PACE 2024 HWoydt: CrossingGuard

Henning Martin Woydt
Heidelberg University, Germany

Abstract

We developed the exact solver `CrossingGuard` to solve the one-sided crossing minimization(OCM) problem. Given a bipartite graph and a fixed ordering on one of the vertex sets, determine the ordering of the other vertex set, such that a layered graph drawing has a minimum number of edge crossings. The problem is known to be NP-hard. Our algorithm operates in two phases: first, it applies exact reduction rules to reduce the graph size and afterward it exhaustively searches the permutation tree via a branch-and-cut search algorithm.

2012 ACM Subject Classification Theory of computation → Parameterized-complexity-and-exact-algorithms; Theory of computation → Branch-and-bound

Keywords and phrases Cross Minimization, Branch-and-Bound, Exact Algorithm

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Supplementary Material Source Code (Zenodod): `https://zenodo.org/doi/10.5281/zenodo.11535251`
Source Code (GitHub): `https://github.com/HenningWoydt/PACE2024Exact`

Funding *Henning Martin Woydt*: Supported by the DFG, grant SCHU 2567/6-1.

1 Introduction

The PACE 2024 Challenge asks to solve the one-sided crossing minimization problem (OCM). The problem stems from layered graph drawing and is defined as follows: Given a bipartite graph $G = ((A \cup B), E)$ and a fixed ordering π_A on A, find a fixed ordering π_B on B that minimizes the number of edge crossings in a straight-line drawing of G. While the PACE 2024 Challenge includes an Exact, Approximate and Parameterized Track, this work focuses solely on the exact case. Solving the OCM problem exactly is known to be NP-hard [2].

Section 2 provides necessary definitions, and Section 3 details our exact algorithm.

2 Preliminaries

A straight-line drawing of a bipartite graph consists of two parallel lines: one line contains all vertices of A placed in their given order, and the other line contains all vertices of B placed in their respective order. In such a drawing, two edges $\{a, b\}, \{c, d\} \in E$ with $a \prec_{\pi_A} c$ will cross each other if $d \prec_{\pi_B} b$. For $u, v \in B$ we define $c(u, v)$ as the number of crossings if $u \prec_{\pi_B} v$ (u is placed left of v). The neighborhood of a vertex v is defined as $N[v] = \{u \mid \{v, u\} \in E\}$.

3 Exact Algorithm

Our exact algorithm consists of two phases, first the reduction phase and afterward the branch-and-cut phase. The reduction phase applies exact reduction rules that decrease the graph size, most importantly the number of vertices in B. The branch-and-cut phase then exhaustively searches the permutation tree of the remaining vertices in B, ensuring that the optimal ordering is found.

3.1 Reduction Phase

The goal of reduction rules is to reduce the size of the graph, especially the size of the vertices in B. Additionally, we try to split the graph into components that can be solved independently.

▶ **Lemma 1** (Independent Components). *Let $(G_1, G_2, \ldots, G_k)$ be a split of G into independent components such that all vertices of A_i are smaller than all vertices of A_j for $i < j$. By solving each component individually and concatenating the solutions based on the component ordering, we obtain a solution for G.*

This rule has proven extremely useful, as splitting B into multiple parts significantly benefits the branch-and-cut phase.

▶ **Lemma 2** (Almost Independent Components). *Let $b \in B$ and $N[b] = \{a_i, a_{i+1}, \ldots, a_{i+k}\}$ with vertices $a_{i+1}, \ldots, a_{i+k-1}$ only being connected to b. Additionally, let $G \setminus \{b\}$ split into two independent components with component G_1 including a_i and all vertices of A smaller a_i and component G_2 including a_{i+k} and all other vertices of A greater than a_{i+k}. Solving both components independently, then concatenating the solutions and placing b in the middle, will result in a solution for G.*

While we hoped that this rule would allow for further splitting of B, its effect was only marginal.

▶ **Lemma 3** (Front-Back Reduction). *Let $b \in B$ and $N[b] = \{a_1\}$ or $N[b] = \{a_1, a_2\}$. In such cases, removing b from the graph, solving the reduced graph, and then placing b at the front of the solution will yield a solution for the original graph. This strategy can also be applied to vertices that connect to the last two vertices in A.*

When removing and inserting multiple vertices, you have to place the vertices with exactly one neighbor n the outside and those with two neighbors on the inside. Unfortunately, the number of reduced vertices was not very large.

▶ **Lemma 4** (Twin Reduction). *Let $b_1, b_2 \in B$ with $N[b_1] = N[b_2]$. Then removing b_1 from G, solving the reduced graph, and placing b_1 next to b_2, will result in a solution for G.*

Twin reduction can also be applied to multiple vertices with the same neighborhood by contracting them all into one vertex. However, you have to keep track that the edges of the remaining vertex lead to more than just one crossing. We additionally keep track of edge weights that increase each time we contract two vertices. This reduction technique was highly beneficial for the challenge, although it appeared that some instances were specifically designed to exploit this reduction.

3.2 Branch-And-Cut Phase

In the branch-and-cut phase, the algorithm searches through a permutation tree. Each node $T = (F, R)$ in the tree consists of two sets F and R: F holds all the fixed vertices, they are already placed and cannot change their ordering (F functions more like a stack, then a set), while R contains the remaining vertices that can still be added to F. The children of a node T are all nodes $\mathrm{CH}(T) = \{(F \cup \{c\}, R \setminus \{c\}) \mid c \in R\}$. The root node of the permutation tree is $T_R = (\emptyset, B)$. The permutation tree therefore holds all permutations of B in its leaf nodes.

The permutation tree holds an exponential number of nodes, and therefore cannot be efficiently searched. To avoid branching into all nodes, we measure the minimum number of remaining crossings γ. Should γ exceed the currently best found solution γ_{best}, we can prune that branch of the search tree. The minimum number of remaining crossings γ consists of three parts:

$$\gamma_{\text{curr}} = \sum_{u,v \in F, u \prec_{\pi_B} v} c(u,v) \qquad \text{\#Crossings in } F$$

$$\gamma_{\text{inter}} = \sum_{u \in F, v \in R} c(u,v) \qquad \text{\#Crossings between } F \text{ and } R$$

$$\gamma_{\text{intra}} = \sum_{u,v \in R} \min(c(u,v), c(u,v)) \qquad \text{\#Min Crossings in } R$$

For each vertex pair in $B = F \cup R$ we either added the exact number of crossings of the pair (γ_{curr}, γ_{inter}) or we underestimated the correct number of crossings (γ_{intra}). The sum therefore is a valid lower bound for the correct number of crossings achievable in the subtree of T. Should γ be smaller than γ_{best}, the currently best-found solution, we have to explore the subtree as it can potentially hold a better solution. Otherwise, if $\gamma exceeds \gamma_{\text{best}}$, we can prune the search tree and continue exploring another subtree.

Determining γ naively takes $\mathcal{O}(n^4)$ time, as we need to calculate the crossing for $\mathcal{O}(n^2)$ pairs, with each calculation taking $\mathcal{O}(n^2)$ time. However, this can be drastically reduced.

First is the calculation of the crossings between all vertex pairs. Since the number of crossings only depends on the relative order of both vertices, they can be precomputed and later reused. This reduces the needed time at each node to $\mathcal{O}(n^2)$, however, it also increases the needed space to $\mathcal{O}(n^2)$.

The second improvement can be made by traversing the tree In-order. We will keep track of γ^b_{inter} for each vertex $b \in R$, and we will keep track of γ_{intra} at each depth of the search tree. Since only vertex c is moved between a child and its parent in the tree, it is much easier to compute the changes based on the move, rather than a total recomputation. We add γ^c_{inter} to γ_{curr}, we update γ^b_{inter} of all vertices $b \in R$ and since c is now part of F we remove c from the γ_{intra} calculation. In total, all operations take at most $\mathcal{O}(n)$ time, however this optimization also introduces an additional space requirement of $\mathcal{O}(n^2)$ as we save γ^b_{inter} at each depth of the tree for all $b \in B$.

The branch-and-cut search also makes use of patterns as described in [1].

▶ **Lemma 5** ($0/j$ Pattern). *Let $u, v \in B$ and let $c(u,v) = 0$ and $c(u,v) = j$. Then fix the ordering $u \prec_{\pi_B} v$.*

This is included when advancing from a parent to a child node. Let v be the vertex that will be added to F. Should $c(u,v) = 0$ for any $u \in R \setminus \{v\}$, then do not explore the child.

References

1 Vida Dujmovic and Sue Whitesides. An Efficient Fixed Parameter Tractable Algorithm for 1-Sided Crossing Minimization. *Algorithmica*, 40(1):15–31, 2004. URL: https://doi.org/10.1007/s00453-004-1093-2.

2 Peter Eades and Nicholas C. Wormald. Edge Crossings in Drawings of Bipartite Graphs. *Algorithmica*, 11(4):379–403, 1994. URL: https://doi.org/10.1007/BF01187020.