lmsrusso

Luís M. S. Russo
INESC-ID and Department of Computer Science and Engineering, Instituto Superior Técnico, Universidade de Lisboa.

Abstract

This document describes the solver submitted to the heuristic track of the PACE 2024 challenge. The problem in question consisted in determining a linear ordering for a set of vertexes B, given a bipartite graph $G = ((A \dot\cup B), E)$. The selected ordering should minimize the number of edge crossings in a straight-line drawing of G with A and B on two parallel lines, following their linear order.

Our approach used the simulated annealing metaheuristic, combined with a partial solution storing technique. A state consists in a specific ordering of set B. A state is modified, in the annealing technique, by swapping two adjacent vertexes in B, or by a sequence of suchs swaps. Besides the solution storing technique the solver used the "barycenter" heuristic as the initial state and to perform sub-problem decomposition. Moreover we used fast data structures, and algorithms, to compute the number of crossings, namely Fenwick trees [4] to obtain the total number crossings and a variation of the set interception algorithm by [2], to compute the number of crossings between two nodes of B.

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases Author: Please fill in \keywords macro

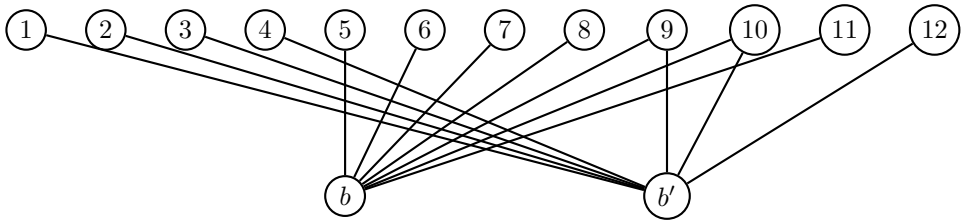
Digital Object Identifier 10.5281/zenodo.11573580

1 Description

The solver was implemented in C. The `graph.h` header handles the loading of the graph information from the `stdin` to memory. The variable `n0` stores the size of the set A, the variable `n1` the size of set B and the variable `m` the number of edges. The storage format is a sort for adjacency matrix. However we store adjacency lists only for the vertexes in B, we had no use for the adjacency of vertexes in A. These adjacency lists are stored as a contiguous interval in array `uint *A`, sorted by their values in A. The beginning and end of each list is indicated by the `uint *C` array. We initialize these arrays with a radix sort procedure in the `loadGraph` function. The radix sort first sorts on A component of the edge and then on the B component. Also in this process, for each vertex $v \in B$ we add up the values of A component of their adjacent edges, into a `double`, for the barycenter heuristic. We also initialize the Fenwick tree data structure and a hash table, used for caching crossing calculations.

Two other crucial functions are the `cjj` and the `totalCross` functions. The `cjj` function determines the number of crossings between the edges that are adjacent to the nodes b and b' of B. As mentioned this process is calculated with an adaptation of the set interception algorithm by [2]. Here is a brief description of the process. We consider the consider the smallest value in Adj(b) and Adj(b'). Assume that it is b, then we use an exponential search for the successor of b in Adj(b'). Now we use the value just found to do an exponential search in Adj(b). Then the process continuous alternating between sets. Hence the overall time of the process is not actually dependent on the size of Adj(b) or Adj(b'). In fact, in a good case, it can take $O(\log n)$ time, when the sets have size n. It actually depends on how interleaved the two sets are, i.e., the actual time is $O(a \log n)$, where a is the number of alternations between sets. Here is an illustrative example:


© Author: Please provide a copyright holder;
licensed under Creative Commons License CC-BY 4.0



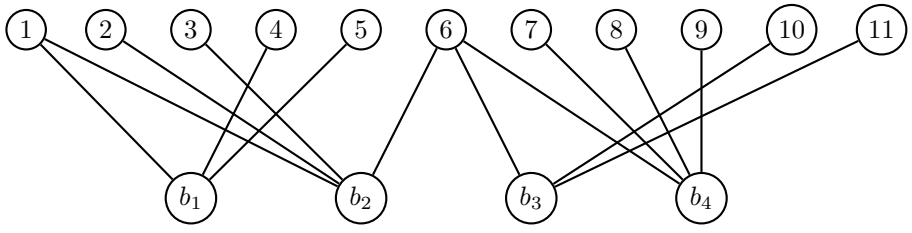
In this case the numbers correspond to the vertexes of A. Hence we first consider 1 and 5. The exponential search of 5 on Adj(b') yields 9. Note that this skips the first four edges of Adj(b'). This indicates that there are four crossing of the edge (b,5) by edges of b'. The next exponential search is for 9 in Adj(b). This also yields 9, by also skipping four edges. Hence this accounts for a total of $4 \times 4 = 16$ crossings. Now the sets are very interleaved and therefore the progress is slower. The exponential search will yield the sequence 10, 10, 11, 12. We considering further improving this process, when the two adjacency sets share a longes common extension. This could be improved with a suffix tree, or suffix array, with LCA computation [5, 1]. However we did not implement this optimization. We further optimized the process by caching these computations. We used a linear probing hash with a least recently used (LRU) discard policy, resumed in the header `LRUHash.h`.

The `totalCross` function performs a similar computation, this time for a whole set of vertexes in B. For this calculation we used a Fenwick Tree. This data structure can be used to store prefix sums. To count crossings we store, for each node of A the number of edges to that node or previous nodes. For example graph above the state of the data structure would progress as follows:

1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0
1	2	3	4	4	4	4	4	5	6	6	7
1	2	3	4	5	6	7	8	10	12	13	14

Initially the structure is all 0's. Then we add the edges from b' and after that the edges from b. In adding the edges from b' this contributes 0 to the total, as each element inspected only contains a zero. However in adding the edges from b we inspect, and update, the indexes from 5 to 11 and therefore obtain $4 + 4 + 4 + 4 + 5 + 6 + 6$ crossings. Note that the updates are performed after all the inspections. Each operation of inspect and update requires $O(\log A)$ time so computing the total amount of crossings in this way takes only $O(E \log A)$ time, where A and E represent the size of the corresponding sets.

An important process that is executed when configuring the initial state is the sub-problem decomposition. In this step we check to see if it possible to divide an instance into two or more independent sub-problems. The next Figure shows an example of such a case.



In this situation it is possible to split the set B into $\{b_1, b_2\} \cup \{b_3, b_4\}$ and A into the set o vertexes less than or equal to 6 and the set of vertexes greater than or equal to 6. Note that node 6 is part of both sub-problems. This division is similar to the notion of suited

pairs given by [3]. To determine this division we compute for each node b in B the min and max values of its adjacency. These values are accumulated left to right for the max and right to left for the min. In the example in the figure we have that $\max b_2 = 6 \leq 6 = \min b_3$. Therefore we can split B into the nodes before b_2 and the nodes after b_3. This conclusion requires an initial ordering of B. For this ordering we used the "barycenter" ordering, as it guarantees that it is possible to determine all such problem divisions.

The consequence of this division is significant in reducing the search space, albeit it is implemented in a subtle way. We do not separate the graph data structure, instead we exclude swaps from the simulated annealing procedure. The nodes b_1 and b_2 are never swapped with the nodes b_3 or b_4. This is implemented by excluding the move $(2, 3)$ from the SA. This represents the node in position 2 in B is never swapped with the node in position 3. Here we are assuming that b_1 or b_2 could be in position 2, in the given example b_2 is in position 2. Also position 3 could have b_3 or b_4. Although this decomposition is crucial for performance its impact is somewhat limited. Consider for example an edge from b_4 to 1. This would invalidate the decomposition, but might end up not having real impact in the annealing process. We therefore devised a way to explore implicit problem decomposition, i.e., keep the best configurations from non interfering changes of the annealing.

We gave a detailed description of SA and temperature selection in [6]. We used the total number of crossings in the current ordering of B as the state energy. The annealing randomly provides a ℓ bound for the current change. Since we are computing a minimization a negative value of ℓ forces a state improvement, whereas a positive value allows us to degrade the current solution. This later case is a tentative process to escape a local minima. A new state is generated by swapping adjacent edges in B. How much such a change contributes to the global value is determined by a difference of the results of the `cjj` function, i.e., `cjj`(b', b) - `cjj`(b, b') would be change for swapping b and b'. We thus store all possible moves, ordered by this value, in a splay tree [7]. We can thus obtain the best possible move in $O(\log B)$ time. This move is used to override excessively low ℓ values. Also we can obtain a random move from the interval of moves that respect the ℓ bound, also in $O(\log B)$ time.

An interesting optimization that we did not implement was to consolidate nodes of B which have the same adjacency set, i.e., $\text{Adj}(b) = \text{Adj}(b')$. In this case we could keep only one of these nodes in the current ordering of B and in the end output these nodes together. Although this is a relevant improvement we considered an even more general approach where even different adjacency sets could be considered the same if there exists no third node b'' which could tell them apart, i.e., for which swapping b with b'' was different from swapping b' with b''. Hence to eliminate some redundant configurations we forced that a node continued being swapped as long as such a change contributed 0 to the total score. More precisely if node b is swapped with the node to its right it would continue moving right until a non-zero change appeared.

The final piece of the solver is a partial solution storing technique. An important problem with the simulated annealing technique is that by default it reports the final state as its solution, independently of it having obtained better configurations during its execution. Therefore we keep a globally best configuration. This configuration is updated, not when a new global best is achieved, but when a new global best configuration is about to be destroyed by a degrading move. Since storing such a solution requires $O(B)$ time this mitigates quadratic time behavior. Further mitigation is obtained by requiring a minimum amount of operations between saves. Moreover besides the global optimum solution we also divide B in half and in quarters and so on and store the optimum for such sub-problems. This avoids the problem a new optimum that is better in one sub-problem but worse in

another.

PUBLIC REPOSITORY: https://github.com/LuisRusso/pace2024-Heuristic

ZENODO: https://zenodo.org/records/11573580

References

1 AFW COULSON. Algorithms on strings, trees and sequences by dan gusfield, cambridge university press 1997, isbn 0 521 58519 8, 534+ xviii pages. *Genetics Research*, 71(1):91–95, 1998.

2 Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '00, page 743752, USA, 2000. Society for Industrial and Applied Mathematics.

3 Vida Dujmovic and Sue Whitesides. An efficient fixed parameter tractable algorithm for 1-sided crossing minimization. *Algorithmica*, 40(1):1531, April 2004. URL: http://dx.doi.org/10.1007/s00453-004-1093-2, doi:10.1007/s00453-004-1093-2.

4 Peter M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336, 1994. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380240306, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380240306, doi:10.1002/spe.4380240306.

5 Gad M Landau and Uzi Vishkin. Introducing efficient parallelism into approximate string matching and a new serial algorithm. In *Proceedings of the Eighteenth annual ACM Symposium on Theory of Computing*, pages 220–230, 1986.

6 Luís M. S. Russo. Searching for a feedback vertex set with the link-cut tree. *J. Comput. Sci.*, 72:102110, 2023.

7 Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652686, jul 1985. doi:10.1145/3828.3835.