Upward Planarity Testing: A Computational Study

Markus Chimani[1] and Robert Zeranski[2]

[1] Theoretical Computer Science, Osnabrück University, Germany
[2] Algorithm Engineering, Friedrich-Schiller-University Jena, Germany
markus.chimani@uni-osnabrueck.de, robert.zeranski@uni-jena.de

Abstract. A directed acyclic graph (DAG) is *upward planar* if it can be drawn without any crossings while all edges—when following them in their direction—are drawn with strictly monotonously increasing y-coordinates. Testing whether a graph allows such a drawing is known to be NP-complete, but there is a substantial collection of different algorithmic approaches known in literature.

In this paper, we give an overview of the known algorithms, ranging from combinatorial FPT and branch-and-bound algorithms to ILP and SAT formulations. Most approaches of the first class have only been considered from the theoretical point of view, but have never been implemented. For the first time, we give an extensive experimental comparison between virtually all known approaches to the problem.

Furthermore, we present a new SAT formulation based on a recent theoretical result by Fulek et al. [8], which turns out to perform best among all known algorithms.

1 Introduction

When drawing directed graphs, in particular DAGs, one often wants to make the edges' orientations clearly recognizable by having all edges pointing in the same general direction, w.l.o.g. upward. A *y-monotone* drawing is thus one, where the curves representing the edges have strictly monotonously increasing y-coordinates when traversing them from their source to the target vertices. More formally, a y-monotone edge intersects any horizontal line at most once, while its source vertex is drawn below its target vertex.

A second central concept in graph drawing is planarity, i.e., we want to avoid crossing edges if possible. The question of *upward planarity* of a DAG G is hence the question whether there exists a crossing-free y-monotone drawing of G. While (undirected) planarity is linear time solvable, upward planarity turns out to be NP-complete to decide [9]. Nonetheless, due to the problem's centrality, several exponential-time algorithms have been developed.

A core result is that the problem becomes polynomial time solvable if the graph's embedding (i.e., the order of the edges around their vertices) is fixed [1]. Based thereon, the historically first algorithms are fixed-parameter tractable (FPT) algorithms where the parameters are essentially bounding the (in general

S. Wismath and A. Wolff (Eds.): GD 2013, LNCS 8242, pp. 13–24, 2013.
© Springer International Publishing Switzerland 2013

exponential) number of possible embeddings; the algorithms are testing upward planarity for each possible embedding [10]. The process can be sped up using a polynomial time algorithm to solve the important special case of series-parallel graphs [7]. The special case of single-source DAGs is also polynomial time solvable [1, 5, 11, 13] but not the focus of this paper.

A different approach is based on relaxing the upward requirement (*quasi-upward planarity* [2], see below) and considering the optimization problem to minimize the violating edges. There, the embedding enumeration is coupled with a sophisticated method to obtain upper and lower bounds for partially embedded graphs, allowing for a branch-and-bound algorithm. Finally, the most recent approach [3] is to formulate the problem as an integer linear program (ILP) or boolean satisfiability problem (SAT), to be solved with a corresponding solver.

In Section 2, we summarize the core ideas of these algorithms. We also present a *new* SAT formulation based on a recent theoretical result by Fulek et al. [8].

Before this paper, the only reported implementations were for the branch-and-bound and the ILP/SAT approach. The former implementation is in fact considering the more general optimization problem (instead of the decision version), and both implementations are based on two different underlying libraries. This made a direct comparison worrisome. In this paper (Section 3), we report on our consistent implementations of all the discussed algorithms. They share as much code as was feasibly possible, to maximize the fairness of the comparison. Hence, we are for the first time able to make substantiated claims about the algorithms' respective applicability in practice.

2 Algorithms

We always consider a DAG $G = (V, E)$ to test for upward planarity. A *combinatorial* embedding of G is specified by cyclically ordering the edges around their incident vertices. A *planar* embedding additionally chooses an external face.

2.1 FPT Algorithms

A fixed-parameter tractable (FPT) algorithm, with respect to some parameter k, is an algorithm with running time $\mathcal{O}(f(k)\cdot poly(n))$ where $poly(n)$ is a polynomial function in the size of the input (here, $n := |G|$), and $f(k)$ is any computable function (typically an exponential function) only dependent on k. A central ingredient of all known combinatorial FPT algorithms is the following result [1]:

Theorem 1 (Bertalozzi, Di Battista, Liotta, Mannino). *Let $G = (V, E)$ be an* embedded *DAG. There is an algorithm testing whether this embedding of G allows an upward planar drawing in $\mathcal{O}(n^2)$ time.*

Let G be planar and biconnected. We can decompose the underlying undirected graph into its triconnectivity components in linear time. These can be efficiently organized as an *SPQR-tree* [6]. For notational simplicity, we may talk about an *SPR-tree* (as Q-nodes, representing single edges, are not necessary):

The SPR-tree $\mathcal{T}(G)$ is a tree with three kinds of *nodes*: S- and P-nodes represent serial and parallel components, respectively; R-nodes represent planar triconnected components. We call these components the *skeletons* associated to the nodes. An edge in a skeleton S may be real or *virtual*; in the latter case, it represents a subgraph, described by the subtree attached to S's node in $\mathcal{T}(G)$.

Embedding Enumeration (EE). The natural approach to test upward planarity of an unembedded graph is to test every possible embedding of G, using the algorithm of Theorem 1. As the number of embeddings is, in general, exponential in the size of the graph, one has to seek for a meaningful parameter to bound the number of embeddings. The SPR-tree can be used to efficiently enumerate all possible embeddings of a graph. We can bound the number of embeddings by $\mathcal{O}(t! \cdot 2^t)$, where t is the number of nodes in our SPR-tree, which leads to an overall running time of $\mathcal{O}(t! \cdot 2^t \cdot n^2)$ to test upward planarity [10].

In the same publication, a kernelization algorithm for sparse (not necessarily biconnected) graphs is presented. Using the following preprocessing steps (until none is applicable anymore), leaves a graph with at most $30k^2 + 2k$ vertices and at most $(2k + 1)!$ embeddings; thereby, $k = |E| - |V|$ is the number of edges (minus 1) the (preprocessed) graph has more than a tree. Let a *chain* be a path in G where all inner vertices have degree 2, The preprocessing steps are: (PP1) remove vertices of degree 1; (PP2) replace chains where each inner vertex v has $\mathit{indeg}(v) = \mathit{outdeg}(v) = 1$ by single correspondingly oriented edges; (PP3) remove chains where both end vertices coincide; (PP4) for each set of *parallel* chains, remove all but one chain (parallel chains are those that have a common start vertex, a common end vertex, and an identical sequence of edge orientations along the chain). This preprocessing requires $\mathcal{O}(n^2)$ time.

After the preprocessing steps, again, all embeddings are tested in overall $\mathcal{O}(n^2 + k^4 \cdot (2k + 1)!)$ time. Observe that (PP1)–(PP4) are valid in general (although one has to specifically consider the case of biconnectivity-breaking PP4). Hence, when testing all embeddings after preprocessing, we in fact obtain an algorithm—denoted by EE in the following—with running time $\mathcal{O}(\min(t! \cdot 2^t \cdot n^2, n^2 + k^4 \cdot (2k + 1)!))$ for biconnected DAGs.

Upward Spirality (SPIR). Consider the SPR-tree rooted at some arbitrary node. Informally, *upward spirality* is a measure of how much a skeleton is "rolled up" around its poles (the end nodes of the virtual edge representing the node's parent). Furthermore, one has to distinguish several *pole categories*, i.e., local properties of the embedding around the pole vertices. For details of the definitions and the following algorithms cf. [7].

For series-parallel graphs, upward spirality allows to develop a polynomial dynamic programming algorithm that traverses the SPR-tree bottom up—recall that for such graphs it only has S- and P-nodes. At each node μ, we store a set of feasible (spirality/pole category)-pairs to upward embed the graph encoded by the SPR-tree rooted at μ. This information can then be used to obtain a corresponding set for the parent node, etc. We denote this algorithm by SPIR-sp. Its running time is $\mathcal{O}(n^4)$, but there are large constants hidden in the $\mathcal{O}$-notation, cf. Section 3.1.

Using this algorithm as a building block, one can establish an FPT algorithm for general DAGs, where the different configurations of the R-nodes w.r.t. each other have to be enumerated. This leads to a running time of $\mathcal{O}(d^r n^3 + dr^2 n + d^2 n^2)$ where d is the largest diameter of any skeleton and r is the number of R-nodes. We call this algorithm `SPIR`.

2.2 Branch-and-Bound via Quasi-Upward Planarity (`BB`)

In a *quasi-upward planar* drawing, we relax upwardness such that each edge only has to be drawn y-monotonously within an arbitrary small neighborhood of its incident vertices. In [2], a branch-and-bound algorithm is established which produces a quasi-upward drawing maximizing the number of fully y-monotone edges: For a given embedding, the minimum number of non-y-monotone edges can be computed in $\mathcal{O}(n^2 \log n)$ time using minimum cost-flow techniques.

Now, we can consider all possible embeddings of the graph via the SPR-tree. At any moment we have a *partial* embedding—several embeddings are fixed while the others are free. The algorithm in [2] is able to compute upper and lower bounds for the number of non-y-monotone edges in this case. If the current lower bound is worse than the global upper bound, we can avoid testing all embeddings further down in the search tree, which have the same fixed embeddings in common.

Observe that a DAG is upward planar iff there is a quasi-upward planar drawing where all edges are y-monotone. Hence, when testing upward planarity, we can prune a partial embedding in the search tree whenever we obtain a lower bound strictly greater than 0. We denote this algorithm by `BB`. Formally, this algorithms could also be considered an FPT algorithm with the same worst-case running time as `EE`.

2.3 SAT Formulations

A SAT formulation of a decision problem instance $\mathcal{I}$ is a propositional logic formula that is satisfiable if and only if $\mathcal{I}$ has the answer *true*. The formula is typically given in conjunctive normal form, i.e., as a set of clauses, each of which has to be satisfied. Each clause is a disjunction of (possibly negated) variables. For the sake of readability, we will provide *rules* as propositional formulae; it is straight-forward to transform them into their corresponding clauses.

Ordered Embeddings (`OE`). An edge e *dominates* an edge f if there is a directed path (possibly of length 0) from e's target to f's source vertex in G. Clearly, f has to be drawn above e in any upward drawing. A pair of edges is *non-dominating* if neither edge dominates the other. Let $\mathcal{N}$ denote the set of all non-dominating edge pairs of G.

In [3], a SAT formulation based on *ordered embeddings* has been proposed. We consider a strict total (vertical) order of the vertices together with a strict partial (horizontal) order of the edges; edges are comparable w.r.t. this order iff they are non-dominating each other. We model the vertical order by introducing

boolean variables $\tau(v,w)$ for each proper pair of vertices v and w. Intuitively, $\tau(v,w) = \texttt{true}$ means that v is drawn *below* w. Since the vertex order is to be strict, $\tau(v,w) = \texttt{false}$ means that v is *above* w. We may use the shorthand $\tau(w,v) := \neg\tau(v,w)$ for notational simplicity. To establish a strict total order, it then suffices to require transitivity via $(\mathrm{R}^{\mathrm{t}}_{\tau})$. The *upward rules* $(\mathrm{R}^{\mathrm{u}})$ ensure that all edges are drawn upward.

$$\tau(u,v) \wedge \tau(v,w) \rightarrow \tau(u,w) \qquad \forall \text{ pairwise distinct } u,v,w \in V. \qquad (\mathrm{R}^{\mathrm{t}}_{\tau})$$

$$\tau(v,w) = \texttt{true} \qquad \forall\ (v,w) \in E. \qquad (\mathrm{R}^{\mathrm{u}})$$

Similarly, we can establish the horizontal order of the edges by introducing variables $\sigma(e,f)$ for each pair $\{e,f\} \in \mathcal{N}$. Thereby (if both edges are vertically overlapping), $\sigma(e,f) = \texttt{true}$ implies that e is to the *left* of f, and the satisfied shorthand $\sigma(f,e) := \neg\sigma(e,f)$ implies that e is to the *right* of f. Again, we simply require transitivity:

$$\sigma(e,f) \wedge \sigma(f,g) \rightarrow \sigma(e,g) \qquad \forall\ \{e,f\},\{f,g\},\{e,g\} \in \mathcal{N}. \qquad (\mathrm{R}^{\mathrm{t}}_{\sigma})$$

Based on this (upward) order system, we can establish planarity using surprisingly simple *planarity rules*: We only have to ensure that two adjacent edges e and f are on the same side of g (non-incident to the common vertex of e and f) if they both vertically overlap with g. Let $e \Cap f$ denote the common vertex of two edges e,f, and $\mathcal{P} := \{(e,f,g) \mid \{e,g\},\{f,g\} \in \mathcal{N} \wedge e \Cap f \notin g\}$ the set of edge-triplets as described above. We have:

$$\Big(\tau(x, e \Cap f) \wedge \tau(e \Cap f, y)\Big) \rightarrow \Big(\sigma(e,g) \leftrightarrow \sigma(f,g)\Big) \quad \forall\ (e,f,g = (x,y)) \in \mathcal{P}. \quad (\mathrm{R}^{\mathrm{p}})$$

The collection of the above rules allows a satisfying truth assignment to τ and σ if and only if G is upward planar [3]. Given such an assignment, it is trivial to construct the embedding in linear time. We denote this formulation by `OE`.

Hanani-Tutte type characterization (`FPSS`). The classical Hanani-Tutte theorem shows that a graph drawn such that all pairs of non-adjacent edges cross an even number of times is planar. A similar result has been established by Pach and Tóth [12], and was only recently improved upon by Fulek et al. [8]:

Theorem 2 (Pach, Tóth; Fulek, Pelsmajer, Schaefer, Štefankovič). *Let $G = (V,E)$ be a DAG. If G has a y-monotone*[1] *drawing such that every pair of non-adjacent edges crosses an even number of times, then there is a y-monotone planar embedding of G with the same location of vertices.*

To prove this theorem, [8] gives a quadratic time algorithm testing whether G allows a y-monotone drawing with prespecified vertex positions. This is essentially done by solving an equation system over (e,v)-*moves*. Such a move redraws the edge e by deforming it, close to the y-coordinate of v, into a horizontal "spike" that passes around v. There is a y-monotone embedding iff there

[1] In the original publications, these theorems were stated in terms of x-monotonicity.

is a set of (e, v)-moves turning a y-monotone drawing into a drawing in which every non-adjacent pair of edges crosses an even number of times. A simple set of only two equation classes suffices to describe all possible selections of (e, v)-moves that may lead to an even-crossing solution. These equations, in fact, resemble a 2SAT formulation (each clause has at most 2 literals), with the only prerequisite to know the vertical relationship between the DAG's vertices. This allows us to cast this powerful theoretical tool into a new SAT formulation not unlike `OE`:

We reuse the above boolean variables τ and rules $(\mathrm{R}^{\mathrm{t}}_{\tau}) \cup (\mathrm{R}^{\mathrm{u}})$ to guarantee an upward strict total order on V. We can then use these variables as indicators to activate or deactivate the above 2SAT-clauses to link move-variables. For every edge e and node v, we introduce a boolean variable $\varrho(e, v)$ that indicates whether we perform an (e, v)-move. Let s_e and t_e denote the start and target vertex of an edge e, respectively. We obtain

$$\Big(\tau(s_e, s_f) \wedge \tau(s_f, t_e) \wedge \tau(t_e, t_f)\Big) \rightarrow \Big(\varrho(e, s_f) \leftrightarrow \neg\varrho(f, t_e)\Big) \quad \forall\, e, f \in E \quad (\mathrm{R}^{\mathrm{m}}_0)$$

$$\Big(\tau(s_e, s_f) \wedge \tau(t_f, t_e)\Big) \rightarrow \Big(\varrho(e, s_f) \leftrightarrow \varrho(e, t_f)\Big) \quad \forall\, e, f \in E \quad (\mathrm{R}^{\mathrm{m}}_1)$$

where the subformulae after the implication are the clauses from the 2SAT suggested by [8], for the scenario described by the rules' left-hand sides.

Corollary 1. *Let $G = (V, E)$ be a DAG. G is upward planar if and only if the formula composed of rules $(\mathrm{R}^{\mathrm{t}}_{\tau}) \cup (\mathrm{R}^{\mathrm{u}}) \cup (\mathrm{R}^{\mathrm{m}}_0) \cup (\mathrm{R}^{\mathrm{m}}_1)$ is satisfiable.*

Hybrid formulations. Constructing an upward planar embedding (in polynomial time) using only a feasible truth assignment for `FPSS` is non-trivial, and in fact an open problem.[2] In order to obtain an embedding from the `FPSS` formulation, we consider two variants of hybridizing `FPSS` and `OE`, as extracting an embedding is trivial for the latter. Recall that both formulations use the same τ variables to establish a vertical order of the vertices. Hence we can simply put all rules together in one large formulation, denoted by `HF`.

Alternatively (denoted by `HL`), we can first compute a satisfying assignment to `FPSS`. If it exists, we can "learn" the subsolution for the τ variables to fix the τ variables in `OE` and solve the so-restricted `OE` to obtain a matching σ assignment.

3 Experiments

3.1 Considered Algorithms and Their Implementations

Overall, we consider the practical performances of the following algorithms:

FPT	b&b	SAT	ILP (see note below)
`EE`, `SPIR-sp`, (`SPIR`)	`BB`	`OE`, `FPSS`, `HF`, `HL`	`iOE`, `iFPSS`, `iHF`, `iHL`

[2] An algorithm achieving this is currently under development [personal communication with Marcus Schaefer]

All experiments were performed on an Intel Xeon E5520, 2.27 GHz, 8 GB RAM running Debian 6. We implemented the algorithms as part of the Open Graph Drawing Framework (`www.ogdf.net`), using *minisat* as our SAT solver and CPLEX 12 as our ILP solver. For both solvers, we used their default settings. For all considered instances (available at `www.cs.uos.de/theoinf`), we first performed the preprocessing steps (PP1)–(PP4), as described in Section 2.1.

A note on the ILPs. Given the SAT formulations, it is straight-forward to construct integer linear programs (using only binary variables) along the same lines, for which to test feasibility. This has been done for `OE` in [3], leading to a vastly weaker practical performance than the SAT approach. We note that `FPSS` (and the hybridizations) also allow such ILPs. We performed all the below experiments also for the ILP variants. However, also for `iFPSS` and the hybridizations, the pure feasibility testing functionality of ILP solvers—lacking sophisticated propagation and backtracking—is clearly too weak to allow compatible performance. We will hence disregard the ILP approaches in the following discussions.

A note on `BB`. The branch-and-bound algorithm has only been devised for biconnected graphs. Our implementation hence inherits this restriction. As described above, we strengthened `BB` for the special case of upward planarity testing by pruning any subproblems with a lower bound larger than 0.

In [2], the practical performance of an implementation (within GDToolkit) has been reported on a set of random instances, denoted by *BDD* in the following. Out of the originally 300 considered instances, only 200 are still available[3]. Of those, we can discard 139, as they are not DAGs. The largest remaining instances have 100 vertices and take 0.03 seconds on average (cf. Table 2). In [2]—on a clearly slower PC and not restricted to the pure upward planarity test!—the graphs with 100 vertices require roughly 100 seconds on average. This gives us a hint that our implementation is not vastly inefficient.

A note on `EE`. As `BB` is already restricted to biconnected graphs, we also restricted our implementation of `EE` to this case, to be able to efficiently generate all embeddings purely via the SPR-tree. Our implementation is able to enumerate ≈100,000 (bimodal) embeddings per second (disregarding any testing time).

A note on `SPIR-sp` and `SPIR`. The spirality-based algorithms are very theoretically demanding algorithms. We provide the seemingly first implementation of the polynomial case for series-parallel graphs (`SPIR-sp`). However, the algorithm's theoretical beauty is unfortunately not matched with practicability. E.g., when combining tables in the bottom-up dynamic programming, there is a very large number of possible combinations of choices to check. Although this number is bounded by a constant, we often need to check close to the theoretical worst-case of 2^{49} ($\approx 5.6 \times 10^{14}$) combinations. This constitutes the main bottleneck of the algorithm; further theoretical research may be able to bring down this vast number. In our implementation, we parallelized this checking via OpenMP to mitigate the effect. However, it remains to slow down the algorithm dramatically.—We will see experimental evidence in Figure 3 below.

[3] Personal communication with Walter Didimo.

Table 1. # of solved instances within given time frames in seconds (*Rome* and *North*)

time (s)	*Rome*					*North*			
	$[0, 0.01]$	$(0.01, 0.1]$	$(0.1, 1]$	$(1, 10]$	> 10	$[0, 0.01]$	$(0.01, 0.1]$	$(0.1, 1]$	> 1
FPSS	1922	993	46	4	2	710	120	6	0
OE	1474	1223	257	11	2	625	172	39	0
HL	1384	1324	253	4	2	614	185	37	0
HF	1393	1311	254	6	3	609	187	40	0

The series-parallel algorithm `SPIR-sp`, as a base case for `SPIR`, can essentially be used as a lower bound for the running time of the latter, when applying it on the SPR-subtrees induced by only S- and P-nodes. This resembles the situation that `SPIR` could decide the R-nodes and all their embedding combinations (i.e., the reason for the exponential running time) in no time. Since obtaining these bounds is already not competitive enough, we refrained from a full implementation of the even more demanding enumeration procedures within `SPIR`.

3.2 Evaluation

In the course of our investigation, we will observe the following central findings throughout all benchmark instances, summarized here:

F1 `FPSS` gives the best solution time over virtually all scenarios. When extraction of the embedding is required, `HL` dominates `OE`; both dominate `HF`.
F2 `FPSS` (and to some extent `OE`) are rather independent of the number of embeddings and on whether we consider a yes- or a no-instance.
F3 Generally, all SAT approaches are preferable over their competitors.
F4 `SPIR` (and `SPIR-sp`) and the ILP variants are not competitive.
F5 `EE` and `BB` work well when the number of embeddings is small. For trivial instances, and for large instances with few embeddings, `EE` and `BB` can dominate the SAT approaches.
F6 `EE` is usually far weaker for no-instances than for yes-instances (as all embeddings have to be checked), while the effective pruning of `BB` allows it to typically solve the former faster than the latter.

Instances from Literature. There are mainly three instance sets that have been used in the context of upward planarity:

The *Rome* graphs [14] are (originally undirected) graphs with up to 100 vertices. They can be directed canonically to obtain DAGs. The *North* DAGs [4] were originally collected by AT&T and Stephen North. After filtering for bimodal planar graphs, 2967 and 836 remain, respectively. As some of our algorithms are restricted to biconnected graphs, we also consider sets *Rome2*, *North2* that are generated from the above by planar biconnectivity augmentation, obtaining (after filtering for bimodal planar graphs) 2671 and 780 instances, respectively. Furthermore, we consider the 61 *BDD* instances [2], described above in the context of `BB`'s implementation.

We observe in Table 1 that all SAT formulations solve the *North* instances very fast: each is solved within 1 second, most of them (≈80%) in less than 0.01

Table 2. Number of solved instances (yes- and no-instances) within given time frames in seconds (*Rome2*, *North2*, and *BDD*). In brackets, are the number of no-instances.

	Rome2			*North2*				*BDD*			
time (s)	$[0, 0.1]$	$(0.1, 1]$	> 1	$[0, 0.1]$	$(0.1, 1]$	$(1, 10]$	> 10	[0.0.1]	$(0.1, 1]$	$(1, 10]$	> 10
FPSS	2474 (26)	195 (6)	2 (2)	711 (33)	68 (7)	1 (0)	0 (0)	27 (3)	23 (1)	11 (2)	1 (1)
OE	1743 (12)	915 (17)	13 (5)	600 (24)	149 (13)	31 (3)	0 (0)	24 (3)	25 (1)	12 (2)	1 (1)
HL	1844 (26)	820 (6)	7 (2)	610 (33)	143 (7)	27 (0)	0 (0)	24 (3)	26 (1)	11 (2)	1 (1)
HF	1743 (11)	923 (19)	14 (5)	605 (28)	143 (9)	32 (3)	0 (0)	21 (3)	27 (1)	13 (2)	1 (1)
EE	2665 (29)	5 (4)	1 (1)	753 (27)	5 (1)	6 (4)	16 (8)	62 (7)	0 (0)	0 (0)	0 (0)
BB	2635 (22)	22 (9)	4 (3)	667 (14)	30 (11)	13 (4)	70 (11)	58 (4)	4 (3)	0 (0)	0 (0)

seconds. Generally, FPSS is the fastest SAT, solving 99% of *Rome* in under 0.1 seconds, whereas OE achieves a ratio of 90% (→**F1**).

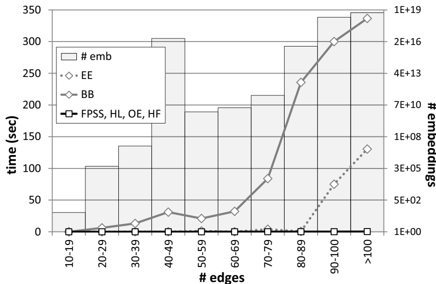


Fig. 1. *North2*; avg. runtime vs. # of edges. The different SATs are visually indistinguishable.

On the biconnected benchmark sets, we can compare the SAT approaches to the other implementations (Table 2): For *Rome2* and *BDD*, both EE and BB seem faster than any SAT formulation. This is mainly due to the triviality of many instances and the necessary overhead of SAT formulations and solvers. Already when considering ≤ 1 second computation time, all approaches solved nearly the same number of instances. We observe that instances of both these sets have only very few embeddings (*Rome2*: max. 36,864, avg. 372), *BDD*: max. 11,528, avg. 512) (→**F5**). As an interesting side note, the lone instance in the last *BDD* column requires roughly 200 seconds for all SAT approaches; it is solved trivially by BB and EE as it has only two planar bimodal embeddings to check. The *North2* instances, however, have many embeddings (avg: 10^{17}), cf. Fig. 1. There, all SAT formulations dominate EE and BB for the non-trivial instances (→**F1**,**F5**).

Constructed Instances. The above instances are very simple, small, and are solved too quickly to deduce general findings. Therefore, we consider a set *Rand* of generated biconnected instances with $n = 50, 100, 150, 200$ nodes and density $|E|/|V| = 1.2, 1.4, \ldots, 2.4$. As suggested in [2], we start with a triangle graph and iteratively perform random edge-subdivisions and face-splits (adding an edge within a face). Now, we orient this embedded graph to be upward planar, and invert $i = 1\%, 2\%, 3\%, 4\%$ of the edges (retaining the DAG property). We generate 1120 instances, 10 per possible parameter setting, which have up to 2.5×10^{15} embeddings (3.7×10^{12} on average). See Fig. 2 for a detailed graphical analysis. We use a timeout of 600 seconds, using this value for unsolved instances when averaging. Overall (Fig. 2, top-left), we can observe FPSS $<$ HL $<$ OE $<$ HF $<$ BB $<$ EE for the running times, FPSS being the clear winner (→**F1**,**F3**,**F5**,**F6**). The

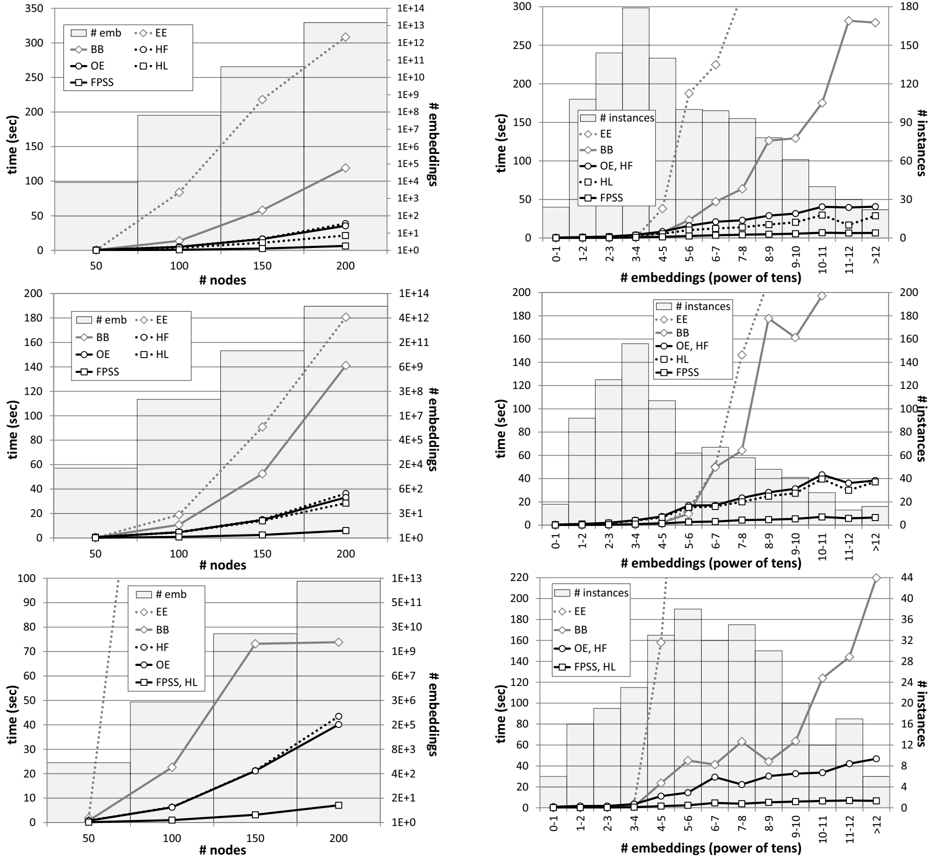
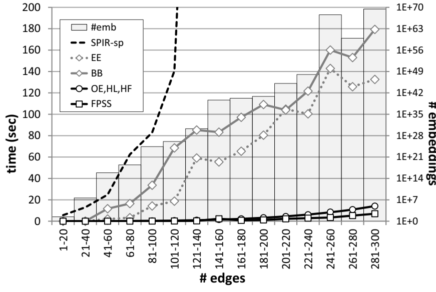


Fig. 2. *Rand* instances; avg. runtime vs. # of nodes (left) or embeddings (right). The latter is given as powers of 10, i.e., $> 10^{12}$ embeddings are considered. We group SAT formulations if they are visually indistinguishable. The first row considers all instances; the second and third row considers only the yes- and no-instances, respectively.

SATs, in particular `FPSS`, seem nearly oblivious to the number of embeddings in the graph (Fig. 2, top-right, $\to$**F2**). It is instructive to consider the yes- and no-instances independently: We see that `EE` performs somewhat reasonable for yes-instances, but fails for no-instances (where it has to check all bimodal embeddings). In contrast to this, `BB` becomes even faster for the latter instances, due to its efficient pruning of large classes of "hopeless" subembeddings ($\to$**F6**). The SAT approaches behave very similar for both kinds of instances ($\to$**F2**).

Now, we consider biconnected series-parallel graphs *SP* to evaluate `SPIR-sp`. We generated a test set of 4500 instances (10 instances per parameter setting) with $m = 20, 40, \ldots, 300$ edges. They are constructed bottom up with probability $p = 0.1, 0.3, \ldots, 0.9$ to perform a serialization instead of a parallelization. We embed the graph, choose an upward planar orientation for the edges, and invert



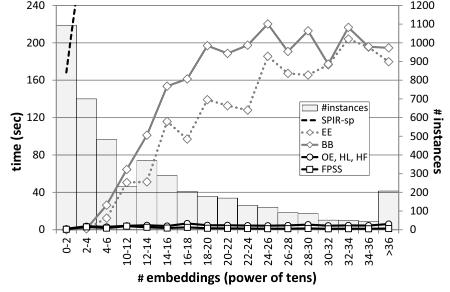
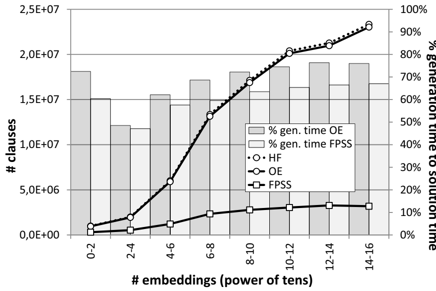


Fig. 3. *SP* instances; avg. runtime vs. # of edges (left) and embeddings (right)



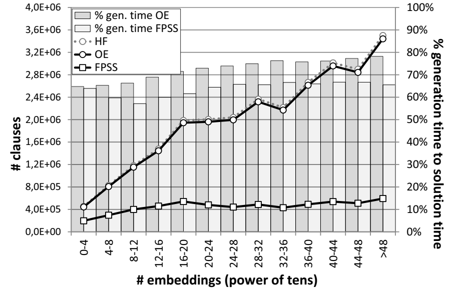


Fig. 4. *Rand* (left) and *SP* (right); the line plots show the number of generated clauses, relative to the number of embeddings (as a power of 10). The bars show the percentage of the running time spent to generate the formula (in contrast to solving it).

$i = 0, 10, 20, 30, 40, 50\%$ of the edges (retaining the DAG property). Again, we use a timeout of 600 seconds, using this value for unsolved instances when averaging. `SPIR-sp`—although formally the only polynomial time algorithm in this comparison—offers the clearly weakest performance, solving no instance with over 100 vertices in under 5 minutes and running into the time limit for all graphs with more than 120 edges (Fig. 3, left). The picture is analogous (Fig. 3, right) when looking at the runtime depending on the number of embeddings ($\rightarrow$**F4**). On *SP*, `EE` performs better than `BB`, but this is since nearly 90% of the instances happen to be upward planar. Considering yes- and no-instances independently, we can observe the same pattern as for *Rand* ($\rightarrow$**F6**).

Details on SAT. Although `FPSS` and `OE` have the same number of clauses in $\mathcal{O}$-notation—dominated by $(\mathrm{R}^{\mathrm{t}}_{\tau})$ over the common τ variables—the former has considerably fewer additional clauses. In fact, this seems to be one of the main reasons of `FPSS`'s superior performance. To back-up this assumption, consider the line diagrams in Fig. 4. They show that `FPSS` is rather independent of the number of embeddings ($\rightarrow$**F2**). Impressively, the (minor) difference between `OE` and `HF` (="`OE` $\cup$`FPSS`") shows that the number of clauses `FPSS` has to consider additionally to $(\mathrm{R}^{\mathrm{t}}_{\tau})$ is negligible.

The bar diagrams in Fig. 4 show that the SATs spend a large portion of their time (≈70%) with (trivially) *constructing* the formula. This explains the overhead for trivial instances where EE and BB can be faster than SAT (→**F5**).

Acknowledgements. We thank Marcus Schaefer for pointing us to [8] and its potential applicability within our SAT approach, Walter Didimo for helpful discussions on BB, and Fabrice Stellmacher and Kerstin Gössner for implementation support with BB and SPIR, respectively.

References

1. Bertolazzi, P., Di Battista, G., Liotta, G., Mannino, C.: Upward drawings of triconnected digraphs. Algorithmica 12(6), 476–497 (1994)
2. Bertolazzi, P., Di Battista, G., Didimo, W.: Quasi-upward planarity. Algorithmica 32(3), 474–506 (2002)
3. Chimani, M., Zeranski, R.: Upward planarity testing via SAT. In: Didimo, W., Patrignani, M. (eds.) GD 2012. LNCS, vol. 7704, pp. 248–259. Springer, Heidelberg (2013)
4. Di Battista, G., Garg, A., Liotta, G., Parise, A., Tamassia, R., Tassinari, E., Vargiu, F., Vismara, L.: Drawing directed acyclic graphs: An experimental study. Int. J. of Computational Geometry and Appl. 10(6), 623–648 (2000)
5. Di Battista, G., Liu, W.P., Rival, I.: Bipartite graphs, upward drawings, and planarity. Information Processing Letters 36(6), 317–322 (1990)
6. Di Battista, G., Tamassia, R.: On-line planarity testing. SIAM J. on Computing 25, 956–997 (1996)
7. Didimo, W., Giordano, F., Liotta, G.: Upward spirality and upward planarity testing. SIAM J. on Discrete Mathematics 23(4), 1842–1899 (2009)
8. Fulek, R., Pelsmajer, M.J., Schaefer, M., Štefankovič, D.: Hanani–Tutte, monotone drawings, and level-planarity. Thirty Essays on Geom. Graph Th., 263–287 (2013)
9. Garg, A., Tamassia, R.: On the computational complexity of upward and rectilinear planarity testing. SIAM J. on Computing 31(2), 601–625 (2002)
10. Healy, P., Lynch, K.: Fixed-parameter tractable algorithms for testing upward planarity. In: Vojtáš, P., Bieliková, M., Charron-Bost, B., Sýkora, O. (eds.) SOFSEM 2005. LNCS, vol. 3381, pp. 199–208. Springer, Heidelberg (2005)
11. Hutton, M.D., Lubiw, A.: Upward planar drawing of single source acyclic digraphs. In: Proc. SODA 1991, pp. 203–211 (1991)
12. Pach, J., Toth, G.: Monotone drawings of planar graphs. J. of Graph Theory 46(1), 39–47 (2004)
13. Papakostas, A.: Upward planarity testing of outerplanar dags. In: Tamassia, R., Tollis, I.G. (eds.) GD 1994. LNCS, vol. 894, pp. 298–306. Springer, Heidelberg (1995)
14. Welzl, E., Di Battista, G., Garg, A., Liotta, G., Tamassia, R., Tassinari, E., Vargiu, F.: An experimental comparison of four graph drawing algorithms. Computational Geometry 7, 303–325 (1997)