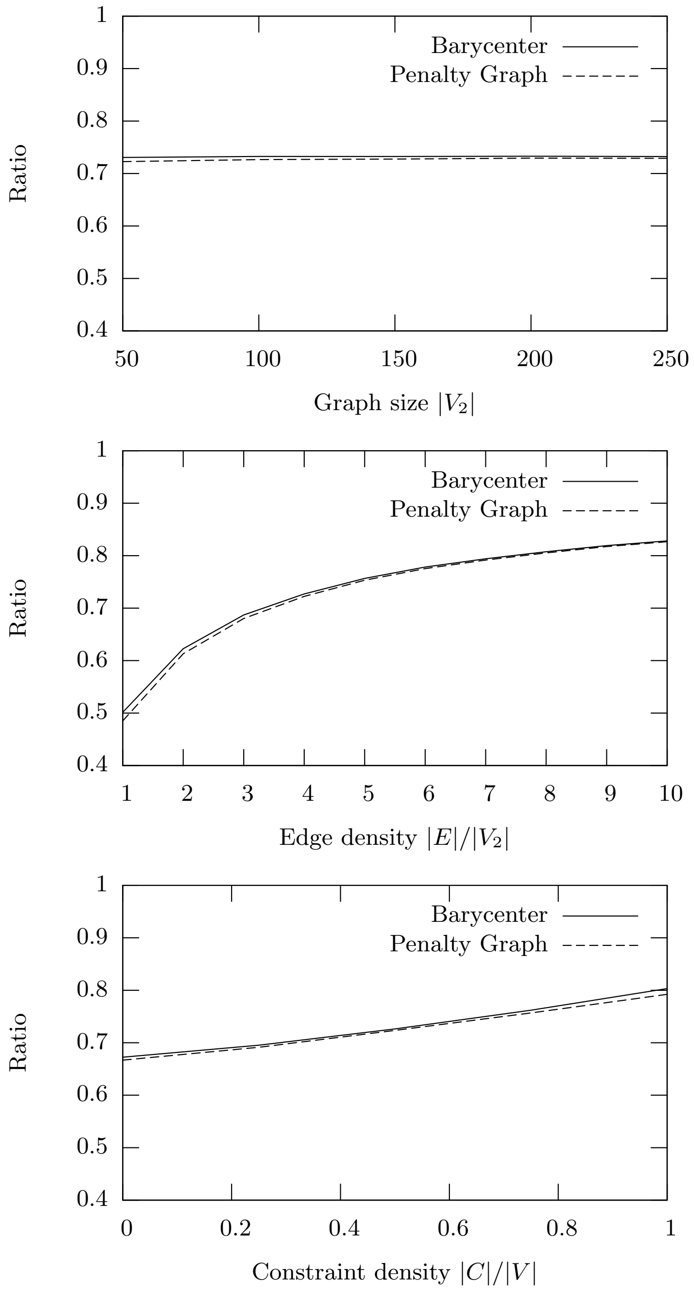


Fig. 5. The ratio of crossings before and after crossing reduction. Lesser values are better

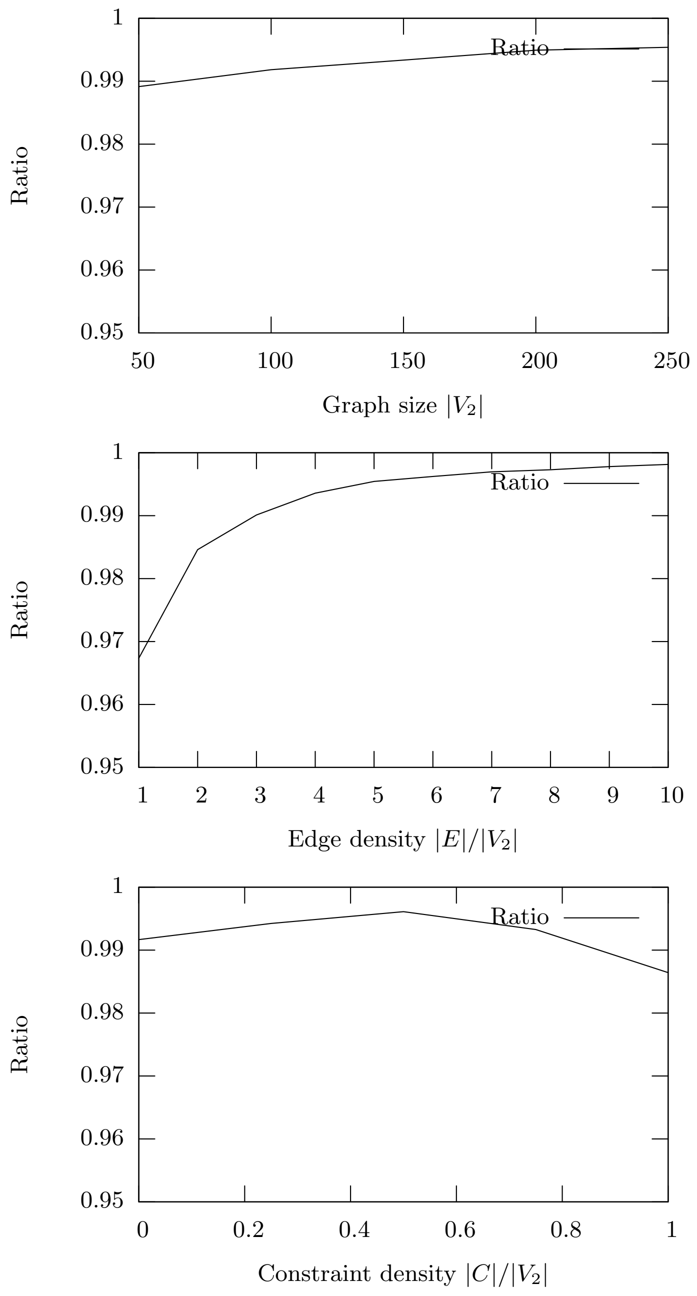


Fig. 6. Number of crossings using the penalty graph approach divided by the number of crossings using the extended barycenter approach

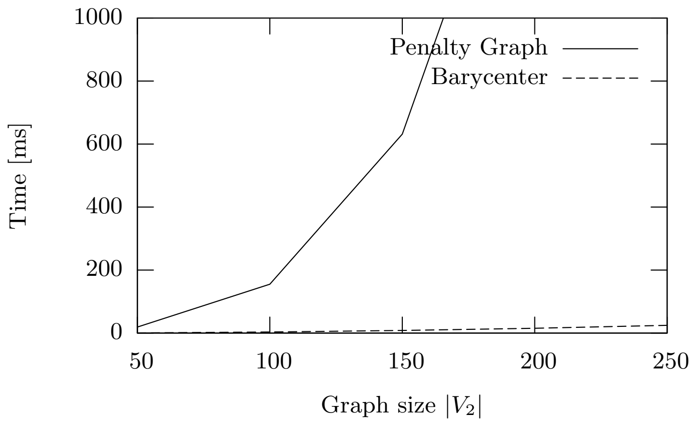


Fig. 7. Running time comparison

References

1. C. Demetrescu and I. Finocchi. Break the "right" cycles and get the "best" drawing. In B. Moret and A. Goldberg, editors, *Proc. ALENEX'00*, pp. 171–182, 2000.
2. C. Demetrescu and I. Finocchi. Removing cycles for minimizing crossings. *ACM Journal on Experimental Algorithmics (JEA)*, 6(1), 2001.
3. G. di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice Hall, 1999.
4. P. Eades, B. D. McKay, and N. C. Wormald. On an edge crossing problem. In *Proc. ACSC'86*, pp. 327–334. Australian National University, 1986.
5. P. Eades and N. C. Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11:379–403, 1994.
6. I. Finocchi. Layered drawings of graphs with crossing constraints. In J. Wang, editor, *Proc. COCOON'01*, volume 2108 of *LNCS*, pp. 357–368. Springer, 2001.
7. M. Forster. Applying crossing reduction strategies to layered compound graphs. In S. G. Kobourov and M. T. Goodrich, editors, *Proc. GD'02*, volume 2528 of *LNCS*, pp. 276–284. Springer, 2002.
8. M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 4(3):312–316, 1983.
9. M. Jünger and P. Mutzel. Exact and heuristic algorithms for 2-layer straightline crossing minimization. In F. J. Brandenburg, editor, *Proc. GD'95*, volume 1027 of *LNCS*, pp. 337–348. Springer, 1996.
10. M. Jünger and P. Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *JGAA*, 1(1):1–25, 1997.
11. G. Sander. *Visualisierungstechniken für den Compilerbau.* PhD thesis, University of Saarbrücken, 1996.
12. F. Schreiber. *Visualisierung biochemischer Reaktionsnetze.* PhD thesis, University of Passau, 2001.
13. K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Trans. SMC*, 11(2):109–125, 1981.
14. V. Waddle. Graph layout for displaying data structures. In J. Marks, editor, *Proc. GD'00*, volume 1984 of *LNCS*, pp. 241–252. Springer, 2001.