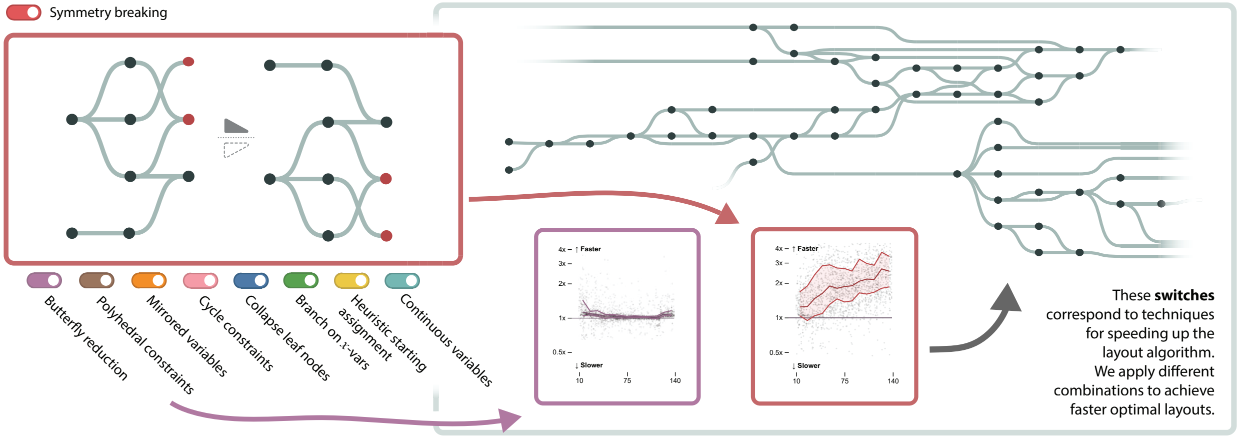
# Evaluating and extending speedup techniques for optimal crossing minimization in layered graph drawings

Connor Wilson, Eduardo Puerta, Tarik Crnovrsanin, Sara Di Bartolomeo, and Cody Dunne



These **switches** correspond to techniques for speeding up the layout algorithm. We apply different combinations to achieve faster optimal layouts.

Fig. 1: We aim to create faster and more scalable methods of finding layered graph layouts with the minimum number of crossings. We characterize nine techniques to improve the performance of an integer linear programming (ILP) formulation and empirically test their improvement. We call these *switches* since they can be toggled and combined. Here, we show the performance of two switches and highlight an optimal control flow graph layout from our case study, with final node placements generated by the bendiness reduction of Di Bartolomeo et al. [9] performed in sequence after crossing minimization (full layouts available at `https://osf.io/5vq79`). These control flow graphs can grow very large but benefit from minimal crossing visualizations to aid human task performance.

**Abstract**—A layered graph is an important category of graph in which every node is assigned to a layer, and layers are drawn as parallel or radial lines. They are commonly used to display temporal data or hierarchical graphs. Previous research has demonstrated that minimizing edge crossings is the most important criterion to consider when looking to improve the readability of such graphs. While heuristic approaches exist for crossing minimization, we are interested in optimal approaches to the problem that prioritize human readability over computational scalability. We aim to improve the usefulness and applicability of such optimal methods by understanding and improving their scalability to larger graphs. This paper categorizes and evaluates the state-of-the-art linear programming formulations for exact crossing minimization and describes nine new and existing techniques that could plausibly accelerate the optimization algorithm. Through a computational evaluation, we explore each technique's effect on calculation time and how the techniques assist or inhibit one another, allowing researchers and practitioners to adapt them to the characteristics of their graphs. Our best-performing techniques yielded a median improvement of 2.5–17× depending on the solver used, giving us the capability to create optimal layouts faster and for larger graphs. We provide an open-source implementation of our methodology in Python, where users can pick which combination of techniques to enable according to their use case. A free copy of this paper and all supplemental materials, datasets used, and source code are available at `https://osf.io/5vq79`.

**Index Terms**—Integer linear programming, layered graph drawing, layered network visualization, crossing minimization, edge crossings

## 1 Introduction

In a **layered graph**, every node in the graph is assigned to a layer and edges connect nodes in different layers. Layered graphs (also called layered networks) are commonly used to represent sequential and hierarchical relationships across a wide variety of domains, including machine learning [33, 54], biology [5], the humanities [20], and more. Usually, layered graphs are shown using node-link visualizations, which means that human task performance in reading them heavily depends on the spatial layout of nodes and edges [2]. Computing this layout is most often done using heuristic algorithms which are fast, but produce suboptimal results, particularly for sparsely connected graphs [31, 46]. Conversely, other methods focus on the optimality of the result but require more time and computational resources and are, therefore, less scalable.

Straightline edge crossing minimization is the most important aesthetic criterion for graph readability [25, 42]. The goal is to create a node-link visualization for a given graph with the fewest number of edges that cross over each other. Traditionally, this is done using heuristics which quickly produce layouts with few crossings. Notably, the Sugiyama framework for graph visualization [46] breaks this into steps: nodes are first assigned to layers, and are reordered within layers using a barycentric method to produce a drawing with few crossings. This approach creates very readable graph visualizations, and is widely used in graph drawing libraries [4, 17].

Typically, methods are implemented with the goal of balancing the runtime and optimality of the layouts generated, but in this paper we focus on the creation of layouts with a provably-optimal number of overlapping edges. Not only does the optimal crossing minimization approach create more readable graphs, but it can also be used to

- *The authors are all with Northeastern University. E-mail: [ wilson.conn | puerta.e | t.crnovrsanin | dibartolomeo.s | c.dunne ]@northeastern.edu.*

*Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx*

benchmark the speed of heuristic layout algorithms and the readability of their layouts. However, optimal layouts can take a long time to compute, making heuristics more suitable for larger graphs.

This paper focuses on speeding up the creation of provably-optimal layouts using **Integer Linear Programming (ILP)**. The advantage of ILP formulations is that they allow us to use powerful existing solvers, many of which are commercially available and very quick in practice. As ILP techniques and computer speeds improve, the computational cost of these exact methods approaches the realm of practicality for larger and larger graphs, leading researchers to speculate that optimal techniques will eventually replace heuristics for layered graph layouts [16]. Another benefit is that ILP is extremely flexible—we can combine constraints for whichever criteria we desire into a combined ILP model and explicitly input how much we want the result to favor one criterion over the others [9, 16]. Finally, ILP is uniquely positioned to solve specific problems, such as the layout of sparse graphs, which are often in greatest need of minimal crossing visualization due to the increased importance of individual edges. While heuristics perform worse on sparser graphs, ILP's performance skyrockets [31].

In this paper, we present two standard formulations for layered crossing minimization ILP and analyze nine techniques for improving runtime across a range of input sizes. To our knowledge, no prior research has reported on the use of two of these nine techniques, nor has any research empirically validated any of the techniques or transitivity formulations we present. To make optimal layered crossing minimization feasible on larger graphs, we categorize existing approaches and classify the performance of each of nine techniques for each of two transitivity types on a 3,200-graph dataset. We then perform a comprehensive benchmark study of 1,280 different ILP formulations on a dataset of 1,700 graphs, to compare the combined performance benefits of the different techniques. Finally, we illustrate our formulation with a case study on visualizing code control flow and compare it with existing formulations in this application.

Specifically, this paper contributes to layered graph visualization:

1. **An empirical comparison of nine new and existing techniques** for faster exact crossing minimization and their scalability to larger graphs, using two different base formulations and two ILP solvers,
2. **A case study** demonstrating the practicality of our recommended method applied to control flow graphs, and
3. **An open-source implementation** of our exact crossing minimization algorithm in Python for both Gurobi 10.0 [21] (license required) and HiGHS 1.5 [26] (fully open source), available on OSF at `https://osf.io/5vq79`.

## 2 Background and Related Work

In the following section, we discuss relevant extant layered graph drawing algorithms. We also survey important heuristics, and describe notable optimal algorithms, including ILP formulations and optimization approaches. A review of general graph visualization and graph drawing is outside the scope of this discussion. However, for an introduction, see Tamassia's book [47], Gibson et al.'s survey [19], and the following systematic review of computational evaluations of graph layout algorithms [7].

### 2.1 Layered Graphs

A *layered graph* $G$, also referred to as a *layered graph*, is a structure consisting of a set of vertices $V$ (also referred to as nodes) and edges $E$, where each edge connects two vertices, and a layering $L$. The *layering* is a function $L : V \rightarrow \{1,2,3,\ldots,K\}$, where $K$ is the number of layers in $G$. We call a layered graph *proper* if, for all $(i, j) \in E$, we have $|L(i)-L(j)| = 1$. That is, all edges in a proper layered graph are between consecutive layers, an important property for the use of the formulations outlined in this paper. When graphs contain edges traversing more than one layer, we split them up into sections using *anchor or dummy nodes* as per Gansner et al. [17]. We refer to the number of nodes in the graph after this dummy node insertion as *total nodes*. We then consider a *drawing* to be an assignment of nodes to 2-dimensional coordinates, where nodes in the same layers are drawn aligned, and these layer lines are parallel and ordered consecutively. For the purposes of layouts in this paper, layers are drawn vertically.

Layered node-link graph visualizations are used to represent hierarchical and sequential relationships between entities [23]. These are useful for visualizing medical time series [1], SQL queries [9], and navigation techniques [24], for example. Layers can be provided as part of the data or assigned as a step in the algorithm. Many criteria influence the readability of these visualizations. Purchase found edge-crossing reduction to have the most significant impact on the readability of relations in a graph [42]. In later work, she formalized seven metrics to measure the aesthetic criteria that influence the readability of graph layouts [43]. Hence, layout algorithms range from optimizing some of these criteria, such as minimizing edge bends [9,46], to constraining the design conventions of specific domains, like calculating readable metro maps [37,41]. While many of these metrics are studied, we focus on crossing minimization as it is the most important metric for many tasks [42, 53].

Many fast but suboptimal heuristic algorithms have been proposed to minimize edge crossings for layered graphs. Among the most prominent, the Sugiyama framework [23, 45, 46] consists of assigning layers to a graph and then sweeping through them to permute nodes with a barycenter heuristic. Eades and Wormald proposed a similar method in 1994, with a median heuristic to compute the positions of nodes [14]. In their paper, they also utilized dummy nodes to ensure multilevel graphs have a proper layering. This technique is still commonly used and allows us to make formulations for layered graphs that would otherwise not be proper. A few years later, Eades, alongside Lin and Tamassia, presented a degree-weighted barycentric method [13]. All these methods focus on iteratively sweeping through layers and computing "average" positions of nodes, with the main differences among them being how they define average. The original barycentric heuristic [46] has exceptional performance [31] and is often implemented in graph visualization libraries [27]. Matuszewski et al. [36] propose a $k$-layer sifting method that often outperforms the barycentric method at the expense of being slower.

### 2.2 Optimal Algorithms for Layered Graphs

Alternative to heuristic approaches, graph layouts can be computed by using precise mathematical expressions, which can be solved for provably optimal solutions. However, finding a minimum crossing drawing for a layered graph is NP-Complete [18] due to the combinatorial nature of reordering nodes. Hence, many approaches in the literature have focused on improving the performance and scalability of existing algorithms. In presenting their framework, Suyigama et al. proposed exact algorithms for permuting nodes within layers subject to a prior layer assignment step. For the exact layout, they designed their objective function to minimize the vertical distance between every pair of nodes at the end of each edge, which is a quadratic optimization problem [46]. Gansner et al. added more constraints to linearize the method proposed by Sugiyama [17].

**Optimal Formulations using ILP.** Jünger and Muntzel proposed a branch-and-bound algorithm to minimize crossings in 2-layer graphs [30]. That same year Valls et al. also proposed another branch-and-bound method [50].

We call a *formulation* a way to define a desired property as a set of integer variables and linear constraints. An *ILP model* is an encoded formulation (or multiple formulations) with a corresponding optimization goal given to an ILP solver. ILP solvers programmatically find an integer assignment to the variables that satisfy all constraints such that the optimization expression is minimized/maximized. ILP solvers leverage years of research [28], and commercial solvers such as Gurobi report yearly performance increases [21]. Therefore, we expect ILP solvers' increased performance to translate to more viable and scalable optimal formulations. This paper aims to compare the performance of existing ILP models by combining techniques that alter the formulations themselves or influence the solver. Below, we discuss the varied formulations and optimization techniques found in the literature.

**Extant ILP Formulations.** In 1997, Jünger and Mutzel proposed an ILP formulation to permute the nodes in the 2-layer case to reduce crossing numbers [30]. They further developed a multilayer approach [31] using the dummy node strategy. Their formulation directly ensures the transitivity of nodes by outlining constraints on the vertical position variables for each pair of nodes. Di Bartolomeo et al. presented another notable formulation in Stratisfimal Layout [9]. While many other formulations

Table 1: Notation Used in This Paper

| | |
|---|---|
| $G=(V,E,L)$ | A layered graph $G$ with vertex set $V$, edge set $E$, and layer assignment on the vertices $L:V\to\{1,2,...,K\}$. |
| $V'$ | Transformed node set of $G$ with dummy nodes inserted on long edges and $\lvert V'\rvert$ *total nodes*. |
| $E_r$ | All $(u,v)\in E$ such that $L(u)=r$ and $L(v)=r+1$. |
| $K$ | Number of layers in $G$. |
| $x_{i,j}$ | Binary variable denoting the relative position of nodes $i$ and $j$ in the layout. |
| $c_{(i,k),(j,l)}$ | Binary variable denoting if edges $(i,k)$ and $(j,l)$ cross. |

ensured transitivity via direct constraints, Stratisfimal Layout implies this property by assigning indexed positions to nodes within layers when minimizing edge length. Indices are natural numbers, which are transitive on the "greater than" relation ($>$). Di Bartolomeo et al. formalized vertical position constraints as a set of implications relating direct transitivity constraints to node positions. They proceed to linearize these implications with a conversion that introduces auxiliary variables to provide constraints for the ILP. Stratisfimal Layout still implements the direct transitivity constraints, which creates redundancy in their formulation.

**Notable Speedup Approaches.** The corpus of ILP formulations also consists of strategies to improve the solver's performance. To our knowledge, we are the first to empirically evaluate these when combined and compared across different transitivity formulations. Gange et al. provide a comprehensive evaluation of ILP and MIP solutions for layered crossing minimization using a few of the switches outlined in our work [16]—the closest computational evaluation to ours in scale of anything we could find in the literature. They propose symmetry breaking, collapsing leaf nodes, and an improved set of constraints based on the vertex exchange graph introduced by Healy and Kuusik [22] as potential improvements.

Mutzel proposed leveraging the planarity of graphs on their formulation by considering constraints based on subgraphs that guarantee the existence of a crossing [39]. Their work started with the 2-layer case [40] and was later extended to $k$-levels by Healy and Kuusik [22]. Notably, Mutzel described how cycles in 2-layer graphs guarantee the existence of crossings. Their approach involved creating constraints that checked for the existence of such subgraphs. Conversely, we use this insight to remove crossing constraints with known solutions.

Zarate et al. abstracted Sankey Diagrams as layered graphs and propose an ILP approach for their layouts [56]. They added symmetric variables on crossings to hone in on possible solutions to increase performance. Some ILP literature discusses how adding redundant variables can help the formulation determine solutions faster [32]. They noticed that if there is a crossing between two vertices, it does not matter which vertex we consider the first one in our notation the crossing is still there. However, adding redundancy is theoretically justified but not experimentally verified. They also proposed enhancing the solver's performance by branching on the binary position variables ($x$-vars). Branch-and-bound techniques are often used in ILP problems to find candidate solutions [38]. Their formulation included a crossing constraint, with $x,c\in\{0,1\}$. However, they highlight that since $x$-vars constraints are binary, the crossing variables ($c$-vars) must be integers. Therefore, they mention they do not have to enforce the constraints that c-vars are also binary since the crossing constraints imply this. Nonetheless, they claim that adding branching priorities on x-vars is more efficient, telling the solver these are more important in finding the solution space.

Some of these optimizations inspire the "switches" used for the different experiments we run to evaluate the performance of different models, which we further discuss in their respective sections. We also compare the formulations of Jünger et al. [29], Zarate et al. [56], Gange et al. [16], and Di Bartolomeo et al. [9] to our own in the context of a case study.

# 3 Methodology

Modern ILP solvers allow complex problems to be solved much more quickly than brute force, provided the problem can be formulated as a set of linear constraints. Exact layered crossing minimization is one such problem. We first define the standard formulation for layered crossing minimization, which is a way to transform any input graph into a set of variables and linear constraints. This formulation has the property that if we can find the assignment to the variables with minimum crossing variable sum, it will correspond to the layout of the graph with the minimum possible edge crossings. We then describe two variations of this formulation which we will later compare empirically. Next, we detail nine "switches", which modify the formulation and ILP solver in ways that could improve the speed of exact crossing minimization while still returning an optimal solution.

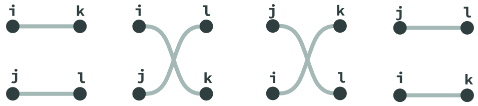
## 3.1 ILP Formulation

Existing techniques for formulating layered crossing minimization using ILP follow the same standard template. We assume a proper layered graph $G$ is given, with a layer assignment $L:V\to\{1,2,...,K\}$ for every node in $V$, such that all edges in $E$ connect nodes in different layers. Given $G$, we say the following variables and constraints define the standard model $M_G$:

- ***Position variables***, denoted $x_{i,j}$, corresponding to two nodes $i$ and $j$. These are defined for every pair of nodes in the same layer. If $i$ is drawn above $j$ in that layer we write $x_{i,j}=1$, and if $i$ is below $j$ we write $x_{i,j}=0$.
- ***Crossing variables***, denoted $c_{(i,k),(j,l)}$, corresponding to whether or not edges $(i,k)$ and $(j,l)$ cross in the drawing of the graph. They are defined when $i,j,k$, and $l$ are all unique, for every pair of edges between the same consecutive layers. If $(i,k)$ and $(j,l)$ cross, we write $c_{(i,k),(j,l)}=1$; if they do not, we write $c_{(i,k),(j,l)}=0$.
- ***Crossing constraints***, which are given in [9, 56], and in [29] as a set of inequalities equivalent to:

$$\begin{aligned} c_{(i,k),(j,l)}+x_{j,i}+x_{k,l}\geq 1\\ c_{(i,k),(j,l)}+x_{i,j}+x_{l,k}\geq 1 \end{aligned} \tag{1}$$

These are defined for every crossing variable $c_{(i,k),(j,l)}$, and enforce $c_{(i,k),(j,l)}\geq 1$ if $(i,k)$ and $(j,l)$ cross. This concept is illustrated here:



The first equation enforces $c_{(i,k),(j,l)}=1$ when $i$ is above $j$ but $k$ is below $l$ (illustration 2 above); the second enforces $c_{(i,k),(j,l)}=1$ when $i$ is below $j$ and $k$ is above $l$ (illustration 3). Note that these constraints can be added only if $G$ is a proper layered graph.
- ***Transitivity constraints***, see Sec. 3.2.

When combined into an ILP model, valid assignments to position and crossing variables correspond to a layout of the input graph. The sum of all crossing variables is the number of crossings in this drawing. Therefore, the assignment to these variables with the smallest sum

$$OBJ=\sum_{r=1}^{K-1}\sum_{e_1,e_2\in E_r} c_{e_1,e_2} \tag{2}$$

represents a layout with the fewest crossings. This is the *objective function* the ILP solver seeks to minimize. The goal of the ILP solver is to assign binary values to all $x$- and $c$-variables such that the crossing and transitivity constraints are satisfied and the sum Eq. (2) is minimal.

So, given some input graph, we encode it using the above constraints to form an ILP model. This is passed to an ILP solver, which works to find assignments minimizing the objective function $OBJ$, and we can post-process these assignments to create and visualize the crossing-minimized drawing.

## 3.2 Transitivity Constraints

Since node placements are encoded using their position relative to other nodes instead of absolute position, it is necessary to include constraints that prevent non-transitive assignments. In order for the variable assignments produced by the ILP solver to correspond to a valid layout, there must be additional constraints that ensure the $x$-variables are not assigned values by the ILP solver which disobey transitivity. For instance, if we start with a graph that has three nodes $i$, $j$, and $k$ in the same layer, nothing stops our ILP solver from setting $x_{i,j}=1$, $x_{j,k}=1$, and $x_{i,k}=0$. But if we decode what this is actually saying, we find out that the ILP solver has just told us to draw $i$ above $j$, $j$ above $k$, and $k$ above $i$—this is impossible since we cannot draw $i$ simultaneously above and below $k$.

There are multiple ways of encoding this transitivity relationship as an ILP constraint, which we will compare empirically.

### 3.2.1 Direct transitivity constraints

The original formulation of Jünger et al. [29] enforces transitivity directly on all triples of nodes in the same layer:

$$\begin{aligned} x_{i,j}+x_{j,k}-x_{i,k} &\leq 1 \\ x_{i,j}+x_{j,k}-x_{i,k} &\geq 0 \end{aligned} \tag{3}$$

Equation (3) enforces the relationships $x_{i,j}=1 \wedge x_{j,k}=1 \Rightarrow x_{i,k}=1$ and $x_{i,j}=0 \wedge x_{j,k}=0 \Rightarrow x_{i,k}=0$, which comes directly from the definition of a transitive relation. That is, if we have that $i$ is above $j$ and $j$ is above $k$, we require that $i$ is above $k$. Similarly for the converse, if we have $i$ below $j$ and $j$ below $k$, we require that $i$ is below $k$. These are added to the ILP model for every combination of nodes $i,j,k$ in the same layer, guaranteeing a transitive relationship across the entire layer. This generates $O(|V|^3)$ constraints.

### 3.2.2 Vertical position transitivity

Di Bartolomeo et al. [9] define an additional integer-valued variable $y$ for each node representing the vertical position of the node when drawn. For this to work, we require constraints that ensure that assignments to $x$-variables (the relative positions of nodes) are consistent with assignments to $y$-variables (the absolute position of each node). The following inequalities from Di Bartolomeo et al. accomplish this:

$$\begin{aligned} z_{i,j}-M\cdot x_{i,j} &\leq 0 \\ z_{i,j}-y_i-M\cdot x_{i,j} &\geq -M \\ y_j-z_{i,j}-x_{i,j} &\geq 0 \\ z_{i,j}-y_i &\leq 0 \\ z_{i,j} &\geq 0 \end{aligned} \tag{4}$$

These constraints implicitly enforce transitivity on the $x$-variables, and the authors also use them to relate the $x$-variables to vertical position $y$-variables to better control the placement of nodes in the resultant layout. $M$ is a fixed upper bound which is set to the size of the largest layer. Eq. (4) also requires an additional variable $z_{i,j}$ for each variable $x_{i,j}$, and adds $O(|V|^2)$ constraints to the model in total. For large graphs, this adds far fewer constraints than Eq. (3). Advanced solvers such as Gurobi [21] can perform this linearization automatically provided $x_{i,j}=1 \Longrightarrow y_i<y_j$ and $x_{i,j}=0 \Longrightarrow y_j>y_i$ for each $x_{i,j}$, the constraints encoded by Eq. (4).

## 3.3 Switches

Having defined the ILP formulation for crossing minimization, we now detail nine techniques we call "switches". In contrast to formulations, switches can be toggled on and off but never alter the value of the solution found by the model. *Switches* are defined as modifications to the variables and constraints in an ILP model, or alterations to the behavior of the ILP solver, in a way that does not affect the optimal solution.

The purpose of these switches is to try and improve the runtime of the ILP solver and its ability to scale to larger graphs. This is accomplished by restricting the space of all possible solutions or giving hints to the solver that guide it toward the optimal solution. Due to the nature of exponential growth, ILP models will always reach a point where the inputs get too big, causing runtime to rapidly increase, thereby dramatically decreasing the feasibility of solution-finding for graphs past a certain size. Our goal in defining these switches is to try to reduce the time complexity of crossing minimization, even if some additional overhead is required. This expands the range of feasible graph sizes for the optimizer. To that end, we first summarize the action behind each switch in Tab. 2 before describing them in detail and explaining our intuition for why it may improve performance.

Table 2: Switch Descriptions

**1. Symmetry breaking:** Select the $x$-variable which appears in the most crossing constraints and fix it to be 0 before optimizing the model.
**2. Butterfly reduction:** For each crossing variable $c_{(i,j),(k,l)}$ whose edges form a butterfly, add the constraint $c_{(i,j),(k,l)}+c_{(i,l),(k,j)}=1$.
**3. Polyhedral constraints:** For each 2-layer 3-claw motif $W$, add the constraint $\sum_W c_{(i,j),(k,l)} \geq 1$, and add the additional dome-path constraints.
**4. Mirrored variables:** Add both $x_{i,j}$ and $x_{j,i}$ to the model for all pairs of nodes $i,j$ in the same layer, and implement the symmetry constraint $x_{i,j}=1-x_{j,i}$.
**5. Cycle constraints:** For each fundamental cycle $C$ in the vertex exchange graph, add constraints $2k_C=\sum_{e\in C} c_e$ for even-labeled and $2k_C+1=\sum_{e\in C} c_e$ for odd-labeled cycles.
**6. Collapse leaf nodes:** Replace leaf nodes in the same layer with one single node, and set the weight of its edge to the number of leaves removed.
**7. Branch on $x$-variables:** Set the ILP solver branching priority on the $x$-variables to the highest level.
**8. Heuristic starting assignments:** Perform the iterated barycenter heuristic on the input graph, and assign starting values to the solver variables according to this initial layout.
**9. Continuous variables:** Explicitly define the crossing variables $c_{(i,k),(j,l)}$ and vertical position variables $y_i$ (if used) in the model to be positive real numbers instead of integers.

### Switch 1: Symmetry breaking

Gange et al. introduce symmetry breaking in their model, which is the process of selecting a single $x$-variable and fixing it to 0 prior to optimization [16]. This works because there is functionally no difference, most importantly in crossing number, between a layered graph layout and the upside-down version of the same layout. This is visible in Fig. 1, where the two layouts of the graph at top-left are functionally the same. So, we can fix $x_{u,v}=0$ before optimizing the model without repercussion. So, fixing one single variable breaks this symmetry and eases the burden on the ILP solver by one decision variable.

Gange et al. implement this by selecting the $x$-variable that corresponds to the first pair of nodes in the same layer. We extend this technique by selecting not the first $x$-variable, but the $x$-variable that appears in the greatest number of crossing constraints. This gives the solver a starting point to leverage for finding the optimal assignment to the other variables and simplifies the model as much as possible.

### Switch 2: Butterfly reduction

This switch involves crossing variables whose edges are part of a $2\times 2$ biclique, or *butterfly* [44]. This graph motif is 2-level non-planar [39]. Zarate et al. [56] make use of this fact by describing an additional ILP constraint for any butterfly $\{(i,j),(k,l),(i,l),(k,j)\}\subseteq E$:

$$c_{(i,j),(k,l)}+c_{(i,l),(k,j)}=1 \tag{5}$$

In Fig. 2a we can see that swapping the within-layer positions of nodes $k$ and $l$ does not change that the graph has exactly one crossing. Likewise for swapping the positions of $i$ and $j$.

### Switch 3: Polyhedral constraints

Jünger et al. study the polytope associated with the solution space of the layered crossing minimization problem [29], and derive several classes of constraints which are "facet-defining", meaning they restrict the solution space of the crossing minimization problem as much as possible.

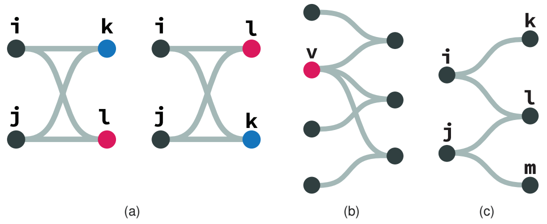


Fig. 2: Butterfly reduction pictured in (a), 3-claw motif and dome-path respectively pictured in (b) and (c).

This switch adds all facet-defining constraints described by Jünger et al., with the exception of two sets of constraints which are cycle-based. The findings detailed in the appendices at https://osf.io/5vq79 and the recommendation of Gange et al. for cycle-based constraints [16] suggest that these constraints do not significantly improve solution time. Furthermore, switch 2 already studies constraints regarding the 4-cycle.

The only other minimal non-planar 2-layer motif besides the butterfly is the *3-claw* [39], shown in Fig. 2b. Jünger et al. describe additional constraints for any occurrence of this subgraph [29].

$$\sum_{e_1,e_2\in W} c_{e_1,e_2} \geq 1 \tag{6}$$

For any pair of edges $e_1,e_2 \in W$ such that the vertex point $v$ of $W$ is part of exactly one of the two edges, for all 3-claws $W$ in $G$.

The other set of constraints involves *dome-paths* (Fig. 2c), which are added for any 2-layer path on four edges and five nodes $i,j,k,l,m$:

$$\begin{aligned} x_{k,l} - 2x_{k,m} + x_{l,m} - c_{(i,k),(j,l)} - c_{(i,l),(j,m)} &\leq 0 \\ -x_{k,l} + 2x_{k,m} - x_{l,m} - c_{(i,k),(j,l)} - c_{(i,l),(j,m)} &\leq 0 \end{aligned} \tag{7}$$

### Switch 4: Mirrored variables with symmetry constraints

This switch doubles the number of variables by adding both $x_{i,j}$ and $x_{j,i}$ to the model. Without this switch on, we assume only one of the two is added, say $x_{i,j}$. If a constraint requires $x_{j,i}$ we instead substitute $1 - x_{i,j}$ (the negation of $x_{i,j}$). The statement "$i$ is above $j$" being false implies that "$i$ is below $j$" is true, hence this substitution is valid. Turning this switch on also implements the following symmetry constraints from Zarate et al. [56]: $x_{i,j} = 1 - x_{j,i}$ and $c_{e_1,e_2} = c_{e_2,e_1}$ for all $x$- and $c$-variables, enforcing the symmetric relationship of the mirrored variables.

This doubles the number of variables used in the model; however, redundancy in ILP models has been shown to sometimes improve solver performance [32]. Other previous works typically follow the precedent of Jünger et al. [29] who do not use mirrored variables, so for the purpose of this paper we use the more minimal model as default.

### Switch 5: Cycle constraints

Healy and Kuusik describe a structure called the vertex exchange graph which gives rise to additional ILP constraints [22]. The vertex exchange graph is created from an input graph $G$ by defining a node $\langle u_1,u_2\rangle$ for every same-layer node pair in $G$ (i.e., one node for every $x$-variable in the standard ILP model) and connecting pairs of nodes $\langle u_1,u_2\rangle,\langle v_1,v_2\rangle$ with an edge if $(u_1,v_1)$ and $(u_2,v_2)$ are edges in $G$ (thus the edges correspond with $c$-variables in the ILP model). Cycles in this new graph are *odd-labeled* if the sum of the $c$-variables of the cycle edges is odd and *even-labeled* otherwise. If this switch is on, add the following constraints for each fundamental cycle $C$ in the vertex exchange graph:

For odd-labeled cycles $C$:

$$\sum_{e\in C} c_e \geq 1 \tag{8}$$

For even-labeled cycles $C$:

$$\sum_{e\in C} c_e \leq |C| - 1 \tag{9}$$

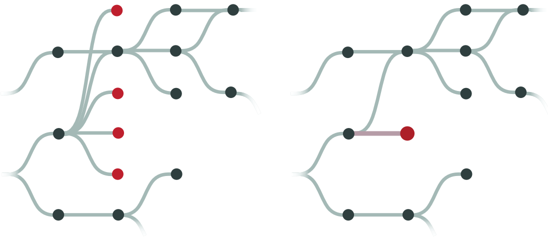


Fig. 3: Example of leaf node collapse. The left graph contains a node with four leaves, highlighted in red, which are combined into a single node as shown on the right.

These a reduced set of the original paper's [22] constraints per the recommendation of Gange et al. [16] (see the appendices at `https://osf.io/5vq79` for an evaluation supporting this recommendation).

### Switch 6: Collapse leaf nodes

Introduced by Gange et al. [16], this switch is unique from the other switches in that it directly modifies the input graph. The principle of the technique is to select a leaf node subgraph—two or more nodes connected only to a single parent node—and remove all the leaf nodes, replacing them with a single "collapsed" node and edge. Once the crossing minimized layout is found, the nodes in this subgraph are re-inserted back into the graph by squeezing them all into the spot left by the collapsed node. To ensure optimality, the objective function in Eq. (2) must be updated to

$$OBJ = \sum_{r=1}^{K-1} \sum_{e_1,e_2\in E_r} w_{e_1} w_{e_2} c_{e_1,e_2} \tag{10}$$

where $w_e$ is the weight of the collapsed edge and $w_e = 1$ if $e$ is not a collapsed edge. This procedure works because if it is optimal to place one leaf node in a certain position, then any other leaf nodes must also be optimally placed if they are immediately adjacent—all leaves will incur the same number of crossings.

### Switch 7: Branching on $x$-variables

This switch sets the branching priority of all $x$-variables to 1 and all other variables to 0. ILP solvers search for potential solutions by *branching* on a decision variable, which involves looking at the potential solutions when the branch variable is fixed. At each branching point, a variable is selected randomly from the set of variables with the highest branching priority that has not yet been branched on [21]. Zarate et al. claim this yields up to 10× performance improvement for large instances [56].

### Switch 8: Heuristic starting assignments

This switch uses the straightforward iterated degree-weighted barycenter heuristic [13] to provide an initial starting assignment for the solver. ILP solvers perform an iterative process of finding assignments to the variables which progressively get closer to the optimal solution, but modern solvers allow the user to input a valid starting assignment to the variables. This is not the same as fixing the variable, as is done in switch 1, because the solver can change the assignments made by the starting value parameter.

The barycenter heuristic is widely used for crossing minimization on layered graphs We use it as a reasonable starting point for the ILP solver. Providing a starting point is suggested to speed up solution time as it potentially skips early iterations of the ILP algorithm [21].

### Switch 9: Continuous variables

Mixed-integer programming (MIP) solvers have traditionally been used to solve layered crossing minimization, utilizing a branch-and-cut approach [29]. Confusingly, the terms ILP and MIP are sometimes used synonymously. With modern solvers, however, users can define the integrality of each variable in the model explicitly, and the solver selects the best algorithms to use on the back end. This switch makes sure the

Table 3: Datasets Used for Empirical Study

| Experiment | Graphs | $\|V'\|$ | Per Bin | By Collection | |
|---|---|---|---|---|---|
| | | | | Rome-Lib | AT&T |
| Ind. switches | 3170 | 10-399 | 100 | 2959 | 211 |
| All combos. | 1713 | 10-399 | 50 | 1615 | 98 |

$x$-variables are kept as binary variables, but the crossing $c$-variables and vertical position $y$-variables (if vertical position transitivity is being used) are explicitly defined to be real-valued.

This does not affect the optimal solution of the model:

- Equation (1) ensures $c \geq 0$ if $c$'s edges do not cross and $c \geq 1$ if they do. Since our objective is to minimize the sum of all $c$-variables, real-valued $c$-variables will still all converge to either 1 or 0 in the optimal solution.
- "$<$" is a transitive relation on the real numbers and the integers, so Eq. (4) still enforces transitivity on real-valued $y$-variables.

A number of solvers have made advancements in algorithms for mixed-integer programming (MIP)problems, with Gurobi 10.0 claiming 24% improvement over the previous release for large models.[1] It may improve solution time if we can expressly define some variables in our model to be real-valued since MIP algorithms may be more efficient than ILP algorithms for the same problem [51], or signal to the solver a more efficient way to treat these continuous variables.

## 4 Experiment

We will now analyze the effect these nine switches and the transitivity formulations have on optimization time via a computational evaluation.

### 4.1 Dataset

The experiment dataset was composed of graphs from standard benchmark collections AT&T [10] and Rome-Lib [11], detailed in Tab. 3. Graphs used are available from the Graph Drawing Benchmark Datasets Repository [8].

For our first experiment on individual switch evaluation, we combined the entire AT&T collection of 1,276 graphs with the Rome-Lib collection of 11,528 graphs and randomly sampled 100 graphs for every 10-node interval up to 400 total nodes. Past the [270,280) interval there were fewer than 100 graphs per 10-node interval, so no sampling was necessary. In practice, no run of the experiment completed the full dataset without being cutoff due to timing out. Note that graph size $|V'|$ used for the sampling is in terms of the post-processed number of vertices, see Sec. 4.2.1. This dataset contains 3,170 graphs and is included at https://osf.io/5vq79 with the code that performed the sampling.

For the second experiment evaluating all combinations of switches, we performed the same steps as for the previous experiment, with the exception that we sampled 50 graphs per 10-node interval up to 400 total nodes. Sampling was necessary up to interval [310,320), and as before, no run of the experiment completed the full dataset. The dataset for the all-combinations experiment contains 1,713 graphs.

### 4.2 Procedure

We now discuss how we pre-processed and modified the graphs to create proper layered graphs. We also detail the evaluation conducted for individual switches and all combinations of switches.

#### 4.2.1 Graph pre-processing

The Rome-Lib [11] and AT&T [10] graphs do not have a predetermined layer assignment, hence one must be created for each graph.

First, we interpret the input graph as directed by assuming the first node in each edge is the source and the second is the target for the Rome-Lib graphs. The AT&T graphs are already both directed and acyclic [10]. We then applied the greedy cycle removal heuristic of Eades et al. [12], followed by assigning layers using the minimum width layering heuristic of Tarassov et al. [48]. They suggest using the layering with minimum width across all combinations of input parameters $UBW = 1,2,3,4$ and $c = 1,2$, which are tuning parameters specific to the minimum-width algorithm. We use only $UBW = 4$ and $c = 2$ as we do not require perfect width minimization, and these parameter choices still consistently produce layered graphs with "rectangular" shapes. I.e., they had a similar number of nodes in each layer. This rectangular shape is desirable as it better approximates real-world layered graphs such as Storyline graphs and time-series data [1, 20], and gives the drawing a conventional aspect ratio without much unused space. More trivial approaches, such as assigning layers based on each node's level in the tree created by a breadth-first search, tend to have a few layers with many more nodes than the others. Finally, we added back the edges removed by the cycle removal step to re-create the original graph topology.

After pre-processing to ensure we had layered graphs, we ensured each graph was proper by replacing edges that skip layers by adding dummy nodes, following the procedure described by Eades and Wormald [14]. That is, if $(u,v)$ is a long edge, where $L(u)+1 < L(v)$, we remove edge $(u,v)$ and add nodes $d_1, d_2, \ldots d_r$ where:

$$\begin{aligned} L(d_1) &= L(u)+1 \\ L(d_2) &= L(u)+2 \\ &\vdots \\ L(d_r) &= L(u)+r = L(v)-1 \end{aligned}$$

We then add dummy edges $(u,d_1),(d_1,d_2),\ldots,(d_r,v)$.

This process transforms the graph into a proper graph so that crossing constraints can be imposed correctly. We refer to the number of nodes in this processed graph (including dummy nodes) as total nodes, which may be larger than the number of nodes in the original graph. We henceforth report data using total nodes as the independent variable, since we find that it more closely correlates with solver runtime than the number of original nodes.

The resultant graph is transformed into its corresponding set of linear constraints as described in Sec. 3.1, and they are modified according to the choice of switches. These are provided to Gurobi 10.0, a state-of-the-art linear programming solver, which computes the optimal layered graph drawing.

#### 4.2.2 Individual switch evaluation

For both transitivity formulations, we ran a baseline experiment with all switches disabled. We also evaluated the effect of turning on only one of the switches at a time. Therefore, we performed 18 runs of the experiment for each pair of the two transitivity formulations and nine switches. With the baseline runs, this left us with 20 sets of results.

Each run comprised solving every graph in the experiment dataset (Secs. 4.1 and 4.2.1) and recording the solver runtime. We separated graphs into bins by their total nodes in 10-node intervals. When less than 75% of the graphs in a bin completed optimization within the cutoff time of 5 minutes, the run was halted. The two baseline runs were continued for every graph completed in the nine individual-switch runs for that formulation. This ensured all data points in the experiment had a corresponding baseline data point for comparison.

#### 4.2.3 Evaluation of all switch combinations

To find the best-performing sets of switches, and to make comparisons between the different transitivity formulations, we ran an experiment for all 1024 combinations of switches ($2^9$) and formulations ($\times 2$). We used a smaller set of graphs to run the experiment within the time available. All 1024 formulation-switch combinations were run on the all-combinations dataset of 1,713 graphs (Tab. 3), with the experiment halted once fewer than half of the graphs in any 10-node bin were solved within the cutoff time. The 5-minute cutoff time per graph was changed to a shorter 1-minute cutoff for this experiment to reduce the number of compute-hours to perform the experiment. Additionally, the halting condition was relaxed for this experiment to better compare switch combinations. Instead, all combinations of switches were evaluated on all graphs solved by the best-performing combination.

[1] https://www.gurobi.com/whats-new-gurobi-10-0/

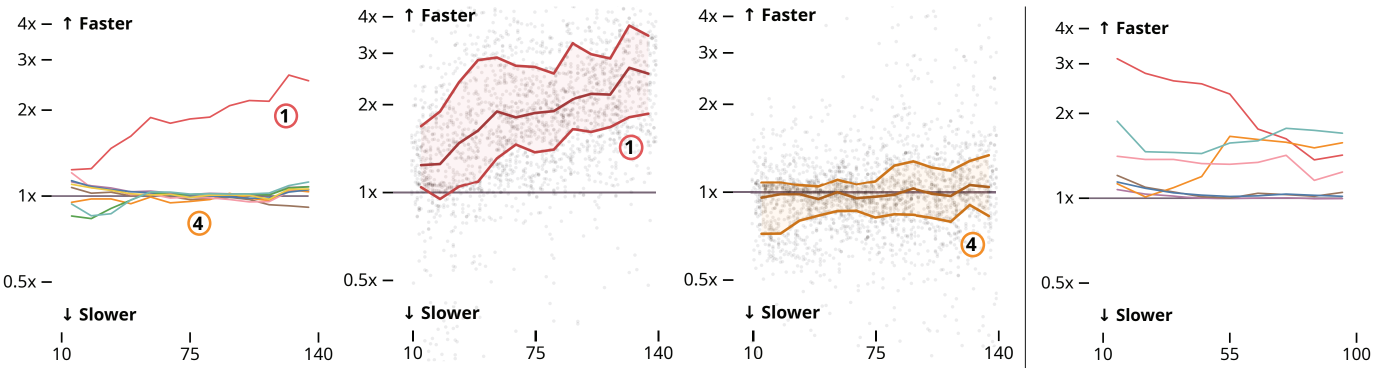


Fig. 4: The left figure shows performance improvement over the baseline for individual switches. All figures plot median runtime as a natural log ratio over the respective baseline, where line color corresponds to the color of the switch. The two center figures show more detailed information for the symmetry breaking (1) and mirrored variables (4). The lines are 25% and 75% quartiles, and the darker center line is the median. Data is truncated once fewer than 75% of the graphs per 10-node interval are solved within the 5-minute cutoff by both the switch technique and the baseline. Symmetry breaking gives a large performance improvement which increases as input sizes grows, while mirrored variables also improved with input size but hurt performance on average for most sizes tested. On the right, individual switch results for the HiGHS solver are shown, where symmetry breaking, continuous variables, mirrored variables, and cycle constraints all perform well. All figures shown combine the direct transitivity and vertical position transitivity results. See appendices at `https://osf.io/5vq79` for more in-depth results figures.

### 4.2.4 Evaluation using different solver

We repeated both of the experiments described above using HiGHS, an open-source linear optimization solver [26]. Heuristic start and *x*-var branch priority are unavailable due to the required functionality being not yet available for HiGHS, and so were left out of both experiments. All experiment parameters were otherwise kept the same.

### 4.2.5 Analysis

We report results for these experiments by calculating, for each 10-node bin, the natural logarithm of the ratio of mean experimental runtime to the mean baseline runtime. Natural log ratios are additive, symmetric, normed indicators of relative change [49], allowing us to more easily compare the impact of a switch on different formulations.

**Units.** Each data point is reported in *optimization time*, the time in seconds it took the ILP solver to determine the optimal solution, with independent variable total nodes. This ignores setup and pre-processing time, which took at most a few seconds for the largest graphs.

**Hardware/software.** We ran the experiments in CentOS Linux with 8 GB RAM & Gurobi 10.0 [21]. For the secondary solver, we used HiGHS 1.5 [26] available through SciPy 1.10's `linprog`.

## 4.3 Discussion and Results

In the following section we discuss key takeaways and ramifications of our experimental results. We also outline ways practitioners can change their models to improve performance, and the degree of speedup that can be expected.

### 4.3.1 Switch performance

**Use symmetry breaking to reliably reduce optimization time.** Compared to the baseline, only one switch had a large runtime impact for both solvers when used in isolation. The overall results are shown in Fig. 4. Deleting the symmetry in the model by fixing one variable makes finding a solution approximately twice as fast (in general), with even more speedup for larger graphs. For both transitivity formulations, every combination of switches that included fixing one variable greatly outperformed the combinations that did not (Fig. 6). It is incredibly helpful as a universal technique and shows that even advanced ILP solvers do not always have the capability to recognize the inherent symmetry of the problem. Adopting this approach may improve runtime performance for other problems in visualization and beyond. We recommend that anyone implementing an ILP/MIP model examine the problem for symmetries that allow you to fix even a single variable.

**Combining switches is very successful, but only when using HiGHS.** With the open-source solver HiGHS [26], all three of symmetry breaking, mirrored variables, and continuous variables contributed significant performance improvements when using direct transitivity—see Fig. 5. Moreover, the formulation using these three switches was the best-performing combination of the experiment, solving an impressive 88% of the 600 graphs tested within one minute. The baseline, meanwhile, solved only 60%. This corresponds to a more than **17× median speedup** across all graphs tested—the combination had a median runtime of 1.9 seconds while the baseline took a median of 32.5 seconds to complete.

A number of other switches also contributed to strong performance results when using HiGHS with direct transitivity, namely butterfly reduction, cycle constraints, and leaf node collapse. Generally speaking, combining switches together greatly improved the performance of the HiGHS solver.

This contrasts with the Gurobi results, for which symmetry breaking drastically improved runtime while the inclusion of additional switches often slowed the solver down. For instance, using symmetry breaking with no other switches was a top-performing combination. Solving the 1150 graphs took a median runtime of 1.7 seconds while the baseline took 4.2 seconds—a more modest 2.5× speedup. We suspect that since Gurobi is a very efficient solver optimized for finding high-quality potential solutions, the addition of some of our described techniques is more of a distraction. That is, the solver wastes time each iteration verifying the additional constraints added by many of the techniques we describe are upheld when that time would often be better spent applying solution-finding and pruning heuristics to advance towards the optimal solution.

**The remaining switches give little performance increase.** By comparison, the remaining six switches do not have much, if any, positive impact when used in isolation, suggesting their usefulness is more restricted to specific graphs, or larger input sizes and cutoff times. Results for all individual switches are included in the supplemental materials at `https://osf.io/5vq79`.

**However, switch performance dependends on more than node count.** We were concerned with the variability exhibited in the ILP solver times and performed additional studies on layer counts and edge density with a more controlled dataset. Procedures and results are described in the appendices at `https://osf.io/5vq79`. We find that when using Gurobi with direct transitivity, both symmetry breaking and *x*-var branch priority delay the exponential spike in runtime that happens when increasing the edge density and number of layers—for the 50-node graphs studied, symmetry breaking in particular was able to solve graphs at up to 35% edge density within 5 minutes, while the baseline was unable to solve the graphs with 25% edge density.

| Gurobi | | | HiGHS | | |
|---|---|---|---|---|---|
| Switch | Direct | Vertical | Switch | Direct | Vertical |
| 🔴 | **13.9%** | **15.6%** | 🔴 | **4.9%** | **4.7%** |
| 🟣 | 0.61% | 0.0% | 🟣 | **1.6%** | 0.30% |
| 🟤 | -2.9% | -5.9% | 🟤 | -0.35% | 0.22% |
| 🟠 | 0.22% | -0.09% | 🟠 | **2.2%** | 0.91% |
| 🩷 | -0.22% | 0.78% | 🩷 | 0.78% | **2.3%** |
| 🔵 | 0.61% | 0.57% | 🔵 | **1.5%** | 0.74% |
| 🟢 | **1.8%** | 0.09% | 🟢 | — | — |
| 🟡 | 0.65% | 0.13% | 🟡 | — | — |
| 🩵 | -2.1% | -2.5% | 🩵 | **1.3%** | **1.9%** |

Fig. 5: Median *improvement* provided by each switch for direct and vertical transitivity formulations, provided as an increase in the number of graphs solved in the all-combinations experiment dataset (1,150 graphs for Gurobi and 600 for HiGHS). Average improvements over 1%—12 graphs for the Gurobi experiment and 6 for HiGHS—are bolded. We define improvement for a switch as the difference between the number of graphs successfully solved for a given formulation and the same formulation without that switch, presenting the median improvement over all formulations including the given switch. We see that including 🔴 symmetry breaking resulted in being able to solve an average of 160 (13.9%) additional graphs when using Gurobi and direct transitivity. This is a huge improvement, seeing as each 10-node interval included 50 graphs: including this switch allowed us to routinely solve graphs 30 nodes larger than without the switch, within the same amount of time.

**Guidelines for using switches.** We provide the following summary of the improvement provided by each switch technique.

| | |
|---|---|
| Substantially improves runtime | 🔴 |
| Improves performance, but is solver- or graph-dependent | 🟠 🩷<br>🔵 🟢<br>🩵 |
| May give small improvement | 🟣 🟡 |
| Not recommended | 🟤 |

For the best guarantee of fast optimal layouts, it is recommended to select a number of combinations of switches shown to perform well on average for your choice of solver and run them in parallel. This also helps offset some of the variability inherent to the ILP solving process. For an example of this, refer to Sec. 5.

### 4.3.2 Transitivity constraint performance

**Direct transitivity generally outperforms vertical position transitivity.** Each combination of switches with direct transitivity solved, on average, 21 more graphs (1.9% of the dataset) than the same combination with vertical position transitivity. It is worth noting that direct transitivity combinations took more setup time than vertical position transitivity combinations, which never took more than 1 second. This is due to the direct transitivity requiring more constraints than vertical position transitivity—for large inputs, this $O(|V|^3)$ versus $O(|V|^2)$ difference starts to become noticeable. Additionally, vertical position transitivity performs better than direct transitivity for specific graphs, oftentimes large graphs with many layers but fewer nodes per layer, as in Sec. 5. However, for the HiGHS solver all-combinations experiment, direct transitivity formulations solved 7.0% more graphs on average, a much more substantial increase than Gurobi. Direct transitivity formulations also responded better to the addition of switches, seen in Fig. 5. Therefore, we recommend using direct transitivity constraints, although experimentation with vertical position transitivity is beneficial for large many-layered graphs when using efficient solvers.

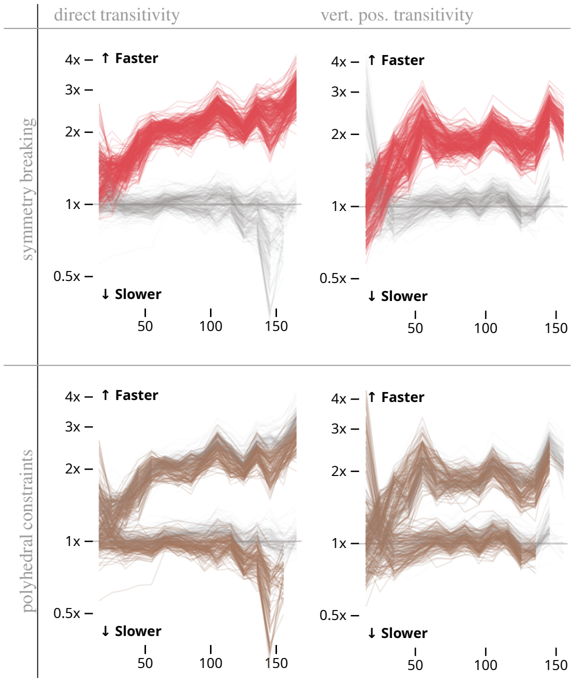


Fig. 6: All 512 combinations for each transitivity formulation plotted against their baseline using the natural log ratio of median runtimes. In the top row, all combinations using 🔴 symmetry breaking are highlighted in red, showing that the use of the switch completely partitions the space of all combinations beyond 50 total nodes. 🟤 Polyhedral constraints, meanwhile, tend to be on the bottom half of the two partitions performance-wise.

**Some types of redundancy are helpful, but others are not.** Lalla-Ruiz et al. claim that redundancy can both help and hinder the solver performance [32]. We can corroborate this since some of our techniques introduce redundancy into the model. Specifically, 🟠 mirrored variables doubles the variables and constraints in the model, but does not typically slow down solution time, even improving it substantially for some solvers and formulations. However, a study we conducted found that including both direct transitivity and vertical position transitivity simultaneously had a large negative impact on solution time—see appendices at `https://osf.io/5vq79`.

## 5 Case Study: Visualizing Software Control Flow

Control Flow Graph (CFG) readability is crucial to help reverse engineers [35, 52] and malware researchers [55] extract knowledge from decompiled binary files, including malware. Therefore minimizing edge crossing should be prioritized to facilitate tasks like tracking variables and tracing activation code in the diagram.

Reverse engineering control flow graphs can be represented as layered graphs, where nodes correspond to blocks of decompiled assembly code, and edges represent jumps taken between code blocks [35]. Code blocks are often ended by JMP or JNZ instructions, followed by an edge connecting to the next code block. Specific structures can appear in CFG visualizations which tell engineers about the nature of the code. For example, switch instructions generate a large number of outgoing edges from a single node. The layer assignment for these drawings is typically done by trying to maximize the number of call arrows pointing in the same direction before assigning positions within each layer, using a Sugiyama-style approach [6]. The layouts of these drawings are conventionally computed with heuristic-based algorithms (Radare2, for instance, cites Buchheim et al. [3]). The optimal approach we propose can be used to produce more readable visualizations for reverse engineers. For example, the readability and utility of control flow graphs generated by tools like the recent work by Devkota et al. [6] could be further improved by using layouts with the optimally fewest crossings. While optimal approaches have rarely been applied in this context in the past because of their high

| Command | $\lvert V'\rvert$ | $\lvert E'\rvert$ | Baseline | Baseline + 🟥 | SL [9] | OSD [56] | PA [29] | OkLCM [16] | **Ensemble** |
|---|---|---|---|---|---|---|---|---|---|
| `chmod` | 511 | 581 | 8.95 s | 1.59 s | 48.7 s | 6.03 s | 6.21 s | 1.61 s | **1.43 s** |
| `echo` | 279 | 333 | 8.83 s | 3.23 s | 3.19 s | 9.71 s | 8.88 s | 2.14 s | **1.22 s** |
| `cp` | 359 | 423 | 9.42 s | 13.7 s | 6.01 s | 12.2 s | 13.5 s | 13.3 s | **2.88 s** |

Table 4: Control Flow Graph optimization results, reported as the median of five trials on each graph. All formulations were tested using Gurobi. The formulations for comparison are, in order, a baseline with direct transitivity and no switches, the same baseline with 🟥 symmetry breaking, Di Bartolomeo et al.'s Stratisfimal Layout (SL) [9], Zarate et al.'s Optimal Sankey Diagrams (OSD) [56], Jünger et al.'s Polyhedral Approach to Crossing Minimization (PA) [29], and Gange et al.'s Optimal $k$-Level Planarization and Crossing Minimization (OkLCM) [16]. Our ensemble method, taking advantage of running multiple formulations in parallel, is quickest for all three graphs.

computational requirements, we demonstrate that our approach is viable for use with control flow graphs with very large numbers of nodes.

We compare an ensemble method of our best switch combinations to formulations from the literature, which we can also represent as switch combinations. Di Bartolomeo et al.'s Stratisfimal Layout [9] uses vertical position transitivity with no other switches. Zarate et al.'s Optimal Sankey Diagrams [56] use direct transitivity with 🟧 mirrored variables, 🟪 butterfly reduction, and 🟩 $x$-var branch priority. Jünger et al.'s Polyhedral Approach [29] uses direct transitivity with 🟫 polyhedral constraints. Gange et al.'s Optimal $k$-Level Crossing Minimization [16] uses direct transitivity with 🟥 symmetry breaking, 🩷 cycle constraints, and 🟦 leaf node collapse.

Our ensemble method ran six combinations in parallel, taking the layout from whichever finished first. The table below lists these combinations by their transitivity type (direct transitivity or vertical transitivity), and the switches combined. These combinations were selected because they performed strongly in the all-combinations experiment.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1) Direct | & | 🟥 | | 🟧 | | | 🟩 | |
| 2) Direct | & | 🟥 | | | | | | |
| 3) Direct | & | 🟥 | | | 🩷 | 🟦 | | |
| 4) Direct | & | 🟥 | 🟪 | | 🩷 | | 🟩 | 🟨 |
| 5) Vertical | & | 🟥 | 🟪 | 🟧 | | | 🟩 | |
| 6) Vertical | & | 🟥 | | 🟧 | 🩷 | 🟦 | 🟩 | |

The case study was run on a desktop computer with an Intel Core i7-8700K CPU with 6 cores, 32 GB RAM, Windows 10, & Gurobi 10. The number of processing threads available to the solver was limited to 1 for all ensemble members, and 6 for all other methods. Our ensemble method found an optimal control flow diagram for the implementation of the Linux `chmod` command in an average of 2.34 seconds over five repeated trials, see Tab. 4. Part of this optimal layout is pictured in Fig. 1. This control flow graph has 511 nodes and 581 edges after layering and inserting dummy nodes and edges—all control flow graphs included had a large number of layers but reasonable layer widths, allowing for quick solving even for the relatively huge graph size.

Our recommended method never took longer than 1.5 seconds to find the optimal drawing, which has 16 crossings. The advantage of the ensemble method is that it leverages our finding in Sec. 4.3 that different formulations perform better on different input graphs. For `chmod`, the fastest-performing ensemble member was model 2, which used only 🟥 symmetry breaking. More complicated models worked much better than this for `echo` and `cp`, on the other hand, which were solved fastest by models 5 and 4 respectively. Each member solved at least one of the graphs very quickly, but speed often varied by up to 5× slower than the fastest member, particularly for `cp`. As for the other models evaluated, the access to 6× more processing power did not seem to improve performance much if at all, likely due to the relatively quick execution times and the solver not needing to visit many branch-and-bound nodes [21]. `chmod` and `echo` were solved quickly by OkLCM [16], but `cp` was solved the quickest by the vertical transitivity-based formulations such as SL [9].

## 6 Limitations

Like all methods that use ILP, the key limitation of our approach is the upper bound on the size of graphs which can be solved in a reasonable amount of time. In our experiments, graphs with up to 100 nodes in the original graph could typically be solved quickly with our techniques. Graphs with more than 300 nodes could not often be solved within a 5-minute cutoff time for the edge densities studied in our experiments.

It should be noted, however, that the number of edges and nodes is not fully indicative of the complexity of a problem. For clarity of presentation, we have categorized graphs throughout this paper by their total nodes. Although it is more consistent than reporting based on the number of nodes in the original graph, it is still far from perfect as a predictor of runtime. We have studied some additional factors (see appendices at `https://osf.io/5vq79`), but more work is needed. There is also little indication in our results of what causes certain techniques to perform much better on some graphs than others, an example being the differences in performance between direct transitivity and vertical position transitivity for the control flow graphs studied in Sec. 5.

Furthermore, it is impossible to perfectly predict the runtime of the ILP solver given only knowledge of the input graph because ILP solvers contain internal randomization such as for selecting branching variables. This randomization causes some variability in optimization time with repeated trials [15, 34]. Sometimes, the solver gets lucky and finds a solution very quickly, when the solver would take much longer on average. Conversely, the solver can get unlucky and take much longer than average. We overcome this variance in the experiments by using a large number of graphs, and in practice by recommending an ensemble method.

## 7 Conclusion and Future Work

Algorithms that arrange node-link graph visualizations with consideration of human factors can assist users in comprehending complex graphs more quickly. With the aim of scaling optimal layouts to larger graphs, we present in this paper an adaptable framework that yields empirically faster layout optimization, and analyzed nine established and innovative techniques using two different solvers. The result is a comprehensive benchmark characterizing the degree to which the nine techniques, two types of transitivity formulation, and two ILP solvers impact the running time of ILP solutions for layered crossing minimization. Our recommended approach can quickly generate optimal layouts for most graphs with up to 150 nodes and works in a reasonable amount of time for many graphs with even more nodes. Our implementation is available as open-source code, as are our benchmark results and datasets used for our empirical study.

**Future work.** More research can be done to evaluate techniques for the scalability of aesthetic criteria besides crossing minimization, such as planarity, edge length, edge bundling, and node groupings. The experiments described in this paper can be replicated using additional ILP solvers, ensemble methods such as the one in Sec. 5, and larger graphs over longer runtimes, to better recommend software to use for the problem and further investigate the trends observed. Additionally, more experimentation is in order to better understand the relationship between the input graph and the solver runtime. Work could be done to integrate our fast crossing minimization algorithm with Devkota et al.'s *CFGConf* for even more understandable, useful control flow graph generation.

## Supplemental Materials

All supplemental materials can be found on OSF at `https://osf.io/5vq79`. We provide:

1. A copy of our paper, including all appendices and supplemental figures.
2. Open-source GitHub repository of our implementation in Python, including a `LayeredOptimizer` class which takes as input a layered graph, and allows the user to choose what switches to use, then generates the optimal drawing.
3. All result data and datasets used in our experiments, as well as code used to sample said datasets.
4. All results figures:
   (a) 21 individual switch evaluation figures, one for each switch on each transitivity formulation.
   (b) 24 total figures which extend Fig. 6 to all switches.
   (c) Full optimal layouts for the three control flow graphs in the case study (Sec. 5).

## References

[1] S. D. Bartolomeo, Y. Zhang, F. Sheng, and C. Dunne. Sequence Braiding: Visual Overviews of Temporal Event Sequences and Attributes. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):1353–1363, 2021. doi: 10.1109/TVCG.2020.3030442 2, 6

[2] J. Blythe, C. McGrath, and D. Krackhardt. The effect of graph layout on inference from social network data. In F. J. Brandenburg, ed., *Graph Drawing*, pp. 40–51. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996. doi: 10.1007/BFb0021789 1

[3] C. Buchheim, M. Jünger, and S. Leipert. A fast layout algorithm for k-level graphs. In J. Marks, ed., *Graph Drawing*, pp. 229–240. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. doi: 10.1007/3-540-44541-2_22 8

[4] M. Chimani, C. Gutwenger, M. Jünger, G. W. Klau, K. Klein, and P. Mutzel. The Open Graph Drawing Framework (OGDF). In R. Tamassia, ed., *Handbook of Graph Drawing and Visualization*, chap. 17. CRC Press, 2014. 1

[5] T. N. Dang, N. Pendar, and A. G. Forbes. Timearcs: Visualizing fluctuations in dynamic networks. *Computer Graphics Forum*, 35(3):61–69, 2016. doi: 10.1111/cgf.12882 1

[6] S. Devkota, M. P. LeGendre, A. Kunen, P. Aschwanden, and K. E. Isaacs. Domain-Centered Support for Layout, Tasks, and Specification for Control Flow Graph Visualization. In *2022 Working Conference on Software Visualization (VISSOFT)*, pp. 40–50, 2022. doi: 10.1109/VISSOFT55257.2022.00013 8

[7] S. Di Bartolomeo, T. Crnovrsanin, D. Saffo, E. Puerta, C. Wilson, and C. Dunne. Evaluating graph layout algorithms: A systematic review of methods and best practices. *Computer Graphics Forum*, n/a(n/a):e15073, 2024. doi: 10.1111/cgf.15073 2

[8] S. Di Bartolomeo, E. Puerta, C. Wilson, T. Crnovrsanin, and C. Dunne. A collection of benchmark datasets for evaluating graph layout algorithms. Graph Drawing Posters, 2023. preprint doi: `https://doi.org/10.31219/osf.io/yftju`. 6

[9] S. Di Bartolomeo, M. Riedewald, W. Gatterbauer, and C. Dunne. STRATISFIMAL LAYOUT: A modular optimization model for laying out layered node-link network visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):324–334, 2021. doi: 10.1109/TVCG.2021.3114756 1, 2, 3, 4, 9

[10] G. Di Battista, A. Garg, G. Liotta, A. Parise, R. Tamassia, E. Tassinari, F. Vargiu, and L. Vismara. Drawing directed acyclic graphs: An experimental study. *International Journal of Computational Geometry & Applications*, 10(06):623–648, 2000. doi: 10.1142/S0218195900000358 6

[11] G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An experimental comparison of four graph drawing algorithms. *Computational Geometry*, 7(5-6):303–325, 1997. doi: 10.1016/S0925-7721(96)00005-3 6

[12] P. Eades, X. Lin, and W. Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6):319–323, 1993. doi: 10.1016/0020-0190(93)90079-O 6

[13] P. Eades, X. Lin, and R. Tamassia. An algorithm for drawing a hierarchical graph. *International Journal of Computational Geometry & Applications*, 06(02):145–155, 1996. doi: 10.1142/S0218195996000101 2, 5

[14] P. Eades and N. C. Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11(4):379–403, 1994. doi: 10.1007/BF01187020 2, 6

[15] M. Fischetti and M. Monaci. Exploiting erraticism in search. *Operations Research*, 62(1):114–122, 2014. 9

[16] G. Gange, P. J. Stuckey, and K. Marriott. Optimal k-Level Planarization and Crossing Minimization. In U. Brandes and S. Cornelsen, eds., *Graph Drawing*, vol. 6502 of *Lecture Notes in Computer Science*, pp. 238–249. Springer Berlin Heidelberg, 2011. doi: 10.1007/978-3-642-18469-7_22 2, 3, 4, 5, 9

[17] E. Gansner, E. Koutsofios, S. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993. doi: 10.1109/32.221135 1, 2

[18] M. R. Garey and D. S. Johnson. Crossing Number is NP-Complete. *SIAM Journal on Algebraic Discrete Methods*, 2006. doi: 10.1137/0604033 2

[19] H. Gibson, J. Faith, and P. Vickers. A survey of two-dimensional graph layout techniques for information visualisation. *Information Visualization*, 12(3-4):324–357, 2013. doi: 10.1177/1473871612455749 2

[20] M. Gronemann, M. Jünger, F. Liers, and F. Mambelli. Crossing Minimization in Storyline Visualization. In Y. Hu and M. Nöllenburg, eds., *Graph Drawing and Network Visualization*, Lecture Notes in Computer Science, pp. 367–381. Springer International Publishing, 2016. doi: 10.1007/978-3-319-50106-2_29 1, 6

[21] Gurobi Optimization LLC. Gurobi Optimizer Reference Manual, 2023. 2, 4, 5, 7, 9

[22] P. Healy and A. Kuusik. The Vertex-Exchange Graph: A New Concept for Multi-level Crossing Minimisation. In J. Kratochvíyl, ed., *Graph Drawing*, vol. 1731 of *Lecture Notes in Computer Science*, pp. 205–216. Springer Berlin Heidelberg, 1999. doi: 10.1007/3-540-46648-7_21 3, 5

[23] P. Healy and N. Nikolov. *Hierarchical Drawing Algorithms*, pp. 409–454. Handbook on Graph Drawing and Visualization, 08 2013. 2

[24] I. Herman, G. Melancon, and M. Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000. doi: 10.1109/2945.841119 2

[25] W. Huang and M. Huang. Exploring the relative importance of crossing number and crossing angle. In *Proceedings of the 3rd International Symposium on Visual Information Communication*, VINCI '10, article no. 10, 8 pages. Association for Computing Machinery, New York, NY, USA, 2010. doi: 10.1145/1865841.1865854 1

[26] Q. Huangfu and J. A. J. Hall. Parallelizing the dual revised simplex method. *Mathematical Programming Computation*, 10(1):119–142, 2018. doi: 10.1007/s12532-017-0130-5 2, 7

[27] igraph Development Team. Visualisation of graphs - igraph stable documentation, python. 2

[28] M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, eds. *50 Years of Integer Programming 1958-2008*. Springer Berlin Heidelberg, Nov. 2009. doi: 10.1007/978-3-540-68279-0 2

[29] M. Jünger, E. K. Lee, P. Mutzel, and T. Odenthal. A polyhedral approach to the multi-layer crossing minimization problem. In G. DiBattista, ed., *Graph Drawing*, vol. 1353 of *Lecture Notes in Computer Science*, pp. 13–24. Springer Berlin Heidelberg, 1997. doi: 10.1007/3-540-63938-1_46 3, 4, 5, 9

[30] M. Jünger and P. Mutzel. Exact and heuristic algorithms for 2-layer straightline crossing minimization. In F. J. Brandenburg, ed., *Graph Drawing*, vol. 1027 of *Lecture Notes in Computer Science*, pp. 337–348. Springer Berlin Heidelberg, 1996. doi: 10.1007/BFb0021817 2

[31] M. Jünger and P. Mutzel. 2-Layer Straightline Crossing Minimization: Performance of Exact and Heuristic Algorithms. *Journal of Graph Algorithms and Applications*, 1, 1997. doi: 10.1142/9789812777638_0001 1, 2

[32] E. Lalla-Ruiz and S. Voß. Improving solver performance through redundancy. *Journal of Systems Science and Systems Engineering*, 25(3):303–325, 2016. doi: 10.1007/s11518-016-5301-9 3, 5, 8

[33] M. Liu, J. Shi, Z. Li, C. Li, J. Zhu, and S. Liu. Towards better analysis of deep convolutional neural networks. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):91–100, 2017. doi: 10.1109/TVCG.2016.2598831 1

[34] A. Lodi and A. Tramontani. *Performance Variability in Mixed-Integer Programming*, chap. Chapter 1, pp. 1–12. Theory Driven by Influential Applications, 2014. doi: 10.1287/educ.2013.0112 9

[35] A. Mantovani, S. Aonzo, Y. Fratantonio, and D. Balzarotti. RE-Mind: a first look inside the mind of a reverse engineer. In *31st USENIX Security Symposium (USENIX Security 22)*, pp. 2727–2745. USENIX Association, Boston, MA, Aug. 2022. 8

[36] C. Matuszewski, R. Schönfeld, and P. Molitor. Using sifting for k-layer straightline crossing minimization. In J. Kratochvíyl, ed., *Graph Drawing*, pp. 217–224. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. doi: 10.1007/3-540-46648-7_22 2

[37] T. Milea, O. Schrijvers, K. Buchin, and H. Haverkort. Shortest-Paths Preserving Metro Maps. In M. van Kreveld and B. Speckmann, eds., *Graph Drawing*, vol. 7034 of *Lecture Notes in Computer Science*, pp. 445–446. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-25878-7_45 2

[38] D. R. Morrison, S. H. Jacobson, J. J. Sauppe, and E. C. Sewell. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization*, 19:79–102, 2016. doi: 10.1016/j.disopt.2016.01.005 3

[39] P. Mutzel. An alternative method to crossing minimization on hierarchical graphs. In S. North, ed., *Graph Drawing*, pp. 318–333. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. 3, 4, 5

[40] P. Mutzel and R. Weiskircher. Two-Layer Planarization in Graph Drawing. In K.-Y. Chwa and O. H. Ibarra, eds., *Algorithms and Computation*, vol. 1533 of *Lecture Notes in Computer Science*, pp. 72–79. Springer Berlin Heidelberg, 1998. doi: 10.1007/3-540-49381-6_9 3

[41] M. Nollenburg and A. Wolff. Drawing and Labeling High-Quality Metro Maps by Mixed-Integer Programming. *IEEE Transactions on Visualization and Computer Graphics*, 17(5):626–641, 2011. doi: 10.1109/TVCG.2010.81 2

[42] H. Purchase. Which aesthetic has the greatest effect on human understanding? In G. DiBattista, ed., *Graph Drawing*, Lecture Notes in Computer Science, pp. 248–261. Springer, 1997. doi: 10.1007/3-540-63938-1_67 1, 2

[43] H. C. Purchase. Metrics for Graph Drawing Aesthetics. *Journal of Visual Languages & Computing*, 13(5):501–516, 2002. doi: 10.1006/jvlc.2002.0232 2

[44] S.-V. Sanei-Mehri, A. E. Sariyuce, and S. Tirthapura. Butterfly Counting in Bipartite Networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2150–2159. ACM, 2018. doi: 10.1145/3219819.3220097 4

[45] K. Sugiyama. *Graph Drawing and Applications for Software and Knowledge Engineers*. WORLD SCIENTIFIC, 2002. doi: 10.1142/4902 2

[46] K. Sugiyama, S. Tagawa, and M. Toda. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981. doi: 10.1109/TSMC.1981.4308636 1, 2

[47] R. Tamassia, ed. *Handbook of Graph Drawing and Visualization*. CRC Press, 2007. 2

[48] A. Tarassov, N. S. Nikolov, and J. Branke. A Heuristic for Minimum-Width Graph Layering with Consideration of Dummy Nodes. In C. C. Ribeiro and S. L. Martins, eds., *Experimental and Efficient Algorithms*, Lecture Notes in Computer Science, pp. 570–583. Springer, 2004. doi: 10.1007/978-3-540-24838-5_42 6

[49] L. Tornqvist, P. Vartia, and Y. O. Vartia. How Should Relative Changes Be Measured? *The American Statistician*, 39(1):43–46, 1985. doi: 10.2307/2683905 7

[50] V. Valls, R. Marti, and P. Lino. A branch and bound algorithm for minimizing the number of crossing arcs in bipartite graphs. *European Journal of Operational Research*, 90(2):303–319, April 1996. doi: 10.1016/0377-2217(95)00356-8 2

[51] J. P. Vielma. Mixed integer linear programming formulation techniques. *SIAM Review*, 57(1):3–57, 2015. doi: 10.1137/130915303 6

[52] D. Votipka, S. Rabin, K. Micinski, J. S. Foster, and M. L. Mazurek. An observational investigation of reverse Engineers' processes. In *29th USENIX Security Symposium (USENIX Security 20)*, pp. 1875–1892. USENIX Association, Aug. 2020. 8

[53] C. Ware, H. Purchase, L. Colpoys, and M. McGill. Cognitive Measurements of Graph Aesthetics. *Information Visualization*, 1(2):103–110, 2002. doi: 10.1057/palgrave.ivs.9500013 2

[54] K. Wongsuphasawat, D. Smilkov, J. Wexler, J. Wilson, D. Mané, D. Fritz, D. Krishnan, F. B. Viégas, and M. Wattenberg. Visualizing dataflow graphs of deep learning models in tensorflow. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):1–12, 2018. doi: 10.1109/TVCG.2017.2744878 1

[55] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith. Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study. In *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 158–177. IEEE, 2016. doi: 10.1109/SP.2016.18 8

[56] D. C. Zarate, P. L. Bodic, T. Dwyer, G. Gange, and P. Stuckey. Optimal Sankey Diagrams Via Integer Programming. In *2018 IEEE Pacific Visualization Symposium (PacificVis)*, pp. 135–139, 2018. doi: 10.1109/PacificVis.2018.00025 3, 4, 5, 9

## A Experiments on Graph Density and Layer Count

We noticed a significant amount of variance in ILP solve time when graphed as a function of the number of nodes, so we performed some additional experimentation to try and explain this. We examined how the number of layers and the edge density affect solve time. Following the experiment design of Gange et al. [16] and other benchmark studies on layered graphs [31, 36], we controlled for other variables by generating random graphs. To generate one graph, a density $d$ and layer count $k$ were selected, and then $k$ layers of 10 nodes per layer were created. For each consecutive pair of layers, edges were randomly and uniformly sampled until the number of edges was equal to the required density $d$. If the sampling process ever reached a point where the next sample could ensure that a node remained unconnected, the sample space was restricted to unconnected nodes, thus ensuring connected graphs. To generate a set of graphs where only density was varied, we sampled 10 random graphs for each $d = 14, 16, \ldots, 48, 50$ fixing $k = 5$ and 10 nodes per layer. To generate a set of graphs where only the number of layers varied, we sampled 10 random graphs for each $k = 3, 4, \ldots, 19, 20$, fixing $d = 0.15$ and 10 nodes per layer.

We find that runtime is strongly exponential in the edge density of the graph, but our techniques, particularly symmetry breaking and *x*-var branch priority do much to delay this exponential explosion in runtime—see Fig. 7. The same holds true for increasing the number of layers $k$.

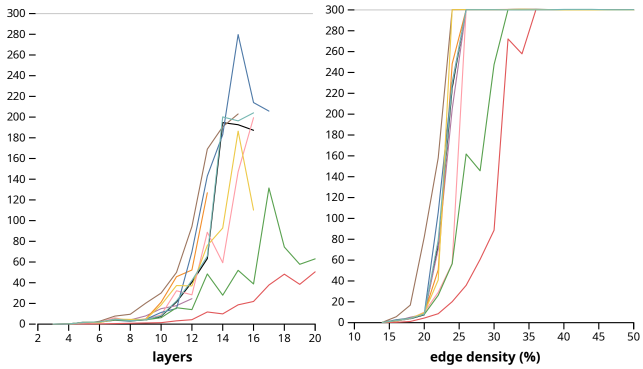


Fig. 7: ILP runtime (s) for each switch when using Gurobi, as the median of the 10 graphs at each layer count/edge density. Individual graph runtime was cut off at 300 seconds, and switches were cut off once fewer than 50% of the graphs completed. Symmetry breaking and *x*-var branch priority delay the spike in runtime, allowing us to solve larger and more dense graphs. These techniques are plotted in red and green respectively, and the baseline is plotted in black.

## B On Using Both Direct and Vertical Transitivity

In previous runs of the individual and all-combinations experiments described in Sec. 4.2, an additional transitivity formulation was included, which was to add both the direct transitivity constraints and the vertical position transitivity constraints simultaneously to the model. The reasoning was that the literature suggests that redundancy can often assist solution finding, such as the mirrored variables technique of Zarate et al. [56] or the findings of Lalla-Ruiz et al. [32]. However, this technique was found to greatly hinder performance, often by 2–3× for the graph sizes tested. Figure 8 details the all-combinations results from this experiment, which found that the redundant transitivity formulation was consistently cut off 30 nodes sooner than the other two types of transitivity.

## C Empirically Testing Healy and Kuusik's Cycle Constraints

While Gange et al. [16] use Healy and Kuusik's vertex exchange graph [22] to recommend several constraints, they use only a small subset of the constraints described by Healy and Kuusik. To ensure a holistic evaluation of layered crossing minimization literature, we studied the performance of both the original set of constraints and the Gange et al. recommendation. We found that, as expected, the Gange et al. recommendation had superior performance, and is therefore the set of constraints used by our cycle constraints switch (Fig. 9). The original vertex exchange constraints, meanwhile, had a negative impact on runtime which grew worse the larger the input size.

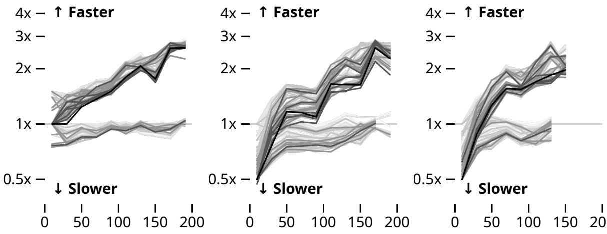


Fig. 8: All-combinations results with additional transitivity type, both direct and vertical position transitivity combined. Pictured left-to-right is direct transitivity, vertical position transitivity, and both combined, with the lines darkened according to the number of switches included in the combination. The both-combined formulations consistently cut off 30 nodes sooner than the other two types, illustrating its tendency to negatively impact performance.

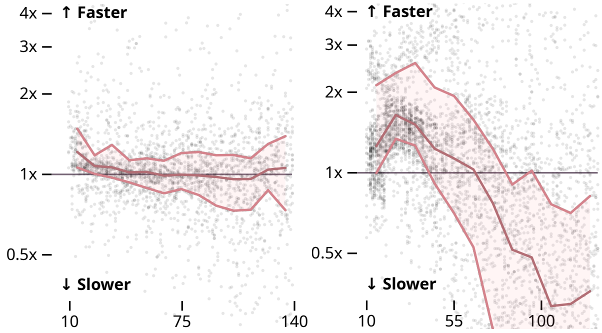
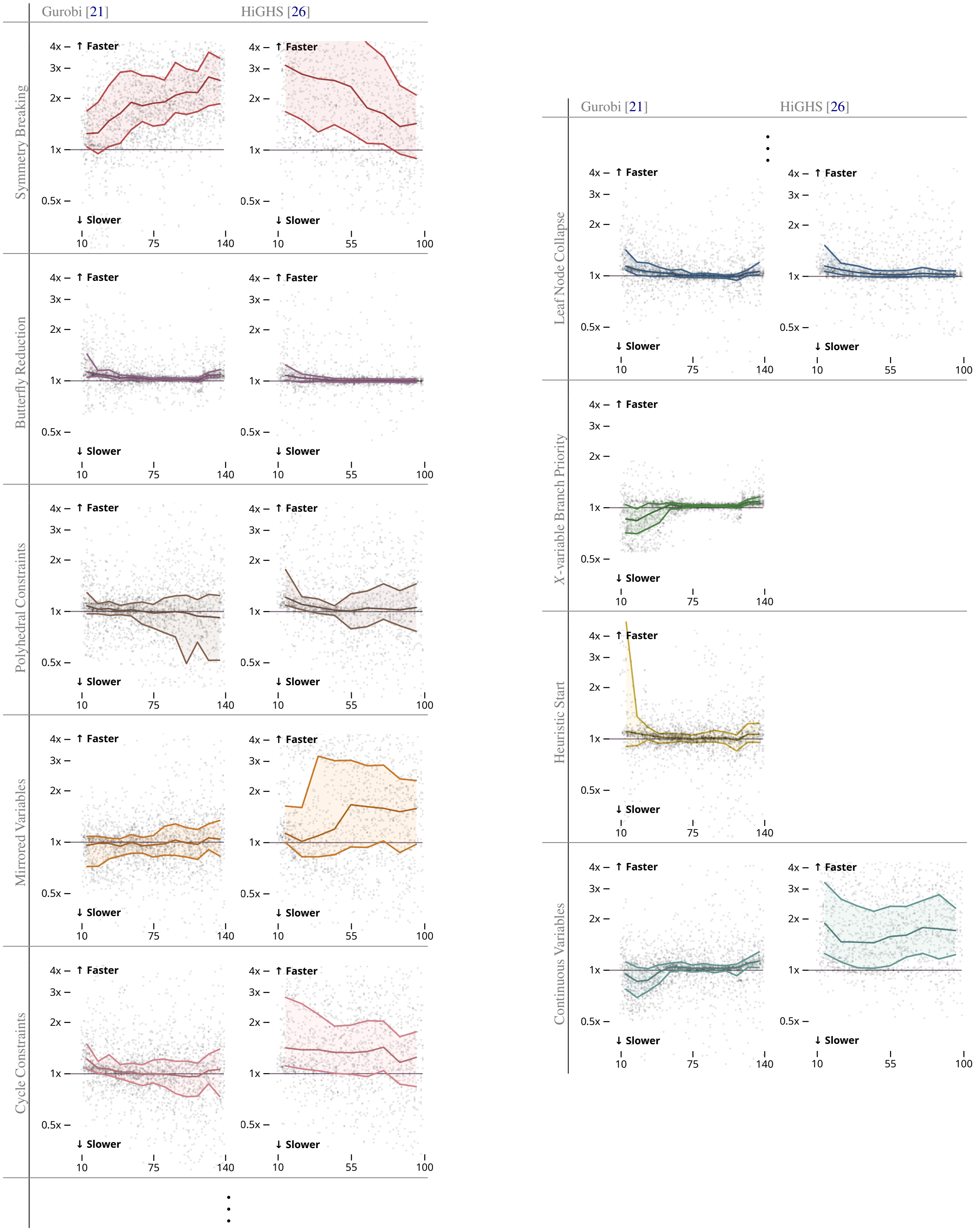


Fig. 9: Natural log ratio of runtime over the baseline, for the abbreviated cycle constraints on the left and the original vertex exchange cycle constraints on the right. While the left often improves runtime, the original constraints greatly decrease performance for graphs larger than 50 total nodes.

## D All Individual Switch Figures

We provide the figures for the individual switch evaluation (Sec. 4.2.2), for all nine switches across both solvers, with results for both transitivity types combined.

Table 5: All individual switch results, both types of transitivity combined. Results are reported using natural log ratio relative to baseline performance.


Gurobi [21]
HiGHS [26]
Symmetry Breaking
Butterfly Reduction
Polyhedral Constraints
Mirrored Variables
Cycle Constraints
Leaf Node Collapse
X-variable Branch Priority
Heuristic Start
Continuous Variables
4x
3x
2x
1x
0.5x
↑ Faster
↓ Slower