

An ILP-based Proof System for the Crossing Number Problem*

Markus Chimani¹ and Tilo Wiedera²

- 1 Theoretical Computer Science, Osnabrück University, Germany
markus.chimani@uni-osnabrueck.de
- 2 Theoretical Computer Science, Osnabrück University, Germany
tilo.wiedera@uni-osnabrueck.de

Abstract

Formally, approaches based on mathematical programming are able to find provably optimal solutions. However, the demands on a verifiable formal proof are typically much higher than the guarantees we can sensibly attribute to implementations of mathematical programs. We consider this in the context of the *crossing number problem*, one of the most prominent problems in topological graph theory. The problem asks for the minimum number of edge crossings in any drawing of a given graph. Graph-theoretic proofs for this problem are known to be notoriously hard to obtain. At the same time, proofs even for very specific graphs are often of interest in crossing number research, as they can, e.g., form the basis for inductive proofs.

We propose a system to automatically generate a formal proof based on an ILP computation. Such a proof is (relatively) easily verifiable, and does not require the understanding of any complex ILP codes. As such, we hope our proof system may serve as a showcase for the necessary steps and central design goals of how to establish formal proof systems based on mathematical programming formulations.

1998 ACM Subject Classification G.1.6 [Optimization] Integer Programming, G.2.2 Graph Theory, G.4 [Mathematical Software] Verification

Keywords and phrases automatic formal proof, crossing number, integer linear programming

Digital Object Identifier 10.4230/LIPIcs.ESA.2016.29

1 Introduction

A typical corner stone of (integral) mathematical programming formulations for combinatorial optimization problems is that solving the formulation constitutes a *proof* that the obtained solution is in fact optimal. While this is true in theory, it is not clear, per se, that this actually transfers into practice, as many factors may influence or invalidate the program's outcome: the probably most prominent ones are hidden bugs in the software or numerical instabilities. E.g. [1] discusses the problems and challenges of proving the correct computation of an optimal TSP tour for *one* specific instance; we are interested in a system to deduce proofs automatically, without any human interaction.

Hence, while there exist many successful formulations, e.g. as ILPs, to many important graph-theoretic (optimization) problems, successful computations are generally not considered to be proofs accepted by the graph theory community.

We aim at bridging this gap for the well-known *crossing number problem*, to be defined below, which is arguably one of the most prominent and notorious problems in topological

* This research has been supported by the German Research Foundation (DFG) project CH 897/2-1.



graph theory. We propose a system to extract a *simply verifiable* proof from a successful ILP computation, which can be accepted by graph theorists: it is shielded against ill-effects based on the software realization of the mathematical model. To understand the proof, the graph theorist does *not* have to have a deeper understanding of mathematical programming nor the required implementations; she only has to understand the mathematical model (and possibly a simple proof verification program, designed to be readable and checkable by non-experts).

When describing the proof system, we will also pinpoint the generally necessary differences in the formulation-, algorithm-, and software-design between the typical goal of obtaining a strong and fast solver and the goal of obtaining a system for easily understandable proofs. Crossing number formulations are in particular interesting in that respect, as their implementations need to combine a diverse set of different tools (branch-and-cut-and-prize with exact and heuristic constraint separation, column generation with non-standard bounding schemes, intricate heuristics for primal bounds, etc.) to obtain a system that can solve realistically-sized instances. This inevitably leads to a software too complex to directly check against all possible bugs. As such, even though we focus on the crossing number problem, our system may serve as a showcase of how to obtain trustworthy mathematical programming based proof systems for graph-theoretic problems whose formulations do not allow easily checkable implementations.

We will start with describing the crossing number problem, the reason why we are interested in formal proofs for the crossing number of specific graphs, and the currently available methods to obtain solutions. In Section 3 we will discuss the central design goals of our proof system, as well as their realization. This includes a new column generation scheme, balancing simplicity and effectiveness. A brief experimental study in Section 4 shows the applicability of our approach.

2 Crossing Number Problem

The crossing number $cr(G)$ of a graph G is the minimum number of pairwise edge crossings in any drawing of G in the plane. The problem garnered a lot of contributions since its first mention over 70 years ago; see [30, 29] for an extensive bibliography and survey. Beside its own inherent appeal, crossing numbers also occur, e.g., in conjectures relating $cr(G)$ to graph colorings and knot theory.

Nonetheless, some of the most natural questions are still open, most importantly the crossing number of nearly all classical graph classes like complete graphs (known only for K_n with $n \leq 12$ [28]), complete bipartite graphs, etc. The quest for crossing number proofs of particular families is a lively research field in graph theory, see, e.g., [3, 16, 21, 22, 26], and even partial results (like proving that K_{13} cannot have crossing number 217 [25]) are publishable in renowned journals. Several such proofs start out with a set of base cases for which the crossing number has to be proven in excruciating detail by hand, before employing an inductive proof to consider a full graph class. Obtaining sound, automatic, and standardized proofs for such base cases is one of the goals of our proof system.

Deciding the crossing number of an arbitrary graph G is known to be NP-complete [17]; this holds even for restricted graph classes like cubic graphs [18] or graphs that become planar when removing a single edge [6]. While there are several known practically strong heuristics and (partial) approximation results, we do, e.g., not even know if the problem allows a constant approximation ratio; we only know that the problem does not allow a PTAS [5]. The problem is known to be fixed-parameter tractable with parameter $cr(G)$ [20]. However, the algorithm's runtime is doubly-exponentially dependent on the parameter and it is, as already mentioned in [20], not feasible in practice.

The only known practical method to obtain the crossing number of a given graph is based in integer linear programs [4, 11, 12], based on two different modeling ideas to be described below. However, the models contain both too many variables and too many constraints to be solved directly via off-the-shelf techniques, and require a lot of (bug-prone) implementation effort. Even if implemented correctly, solving such ILPs can be error-prone due to numerical instabilities. This constitutes a problem for graph theorists, interested in utilizing the computed crossing number in a proof.

Basics

Kuratowski's famous theorem [24] states that a graph G is planar if and only if it does not contain a subdivision of a complete graph on five vertices (K_5) or a complete bipartite graph on three vertices per partition set ($K_{3,3}$) as a subgraph. We will call these subgraphs *Kuratowski subdivisions* of G . The paths in the subdivision that resemble a single edge of the K_5 or $K_{3,3}$ are referred to as *Kuratowski paths* in the following. We use this theorem to argue that every such subdivision in given G is required to be drawn with at least one crossing.

When considering the crossing number of a graph, it is well-known that it suffices to consider *good* drawings: no edge crosses itself; adjacent edges do not cross; each pair of edges crosses at most once; and no three edges cross in a common point. Considering such an optimal drawing of G , we can obtain a *planarization* of G , which is the graph arising from G when replacing each crossing with a new *dummy* vertex of degree 4. We may speak of a *partial planarization* if we substitute only certain crossings via new vertices such that the obtained graph possibly remains non-planar.

2.1 Known ILP Models

All known ILP models have a common core idea; they differ in how to handle the arising *realizability problem*, described below. We will only describe the formulation on a level necessary to comprehend the proof system (and the design decisions that lead to it). For a more detailed and formal description see the individual publications [4, 11, 12] or the full compilation in [7].

Let $G = (V, E)$ be the given simple and undirected graph for which to compute $cr(G)$, and let $\mathcal{CP} := \{\{e, f\} \subseteq E : e \cap f = \emptyset\}$ be the set of edge pairs that potentially cross in a good drawing of G . Consider a binary variable x_c for each $c \in \mathcal{CP}$ that is 1 if and only if the edge pair crosses. This gives the objective function

$$\min \sum_{\{e, f\} \in \mathcal{CP}} w(e) \cdot w(f) \cdot x_{\{e, f\}} \quad (1)$$

where $w(e)$ denotes the (integral) weight of edge e . For usual (i.e., unweighted) graphs we have $w(e) = 1$ for all $e \in E$. In our implementation we first preprocess G to obtain a smaller but integrally-weighted graph with the same crossing number [9].

To guarantee feasible solutions we may be tempted to introduce the following *Kuratowski constraints*. Let $K \subseteq E$ be an edge subset forming a Kuratowski subdivision; we want to ensure that each such K gives rise to at least one crossing. We say that a crossing $c \in \binom{K}{2} \cap \mathcal{CP}$ is *planarizing* if the graph obtained from K by realizing c via a dummy vertex is planar. Clearly, a crossing is planarizing if and only if the crossing edges do *not* belong to adjacent (or identical) Kuratowski paths. Let $\mathcal{CP}(K)$ be the set of planarizing crossings for

$K \subseteq E$.

$$\sum_{c \in \mathcal{CP}(K)} x_c \geq 1 \quad \forall \text{ Kuratowski subdivisions } K \text{ in } G.$$

While these constraints form facets of the crossing number polytope [8], they do not suffice to guarantee feasibility: On the one hand, we also have to consider Kuratowski subdivisions that only appear in partial planarizations due to dummy vertices. On the other hand, even those do not suffice: Let $R \subseteq \mathcal{CP}$ be edge pairs that are supposed to cross. The *realizability problem* is to decide whether there exists a drawing of G such that only the edge pairs R cross. Interestingly, even this seemingly simpler problem is still NP-hard for general graphs [23]. Hence, our simple set of x -variables cannot suffice to describe the crossing number polytope. The key problem is that when two edges f, g both cross an edge e , the *order* of these two crossings along e is of central importance and cannot be deduced in polynomial time (unless $P = NP$).

There are two approaches to tackle this problem. Both lead to a variable increase that, although polynomial, makes the models intractable in practice unless a dynamic column generation scheme is used. Assume in the following that we assign an arbitrary but fixed direction to each edge.

Subdivision-based exact crossing minimization (SECM)

Let $G^{[\ell]}$ be the graph obtained from G by splitting each edge into a chain of $\ell \in \mathbb{N}^+$ edges (henceforth called segments). Instead of directly using the above model on G , we consider $G^{[\ell]}$ instead. We observe that the corresponding set $\mathcal{CP}^{[\ell]}$ will not need to contain edge pairs (segment pairs, in fact) where both segments belong to the same original edge in G (the underlying G does not require self-crossings).

On $G^{[\ell]}$, we search for the smallest number of crossings under the restriction that each segment is involved in at most one crossing. This restriction is trivial to ensure via linear constraints (see later for details). The so-restricted crossing number is often called the *simple crossing number*, even though it is still NP-complete to decide. There can be at most $\chi := \min\{cr(G), |E| - 1\}$ crossings on an edge in the optimal drawing of G . Hence, we may use any upper bound on χ as ℓ to ensure that the optimal solution to the restricted crossing number problem on $G^{[\ell]}$ induces an optimal solution for the usual crossing number on G . Since there are instances where an edge e needs to be crossed by $\Omega(|E|)$ many other edges in the optimal solution, our transformation may increase the number of variables $\Theta(|E|^2)$ -fold.

The benefit of considering the simple crossing number is that the realizability problem becomes linear time solvable. We say a subset $R \subseteq \mathcal{CP}^{[\ell]}$ is *simple* if each segment occurs at most once over all segment pairs in R (i.e., it is a potential solution to the simple crossing number). For such an R , its corresponding (partial) planarization $P(R)$ – obtained by substituting the crossings R in G with dummy vertices – is hence unique. We have R realizable iff $P(R)$ is planar.

Finally, we can ensure feasible solutions using more general Kuratowski-constraints. Let $\mathcal{K}(R)$ be the set of all Kuratowski subdivisions in $P(R)$. Each subdivision is specified by its edge set. Clearly, if the crossings R are part of the solution, each Kuratowski subdivision in $\mathcal{K}(R)$ will require at least one crossing. We have:

$$\sum_{c \in \mathcal{CP}(K)} x_c \geq 1 - \sum_{c \in R} (1 - x_c) \quad \forall \text{ simple } R \subseteq \mathcal{CP}, K \in \mathcal{K}(R) \quad (2)$$

In order to prove a lower bound of the crossing number, it suffices to understand that the constraints in the above model need to hold for any feasible solution. We do not need to argue about sufficiency (see also property ⟨2⟩ below).

Ordering-based exact crossing minimization (OECM)

The alternative formulation [12] introduces *linear ordering variables* to resolve the order of crossings along each edge without subdividing the input graph. This has some advantages regarding performance, e.g., since every Kuratowski constraint in this model can cover more than just one specific partial planarization. Technically, these linear orderings are modeled using $\Theta(|E|^3)$ additional variables that are linked to the crossing variables using several constraint classes. Overall, it has to be observed that the OECM model, while offering superior performance, is much harder to understand and requires even more technically intricate column generation schemes, book-keeping, and subalgorithms, compared to SECM.

3 Proof System

Without a proof system, one would need to check the ILP algorithms for correctness. All SECM and OECM implementations known to the authors are intertwined with the Open Graph Drawing Framework (OGDF, www.ogdf.net, [10]) and heavily utilize the ABACUS framework (<http://www.informatik.uni-koeln.de/abacus>, [19]); they are all written in C++. The core of both algorithms roughly spans across 8,000 Lines of Code (LOC) while the OGDF amounts to a total of about 170,000 LOC. Already the main code paths of the research code are hard to comprehend without intricate knowledge of the algorithms. Furthermore, the programs use sophisticated column generation routines, requiring complex book-keeping and special constraint liftings to prevent a decrease of the lower bound when adding variables. Tracking variables and constraints over an entire algorithm is disproportionately harder than simply verifying each branch-and-bound (B&B) leaf. Furthermore, there are several possibilities for hidden bugs due to numerical instabilities that may arise without any means of detection, or hidden buffer overruns when generating atypically many variables or constraints in one pass.

All these facts make a formal verification of the main algorithms intractable in practice. For comparison, our proof system proposes a verification procedure (written in Java) of less than 1,000 LOC (including rich documentation and comments), with a virtually complete test coverage.

In our context, a *proof* consists of three parts:

- a mathematical model (in our case the ILP formulation),
- a *witness* of the dual bound, and
- a primal *solution*, matching the above dual bound.

The first is independent of the specific instance but needs to be understood only once for a specific problem domain (crossing number, in our case). The latter two are instance-dependent. We want to make the proof as easily digestible by pure graph theorists as possible. To understand the proof it must be sufficient to understand the following:

- ⟨1⟩ *Solution*. One needs to be able to check the feasibility of a primal solution and evaluate its objective value. In our case, one needs to be able to recognize a feasible planarization of G , and to count the number of dummy vertices.
- ⟨2⟩ *Feasibility of the mathematical model*. It is only necessary to understand that all described constraints of the formulation are feasible; one need not concern oneself with

understanding why the model is sufficient. Generally, it should be understood that any optimal fractional solution w.r.t. a subset of the constraints gives a feasible dual (in our case, lower) bound.

- ⟨3⟩ *Witness format.* The information contained in the witness that is required to verify the dual bound.
- ⟨4⟩ *Verification procedure.* The steps required to verify the dual bound claimed by the witness.

An ILP-based *proof system* should consist of two major components: the proof generation and the verification procedure. The arbitrarily complex proof generation produces a witness for the optimality of the primal solution. This witness contains all information required to create relaxed linear programs for several *subcases* (the leaves in the B&B tree), all of which yield the dual bound. B&B leaves naturally resemble an easily verifiable case distinction, as used ubiquitously in graph-theoretic proofs.

The verification of the witness could *theoretically* be done by hand. It follows from the nature of NP-complete problems that (unless $P = NP$) there *cannot* be a really “simple” proof for the dual bound in general. If we want a simple-to-check witness for the dual bound (which is, most importantly, checkable in polynomial time w.r.t. to its size), we *have* to live with the fact that the witness’ size can grow exponentially with the size of G . In most cases, this sheer size will require us to introduce an – algorithmically very simple – computer-based verification procedure. Most importantly, the verification procedure only needs to check that the subcases described in the witness form linear programs that are subsets of the underlying mathematical model. It requires no knowledge about the generation of constraints, variables, or branches.

We can summarize the general design goals for an ILP-based proof system:

- G1.** *Simplicity of model.* There should be few classes of constraints and variables. Comprehensibility outweighs performance as long as the proof procedure is still “fast enough”.
- G2.** *Column generation.* Column generation should only be used if ultimately required. The variable subsets need to be as simple as possible.
- G3.** *LP-solver flexibility/provability.* The LP-solver used during verification should be easily interchangeable or self-proving.
- G4.** *Few Branches.* Superfluous branching decisions should be eliminated from the witness to keep it small. This can, e.g., be achieved by starting the extraction with a supposedly optimal primal bound, see below.
- G5.** *Human-readability.* One should be able to investigate certain aspects of the proof by hand. To achieve this, we may allow redundancy as long as conflicts are detected easily by the verifier.
- G6.** *Standalone verification.* The verifier must *not* share any resources (most importantly code fragments) with the extraction procedure. The witness and the primal solution constitute the sole interchange of information between generation and verification.
- G7.** *Coding standards.* Adhering to well-established coding standards when implementing the verifier increases readability. Likewise, an established programming language should be used. The verification procedure should be described in detail such that one might re-implement it easily.

Although the OEMC formulation offers better performance than SECM in practice, goal G1 lets us favor the comparably much easier to understand SECM formulation. It also allows us to sacrifice more constraints classes to adhere to G1. Concerning G2, both formulations require complex column generation schemes to be feasible in practice. However, as we will describe below, SECM allows us to propose a new column generation scheme

Algorithm 1: Proof generation	
Require: graph G	// we are interested in $cr(G)$
1: $P, UB \leftarrow \text{OECM}(G)$	// find presumably optimal planarization P // (with objective value UB)
2: $W \leftarrow \text{Modified-SECM}(G, UB)$	// generate a witness W for the lower bound, // i.e., $UB - 1$ is infeasible to achieve
3: print P and W	// output the proof

that is considerably simpler than any of the previously published ones for either of the two formulations, while increasing the runtime and number of variables only mildly.

Alg. 1 gives an outline of the proof generation. In order to obtain a small proof (G4), we solve the crossing number problem *twice*: First, we use the fastest OECM variant together with strong upper bound heuristics to obtain the presumably optimal solution. This procedure can be seen as a black box, as we are only interested in the fact that the solution gives an upper bound – we can check the feasibility of the primal solution straight-forwardly. If our proof generation succeeds, this is the solution used as part of the overall proof. Now having this primal bound, we can start our modified SECM formulation (see below for details) without any primal heuristics and ask for a solution strictly better (at least one crossing less) than the obtained upper bound. From this second ILP run, we can extract all required information for each B&B leaf, to reconstruct each linear program that yields a lower bound on the number of crossings restricted to the solution space spanned by the branch. In general, the set of variables and constraints differ for any two leaves.

We observe that if OECM did *not* find an optimum solution (e.g., due to a hidden bug), we may already detect this now as SECM’s dual bound does not match our upper bound.

3.1 Modified-SECM

There are two known column generation schemes for SECM [11]. The *algebraic pricing*, based on the standard Dantzig-Wolfe decomposition theory, performs relatively weak and uses a quite unstructured variable subset. The more efficient *combinatorial* column generation scheme (denoted as *sparse* column generation in the following) can decide upon the addition of variables in a purely combinatorial fashion, and requires the fewest active variables in general. However, the required variable subset structure (and hence the reasoning for its sufficiency) is too complicated to easily describe and comprehend for the purpose of a graph-theoretic proof. We hence propose a new column generation scheme – called *homogeneous* in the following – by means of describing an SECM variant that is slightly modified compared to the model described above.

Instead of a simple number, let $\ell: E \rightarrow \mathbb{N}$ be a mapping describing the *expansion status* of G , i.e., we consider each edge $e \in E$ of G to be subdivided into $\ell(e)$ segments. We define our ILP using the resulting graph G^ℓ . For notational simplicity, let $e_1, e_2, \dots, e_{\ell(e)}$ denote the segments of an original edge $e \in E$. As before, the new graph induces a set of segment pairs \mathcal{CP}^ℓ that may cross in an optimal solution. We explicitly allow (and expect) values $\ell(e)$ to be smaller than the upper bound of crossings over e . To this end, we allow at most one crossing over each segment *except for the first segment of each edge*: it may be crossed an arbitrary number of times. In general, this could lead to problems with testing realizability. However, this is of no concern to us, as we only require that our model is *feasible*, i.e., it allows to describe an optimal solution (see (2)). This is trivially the case in this setting (already when $\ell(e) = 1$ for all $e \in E$).

Algorithm 2: Proof verification

Require: graph G , subcases \mathcal{L} (=B&B leaves), claimed lower bound $b \in \mathbb{N}^+$

- 1: **assert** branchCoverage(\mathcal{L})
- 2: **for all** $\nu \in \mathcal{L}$ **do**
- 3: $\ell \leftarrow$ expansion status at ν
- 4: $\mathcal{K} \leftarrow$ set of Kuratowski subdivisions observed at ν
- 5: **for all** $C \in \mathcal{K}$ **do**
- 6: **assert** isKuratowski(G^ℓ, C)
- 7: $P \leftarrow$ generateLinearProgram(G, \mathcal{K})
- 8: **assert** lpsolve(P) $> b - 1$

Symmetric solutions increase our set of subcases in the proof, often drastically. We can require w.l.o.g. that the crossings over an original edge may be *aligned* in such a way that there is only a crossing on segment $i > 2$ if there also is a crossing on segment $i - 1$. Segment 1 is typically not part of this alignment scheme, as it allows multiple crossings. However, observe that any edge $e = \{u, v\} \in E$ may be crossed at most $u_e := \min\{UB, |E| + 1 - \deg(u) - \deg(v)\}$ times in the optimum solution, where UB is an upper bound on $cr(G)$. If an edge is fully expanded, i.e., $\ell(e) = u_e$, we do allow at most one crossing over segment 1; to avoid symmetries we can assume to have less crossings on segment 1 than on segment $\ell(e)$. The following constraints establish these segment properties. They, together with the objective function (1) and the Kuratowski constraints (2) (both applied to the set \mathcal{CP}^ℓ), form our full mathematical model.

$$\sum_{\{e_i, f\} \in \mathcal{CP}^\ell} x_{\{e_i, f\}} \leq \begin{cases} 1 & \text{if } i = 2 \\ \sum_{\{e_{i-1}, f\} \in \mathcal{CP}^\ell} x_{\{e_{i-1}, f\}} & \text{else} \end{cases} \quad \forall e \in E, 2 \leq i \leq \ell(e) \quad (3)$$

$$\sum_{\{e_1, f\} \in \mathcal{CP}^\ell} x_{\{e_1, f\}} \leq \sum_{\{e_{\ell(e)}, f\} \in \mathcal{CP}^\ell} x_{\{e_{\ell(e)}, f\}} \quad \forall e \in E : \ell(e) = u_e \quad (4)$$

Remark (Irrelevant to understanding the proof)

As in SECM, Kuratowski constraints are separated via a planarity-test based procedure on a rounded solution S . An effective column generation similar to [4] is achieved by starting with a unit vector ℓ and incrementing $\ell(e)$ whenever there are at least 2 crossings on e_1 in S (i.e., the realization problem cannot be solved uniquely).

3.2 Verification Procedure

Finally, we can focus on the actual verification steps necessary to prove the lower bound obtained by the above Modified-SECM model. Alg. 2 gives an overview.

Branch Coverage

We need to make sure that the entire solution space is covered. Therefore, we consider the variable fixings in all subcases. Since we only branch on single variables, we iteratively merge two subcases that differ by the assignment of a single variable, giving a more general subcase without this variable being fixed at all. In a valid proof, this procedure must end with a single subcase, which does not have any variable fixings at all.

Algorithm 3: Coverage verification

<p>Require: subcases \mathcal{L} (see text)</p> <p>1: while $\exists \mu, \nu \in \mathcal{L}$ with $\exists c \in \mathcal{CP}^\ell : \mu \Delta \nu = \{(c, 0), (c, 1)\}$ do</p> <p>2: $L \leftarrow \{\mu \cap \nu\} \cup L \setminus \{\mu, \nu\}$</p> <p>3: assert $\mathcal{L} = \{\emptyset\}$</p>
--

The following pseudo-code shows how to algorithmically verify that the subcases span the whole solution space. Here, we consider each subcase ν to be a set of tuples from $\mathcal{CP}^\ell \times \{0, 1\}$, i.e., a set of segment pairs that are specified to either cross (1) or not (0). Segment pairs not listed in ν are free to do either. Let Δ denote the symmetric difference.

Kuratowski Constraints

For each subcase, we need to check that only feasible constraints are considered. A Kuratowski constraint for a subgraph K that is not a Kuratowski subdivision would be an error. It would enforce a crossing that may not be necessary in the optimal solution. For each subcase, our witness explicitly stores each used Kuratowski subgraph K , together with the required crossings (R) that need to exist for K to arise¹. More specifically, K is stored by means of Kuratowski paths p_1^K, \dots, p_k^K . This storage pattern allows for a simpler verification (see below) than a general Kuratowski verification routine as described in [27].

To verify that K is really a Kuratowski subdivision, we first check whether each p_i^K is in fact a valid path (only exploiting crossings of R , if any). Then we check that all paths are pairwise internally-disjoint (i.e., they are disjoint except for possibly common start/end vertices). Finally, the set of nodes that constitute the start or end of all Kuratowski paths is collected. The size of this set (5, 6) and the number of Kuratowski paths (10, 9) is verified according to the type of the subdivision (K_5 , $K_{3,3}$, respectively). The structural verification of a K_5 subdivision simply checks whether all 5 nodes are directly connected to one another via Kuratowski paths. For a $K_{3,3}$, we perform a two-coloring (interpreting the paths as edges); each of the 6 nodes must be connected to exactly 3 distinct nodes of the opposite color.

Lower Bound

Finally, we need to verify the lower bound for each subcase. We can trivially generate a linear program (no integrality constraints) according to our model. From the expansion status ℓ we can construct \mathcal{CP}^ℓ , the objective function (1) and the segment-oriented constraints (3) and (4). For each (already verified) Kuratowski subdivision considered in the subcase, we generate the corresponding constraint (2).

By writing this LP in a standard file, we can use *any* LP-solver (or multiple, to gain confidence) to verify that the LP's solution value is strictly larger than $UB - 1$. For a more formal proof, we may check the basis of the final tableau/the dual solution to verify the lower bound and/or use self-proving LP solvers [2, 13].

¹ While constraints *could* be stored without explicitly stating R , this would decrease the readability of the proof and the verification procedure, cf. G5.

4 Practice and Experiments

4.1 Web-Service

Already prior to our proof system, we offered a (free) web-service to compute the crossing number of an uploaded graph (see <http://crossings.uos.de>). Over the last years it has been used as a tool by several research groups worldwide, to help validate or falsify crossing number conjectures and ideas. We collected the thereby uploaded instances. However, the web-service would (formally) only give a primal solution, together with the assertion that this *should* be the optimum. Now, we relaunched the web-service to also hand out the formal proof. The user can download the stand-alone Java verification program, check it, and use it to verify her proof independently.

4.2 Experimental Evaluation

To determine the applicability of the proof system, we tested the algorithms on three benchmark sets: the 3110 non-trivial Rome graphs [14], the 1277 North graphs [15], and the 145 non-planar graphs (<http://crossings.uos.de/instances>) collected by our crossing number web-service.

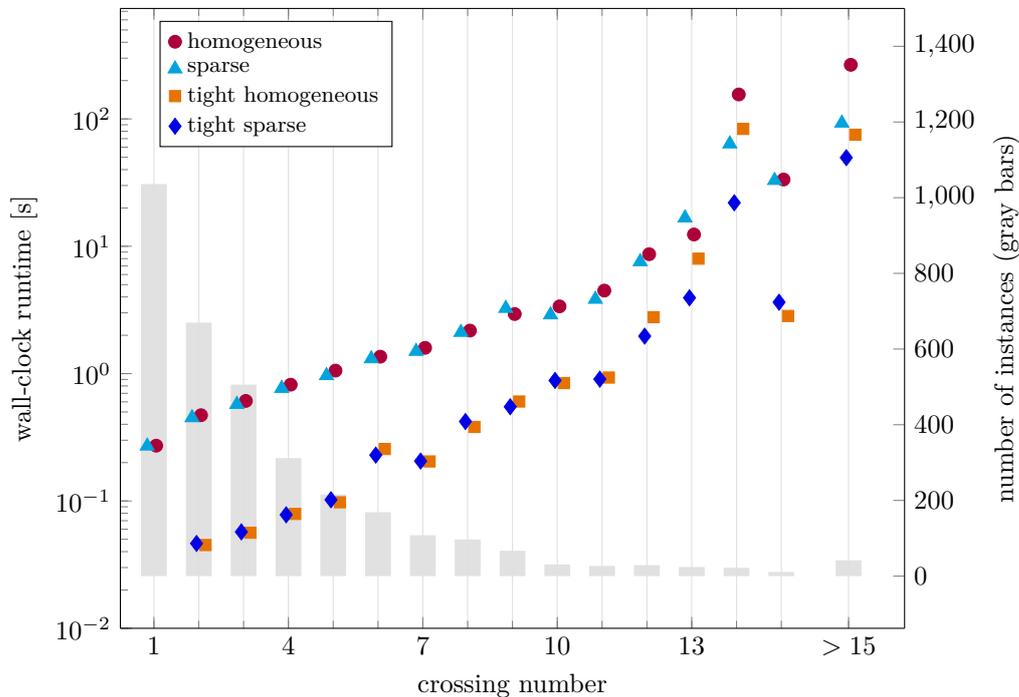
All experiments were conducted using an Intel Xeon E5-2430 v2, 2.50 GHz with 192 GB RAM running on Debian 8. We compiled with g++ 4.9.0 (64bit, -O3), used CPLEX 12.6.0 as the backend LP-solver, and applied a time-limit of 60 minutes for each computation. All algorithms except the verifier are implemented as part of the OGDF (using ABACUS as the ILP-framework). We compare the *sparse* to the newly introduced *homogeneous* column generation scheme, to understand the runtime costs of the simpler but supposedly weaker column generation scheme. For both schemes we consider the cases whether we start with a *tight* upper bound (the optimum) or not; the former is the setting we typically use within our proof system. Fig. 1 summarizes the results. While *tight homogeneous* requires more time than *tight sparse* on larger instances, it is still faster than *sparse* without a tight upper bound. On all instances with crossing number at most 22, *homogeneous* is at most 5 times slower than *sparse* (for both upper bound modes, considering only those instances solved by both schemes). Using the tight upper bound reduces the runtime to about 30% on average for *homogeneous*. Out of all 3393 instances solved by *tight sparse*, only 11 could not also be solved by *tight homogeneous* in time. Thus, we conclude that the increase in running time due to the simpler column generation is reasonable in practice. The runtime of the verification procedure is negligibly small.

5 Conclusion

We considered the problem of bridging the gap between “provably optimal” solutions obtained via mathematical programming and the demands on a verifiable formal proof. To this end, we laid out the general central design goals and steps to turn a mathematical program into a proof system.

We combined this with a showcase of how to automatically obtain a verifiable proof for the graph-theoretic crossing number problem, whose known ILP implementations are far from being formally checkable. To this end, we also introduced a novel column generation scheme for the problem’s model, which, while much simpler, is still very effective in practice. The final proof system is available online for free academic use.

■ **Figure 1** Running time of different SECM variants (see text). The tight variants finished in roughly 10^{-5} seconds for instances with crossing number 1.



Acknowledgements. We thank an anonymous reviewer for her valuable feedback including detailed information on related publications.

References

- 1 David Applegate, Robert E. Bixby, Vasek Chvátal, William J. Cook, Daniel G. Espinoza, Marcos Goycoolea, and Keld Helsgaun. Certification of an optimal TSP tour through 85, 900 cities. *Oper. Res. Lett.*, 37(1):11–15, 2009. doi:10.1016/j.orl.2008.09.006.
- 2 David Applegate, William J. Cook, Sanjeeb Dash, and Daniel G. Espinoza. Exact solutions to linear programming problems. *Oper. Res. Lett.*, 35(6):693–699, 2007. doi:10.1016/j.orl.2006.12.010.
- 3 Drago Bokal. On the crossing numbers of cartesian products with paths. *Journal of Combinatorial Theory, Series B*, 97(3):381–384, 2007. doi:10.1016/j.jctb.2006.06.003.
- 4 Christoph Buchheim, Markus Chimani, Dietmar Ebner, Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Petra Mutzel, and René Weiskircher. A branch-and-cut approach to the crossing number problem. *Discrete Optimization*, 5(2):373–388, 2008. doi:10.1016/j.disopt.2007.05.006.
- 5 Sergio Cabello. Hardness of approximation for crossing number. *Discrete & Computational Geometry*, 49(2):348–358, 2013. doi:10.1007/s00454-012-9440-6.
- 6 Sergio Cabello and Bojan Mohar. Adding one edge to planar graphs makes crossing number and 1-planarity hard. *SIAM Journal on Computing*, 42:1803–1829, 2013.
- 7 Markus Chimani. *Computing Crossing Numbers*. PhD thesis, TU Dortmund, 2008. URL: <http://hdl.handle.net/2003/25955>.

- 8 Markus Chimani. Facets in the crossing number polytope. *SIAM Journal on Discrete Mathematics*, 25(1):95–111, 2011. URL: http://epubs.siam.org/sidma/resource/1/sjdmec/v25/i1/p95_s1, doi:10.1137/09076965X.
- 9 Markus Chimani and Carsten Gutwenger. Non-planar core reduction of graphs. *Discrete Mathematics*, 309(7):1838–1855, 2009. doi:10.1016/j.disc.2007.12.078.
- 10 Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Karsten Klein, and Petra Mutzel. The open graph drawing framework (OGDF). In Roberto Tamassia, editor, *Handbook on Graph Drawing and Visualization.*, pages 543–569. Chapman & Hall/CRC, 2013.
- 11 Markus Chimani, Carsten Gutwenger, and Petra Mutzel. Experiments on exact crossing minimization using column generation. *ACM J. Experim. Alg.*, 14:4:3.4–4:3.18, 2010. doi:10.1145/1498698.1564504.
- 12 Markus Chimani, Petra Mutzel, and Immanuel Bomze. A new approach to exact crossing minimization. In *Proc. ESA*, volume 5193 of *LNCS*, pages 284–296, 2008. doi:10.1007/978-3-540-87744-8_24.
- 13 Marcel Dhiiflaoui, Stefan Funke, Carsten Kwappik, Kurt Mehlhorn, Michael Seel, Elmar Schömer, Ralph Schulte, and Dennis Weber. Certifying and repairing solutions to large LPs how good are LP-solvers? In *Proc. Fourteenth SODA*, pages 255–256. ACM/SIAM, 2003.
- 14 Giuseppe Di Battista, Ashim Garg, Giuseppe Liotta, Roberto Tamassia, Emanuele Tassinari, and Francesco Vargiu. An experimental comparison of four graph drawing algorithms. *Computational Geometry*, 7(5–6):303–325, 1997. 11th ACM Symposium on Computational Geometry. doi:10.1016/S0925-7721(96)00005-3.
- 15 Giuseppe Di Battista, Ashim Garg, Giuseppe Liotta, Armando Parise, Roberto Tassinari, Emanuele Tassinari, Francesco Vargiu, and Luca Vismara. Drawing directed acyclic graphs: An experimental study. *International Journal of Computational Geometry & Applications*, 10(06):623–648, 2000. doi:10.1142/S0218195900000358.
- 16 Geoffrey Exoo, Frank Harary, and Jerald Kabell. The crossing numbers of some generalized Petersen graphs. *Mathematica Scandinavica*, 48(1):184–188, 1981.
- 17 M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM Journal on Algebraic Discrete Methods*, 4(3):312–316, 1983. doi:10.1137/0604033.
- 18 Petr Hliněný. Crossing number is hard for cubic graphs. *Journal of Combinatorial Theory. Series B*, 96(4):455–471, 2006. doi:10.1016/j.jctb.2005.09.009.
- 19 Michael Jünger and Stefan Thienel. The ABACUS system for branch-and-cut-and-price algorithms in integer programming and combinatorial optimization. *Softw., Pract. Exper.*, 30(11):1325–1352, 2000. doi:10.1002/1097-024X(200009)30:11<1325::AID-SPE342>3.0.CO;2-T.
- 20 Ken-ichi Kawarabayashi and Bruce Reed. Computing crossing number in linear time. In *Proc. STOC*, pages 382–390, 2007. doi:10.1145/1250790.1250848.
- 21 Marián Klešč. The crossing numbers of join of the special graph on six vertices with path and cycle. *Discrete Math.*, 310(9):1475–1481, 2010. doi:10.1016/j.disc.2009.08.018.
- 22 Marián Klešč, R. Bruce Richter, and Ian Stobert. The crossing number of $C_5 \times C_n$. *Journal of Graph Theory*, 22(3):239–243, 1996. doi:10.1002/(SICI)1097-0118(199607)22:3<239::AID-JGT4>3.0.CO;2-N.
- 23 Jan Kratochvíl. String graphs. II. recognizing string graphs is NP-hard. *Journal of Combinatorial Theory, Series B*, 52(1):67–78, 1991. doi:10.1016/0095-8956(91)90091-W.
- 24 Casimir Kuratowski. Sur le problème des courbes gauches en topologie. *Fundamenta Mathematicae*, 15(1):271–283, 1930. URL: <http://eudml.org/doc/212352>.

- 25 Dan McQuillan, Shengjun Pan, and R. Bruce Richter. On the crossing number of K_{13} . *Journal of Combinatorial Theory, Series B*, 115:224–235, 2015. doi:10.1016/j.jctb.2015.06.002.
- 26 Dan McQuillan and R. Bruce Richter. On the crossing numbers of certain generalized Petersen graphs. *Discrete Mathematics*, 104(3):311–320, 1992. doi:10.1016/0012-365X(92)90453-M.
- 27 Lars Noschinski, Christine Rizkallah, and Kurt Mehlhorn. Verification of certifying computations through autocorres and simpl. In *Proc. NASA Formal Methods - 6th International Symposium*, volume 8430 of *LNCS*, pages 46–61, 2014. doi:10.1007/978-3-319-06200-6_4.
- 28 Shengjun Pan and R. Bruce Richter. The crossing number of K_{11} is 100. *Journal of Graph Theory*, 56(2):128–134, 2007. doi:10.1002/jgt.v56:2.
- 29 Marcus Schaefer. The graph crossing number and its variants: a survey. *Electronic Journal of Combinatorics*, 2013.
- 30 Imrich Vrt'ó. Crossing numbers of graphs: A bibliography. <ftp://ftp.ifi.savba.sk/pub/imrich/crobib.pdf>, 2014.