PACE 2024: One-Sided Crossing Minimization by Reduction to Weighted Feedback Arc Set

Mohamed Mahmoud Abdelwahab
Lebanese American University, Lebanon

Faisal N. Abu-Khzam
Lebanese American University, Lebanon

Lucas Isenmann
Lebanese American University, Lebanon

Abstract

In this note we describe an approach for solving the One-Sided Crossing Minimization problem, in an attempt to address the PACE 2024 challenge. This problem assumes the input is a bipartite graph where one part (or subset of the vertices) is fixed on a line and the other part consists of vertices that can be reordered on a parallel line with the objective to minimize the total number of edge crossings.

The presented approach consists of transforming the problem into an instance of the Weighted Feedback Arc Set problem. The latter problem asks to find an order on the vertices of an oriented graph so that it minimizes the sum of the weights of the backward arcs (the arcs ab such that $b < a$ in the order). Our solver mainly consists of a branching algorithm with some pre-processing steps and tries to make good use of cuts and reductions.

2012 ACM Subject Classification Mathematics of computing → Discrete mathematics → Graph theory → Graph algorithms

Keywords and phrases one-sided crossing minimization, pace-2024

Digital Object Identifier 10.4230/LIPIcs...

1 One-Sided Crossing Minimization and Weighted Feedback Arc Set

Let G be a bipartite graph with bipartition (A, B), such that the vertices of A are fixed on a line. We look for an order of B on a parallel line which minimizes the number of crossings between the edges of G.

▶ **Definition 1** (Crossings). *Let u and v be two distinct vertices of B and an order $<$ on B. We denote by $c(u, v)$ the number of crossings between the edges incident to u and the edges incident to v supposing that $u < v$.*

Given an instance (A, B, E) of OCM, we define a weighted oriented graph whose vertex set is B and whose arcs and weights depend on the crossings between the two end-vertices of each arc as follows.

▶ **Definition 2** (Associated weighted oriented graph). *We define G' a weighted oriented graph over the vertex set B as follows: There is an arc from x to y if and only if $c(x, y) < c(y, x)$. The weight of this arc is defined by $c(y, x) - c(x, y)$ (which is equal or greater than 1).*

▶ **Definition 3.** *Given an order $<$ on B, a backward arc is an arc ab of G' such that $b < a$.*

We can prove that finding an order on B minimizing the number of crossings is equivalent to finding an order on $V(G')$ minimizing the sum of the weights of the backward arcs. This problem is known as the Weighted Feedback Arc Set problem.

Figure 1 Example of an instance of OCM and its associated weighted oriented graph. The weight of the arc ab is 1 because $c(b,a) = 2$ and $c(a,b) = 1$. Remark that the order abc is minimizing the number of crossings in the OCM instance and is minimizing the sum of the weights of the backward arcs in the oriented graph.

2 Reduction Rules

We consider the strongly connected components of G'. We can prove that solving the Weighted Feedback Arc Set problem for a graph G' is equivalent to solving the problem in each strongly connected component. Indeed, by finding an order for every component, we can find an optimal order by concatenating all the orders with respect to the acyclicity of the components.

2.1 Twin Reduction

Two vertices of B are said to be twins if they have the same adjacencies in A. We can prove that if x and y are twins, then there exists an optimal order such that x and y are consecutive. Therefore we can apply the following reduction: We delete y and we update the weights of the arcs xz incident to x by adding the weights of the arc yz.

2.2 Edge-Disjoint Cycles

For each component, after the twins have been reduced, we look for edge disjoint cycles in the oriented graph. By defining the weight of a cycle by the minimum weight of its arcs, we can prove that the sum of the weights of the edge disjoint cycles gives a lower bound on the minimum weights of backward arcs of any order.

In our algorithm we precompute a set of disjoint edge triangles and oriented 4-cycles in a greedy way.

3 Branching algorithm

We use a classic branching approach. We consider a variable `preorder` which is a list containing the order of the first vertices of the solution. All the vertices which remains to be inserted will be inserted one by one on the right of the preorder. We also keep track of the best current order to compare the other orders found.

The weights of the backward arcs between the vertices of the preorder and the weights of the backward arcs between the vertices of the preorder and the remaining vertices have just to be updated by computing the weights of the backward arcs incident to the newly inserted vertex with the remaining vertices.

We use different variables to keep track of the in-neighbors, the out-neighbors and the weights of the arcs so that the update of these structures and the detections of sources and cuts are linear.

3.1 Sources

If there exists a remaining vertex which has no in-neighbor in the remaining vertices (or equivalently if it is a source in the graph induced by the remaining vertices), then we can prove that this vertex can be in any case be the next one inserted to the preorder.

3.2 Non-optimal cuts

We use several ideas for avoiding to explore a branch when we try to insert a vertex x to the preorder. The idea is to look at a few number of swaps or moves in the preorder to find a strictly better preorder.

3.2.1 Non optimal order of the last k vertices

For some k, if we detect that the last k vertices inserted cannot lead to an optimal order, we cut the branch.

For example, we denote by a the last inserted vertex, if it exists, and by x a vertex that we want to insert. If a is an out-neighbor of x then any extension of this order will not be optimal because we can swap a and x in any extension to get a strictly better order because we remove exactly 1 backward arc.

Another example of a cut is the following: We denote by b and $,a$ the last inserted vertices, it they exist, and by x a vertex that we want to insert. We can suppose that b is not an out-neighbor of a and that a is not neighbor of x otherwise it would have been cut previously. If b is an out-neighbor of x and its weight is strictly bigger than the weight of the possible arcs ba and ax, then we cut. Indeed reordering the vertices bax to xba or axb is strictly better.

In a more general way, if the order of the last k vertices and x is not optimal, then we can avoid to explore this branch. In practice we added cuts on the last 3 and 4 vertices for only some orders.

3.2.2 Non optimal insertion of remaining vertex.

If the vertex we want x to insert has a strictly better place in the preorder, then we do not need to explore this branch.

If there is a vertex in the preorder such that moving it after the vertex x which would be inserted next is strictly better, then we do not need to explore this branch.

3.3 Greedy initial order

Before launching the branching algorithm, we start by computing an order with a $O(n^2)$ greedy algorithm. The algorithm works as follows: For a given order of the vertices, insert one by one the vertices following the previous order in a list by inserting it at the position which minimizes the weights of newly created backward arcs.

We start with different orders (for example by sorting the vertices by ascending degrees, by decreasing in-weights, etc ...) and we iterate the greedy algorithm over the produced order until the newly created order is not better than the previous one. We finally keep the best order.