PACE Challenge 2024: AXS Heuristic Solver Description

Haocheng Zhu
University of Electronic Science and Technology of China, Chengdu, China

Yi Zhou
University of Electronic Science and Technology of China, Chengdu, China

Bo Peng
Southwestern University of Finance and Economics, Chengdu, China

Abstract

The one-sided crossing minimization (OCM) problem involves arranging the nodes of a bipartite graph on two layers (typically horizontal), with one of the layers fixed, aiming to minimize the number of edge crossings. The OCM is one of the basic building blocks used for drawing hierarchical graphs, but it is NP-hard. In this paper, we introduce a local search algorithm to solve this problem. The algorithm is characterized by a dynamic programming-based neigborhoods framework and multiple neighbor move operations. The algorithm has been implemented in C++ and submitted to the 2024 PACE challenge.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases Cross minimization, Local search, Student submission

Supplementary Material The source code is available on GitHub (`https://github.com/axs7385/pace2024`) and Zenodo (`https://doi.org/10.5281/zenodo.11601355`).

1 Preliminaries

The input of the one-sided crossing minimization (OCM) problem is a bipartite graph $G = ((A \cup B), E)$ and a fixed linear order of A. The objective is to find a linear order of B such that the number of edge crossings in a straight-line drawing of G with A and B on two parallel lines, following their linear order, is minimized.

For convenience, we define $n = |A|, m = |B|, k = |E|$. We use p to denote a linear order of B, and p_i to denote the i_{th} node of the order.

Given an order p, for two vertices $u, v \in B$, we define $c_{u,v}$ as the number of edge crossings between u, v while u is positioned in front of v. Note that $c_{u,v} \neq c_{v,u}$ because $c_{v,u}$ is the number of crossings between u, v, while v is in front of u.

2 The General Framework

The graph is represented as an adjacency list from B. First, we pre-process $c_{u,v}$ in $O(mk)$, and we obtain the initial order of B by sorting according to the average number of neighbors of vertices in B. Now, every linear order of B is called a solution.

After that, we use dynamic programming-based local search to obtain several solutions as the initial population, and then perform crossover on them. We execute the genetic algorithm until we reach half the time limit. Finally, we choose the best solution in population. We alternate between dynamic programming local search and block local search, trying to optimize the solution.

It is a **student submission**. Meanwhile, we submit a similar program on the exact and parameterized track.

2.1 Genetic Algorithm

The genetic algorithm in our algorithm follows the regular framework evolutionary algorithm. We maintain a pool of 10 solutions (called chromosome using the language of evolutionary algorithm). In each round, we randomly pick up two chromosomes $p1$ and $p2$ and use a crossover to generate another offspring solution. The generating procedure is specified as follows.

1. Randomly select two crossover positions, a and b, in $p1$, assume $a < b$ w.l.o.g.
2. Copy the subsequence between the two positions, $p1[a, ..., b]$, to the offspring solution *child*.
3. Fill the remaining positions in *child* by remaining elements of B, keeping the order of these elements the same as $p2$.

For example, given two parent chromosomes: $Parent1 = [1, 2, 3, 4, 5]$ and $Parent2 = [5, 4, 3, 2, 1]$, we randomly select two crossover points $3, 4$. Next, we copy the subsequence between the crossover points from $Parent1$ to the offspring, and obtain $Child = [\_, \_, |3, 4, |\_]$. Then, we fill the elements in $B \setminus \{3, 4\}$ by following the order $p2$. We finally obtain $Child = [5, 2, |3, 4, |1]$.

2.2 Dynamic Programming-Based Local Search

The core of our solver is a local search algorithm based on an extended dynamic programming algorithm in [1]. First, suppose that we have an order p. We let $dp(i)$ denote the maximum reduction in the number of edge crossings for the first i vertices, and $f(l, r)$ denote the reduction in the number of edge crossings for one or some *move* within the interval $[l, r]$. We can derive the following dynamic programming transition equation:

$$dp(i) = \begin{cases} 0 & \text{if } i = 0, \\ \max_{j=0..i-1} dp(j) + f(j+1, i) & \text{if } i \in [1, m]. \end{cases}$$

In our algorithm, we include the following types of *move*.

1. **Insert** Without loss of generality, we currently consider only left-to-right insertion. For an order p, we move p_l after p_r. We call this move "Insert". The move can be divided into a sequence of adjacent swaps, that is, the move is equal to swapping p_l with p_{l+1},p_{l+1} with p_{l+2} ,..., and p_{r-1} with p_r. After any swap between two adjacent vertices u, v, the change of the crossing number is $c_{v,u} - c_{u,v}$. Therefore, the crossing number is changed by $\sum_{i=l+1}^{r}(c_{p_i,p_l} - c_{p_l,p_i})$ after inserting p_l after p_r. By preprocessing the prefix sum of $c_{u,v} - c_{v,u}$ according to the order of p, we can quickly calculate the change in crossing number for each insertion.
   The following types of moves can all be decomposed into several insertion operations, so our explanation will be relatively straightforward.
2. **Swap**
   For an order p, we call the swap between p_l and p_r as *swap* move. The swap can be divided into inserting p_l after p_{r-1} and inserting p_r before p_l, the crossing number is changed by$\sum_{i=l+1}^{r-1}(c_{p_i,p_l} - c_{p_l,p_i}) + \sum_{i=l}^{r-1}(c_{p_r,p_i} - c_{p_i,p_r})$.
3. **Block-Insert**
   We modify the insert move by moving a single vertex to moving a continuous block and obtaining the *block-insert*. Without loss of generality, we consider only the insertion of lower-rank vertices into the higher-rank position. That is, for an order p, we move

$p_l, p_{l+1}, \ldots, p_{l+sz-1}$ after p_r where $r > l + sz - 1$ by *block-insert*. The crossing number is changed by $\sum_{i=l}^{l+sz-1}(\sum_{j=l+sz}^{r}(c_{p_j,p_i} - c_{p_i,p_j}))$.

4. **Block-Swap**
   It is clear that we can also swap two blocks. For an order p, the swap between a subsequence p_l, p_{l+1}, ... , p_{l+sz-1} and another subsequence $p_{r-sz`+1}$, $p_{r-sz`+1}$, ..., p_r is called *block-swap*. The crossing number is changed by $\sum_{i=l}^{l+sz-1}(\sum_{j=l+sz}^{r}(c_{p_j,p_i} - c_{p_i,p_j})) + \sum_{i=r-sz`+1}^{r}(\sum_{j=l+sz}^{r-sz`}(c_{p_i,p_j} - c_{p_j,p_i}))$ after a block-swap.

This best size of a block depends heavily on the data instance. To balance time and effectiveness, we set it as 2.

Based on the various moves mentioned above, for each subsequence in the interval $[l, r]$, we select the move that maximizes the reduction of the crossing number to compute $f(l, r)$. We can preprocess the contributions of all moves in $O(m^2)$ time. The time complexity of each transition is also $O(m^2)$, so the overall time complexity for performing dynamic programming is $O(m^2)$.

2.3 Block Local Search

First, suppose that we have a solution p, and we can improve it by calculating a better solution by reordering a subsequence $Block$ which was chosen randomdly.

Let $g(S)$ denote the minimal number of edge crossings for the vertex set $S \subseteq Block$. We can derive the following dynamic programming transition equation:

$$g(S) = \begin{cases} 0 & \text{if } S = \emptyset, \\ \min_{u \in S}(g(S \setminus \{u\}) + \sum_{v \in S \setminus \{u\}} c(v, u)) & \text{if } S \subseteq Block \text{ and } S \neq \emptyset. \end{cases}$$

The time complexity of this dynamic programming is $O(|Block| * 2^{|Block|})$, and the space complexity is $O(2^{|Block|})$. To balance time consumption and effectiveness, the block size is set to 18.

3

References

1 彭博，卢晨贝，赵岳虎，苏宙行，廖毅，吕志鹏. 求解增量二分图优化问题的动态规划驱动的局部搜索算法. 中国科学 *(信息科学)*, pages 582–601, 2021.