


1 **slimmer**

2 **Steffen Limmer** ✉ 

3 Honda Research Institute Europe GmbH, Germany

4 **Nils Einecke** ✉ 

5 Honda Research Institute Europe GmbH, Germany

6 **Abstract**

7 This document describes the solver, we submitted to the heuristic track of the PACE 2024 competition
8 under the user name “slimmer”. The task of the competition is to solve the one-sided minimum
9 crossing problem. For this, we employ a large neighborhood search heuristic.

10 **2012 ACM Subject Classification** Mathematics of computing → Randomized local search

11 **Keywords and phrases** Large neighborhood search, One-sided crossing minimization

12 **Supplementary Material** The source code is available on GitHub ([https://github.com/HRI-EU/](https://github.com/HRI-EU/PACE_LNS)
13 [PACE_LNS](https://github.com/HRI-EU/PACE_LNS)) and Zenodo (<https://zenodo.org/doi/10.5281/zenodo.11608387>).

14 **1 Introduction**

15 In the one-sided minimum crossing problem, a bipartite graph $G = (A \cup B, E)$ is given with
16 node partitions $A = \{l_1, \dots, l_L\}$ and $B = \{r_1, \dots, r_R\}$ and edges E . It is assumed that the
17 graph is drawn, where the nodes are drawn in two parallel lines - one line with the nodes
18 in A and one line with the nodes in B . The order in which the nodes in A are drawn is
19 fixed. The task is to compute an order in which the nodes in B are drawn, which minimizes
20 the number of pairwise crossings of edges in the drawing. Thus, a potential solution is a
21 permutation of the nodes in B (or their indices, respectively).

22 We employ a large neighborhoods search (LNS) heuristic [3] to solve this problem. LNS
23 is a popular heuristic for solving combinatorial optimization problems. Starting with an
24 initial solution, it iteratively destroys and repairs the current incumbent solution (i.e., the
25 best solution found so far). If the resulting solution is better than the incumbent, it is used
26 as new incumbent and otherwise it is rejected. In our case, the destroy and repair simply
27 removes and reinserts nodes from/to the permutation. We employ a restart strategy to avoid
28 premature convergence and a segmentation strategy to handle large problem instances. The
29 solver is implemented in C/C++. The following section describes the solver more in detail.

30 **2 Solver Description**

31 **2.1 Destroy and Repair Operators**

32 Destroy and repair operators are used to remove and to reinsert nodes from/into a given
33 (partial) solution. We employ two different destroy operators:

- 34 ■ Random destroy, which removes randomly selected nodes from the given solution, and
- 35 ■ Block destroy, which randomly selects and removes a contiguous sequence of nodes from
36 the current solution.

37 For the repair, we also employ two different operators:

- 38 ■ Random insertion, which reinserts nodes at randomly selected positions of the current
39 solution, and
- 40 ■ Greedy insertion, which reinserts each node at the (lowest) position that results in a
41 minimum increase of the number of edge crossings.

42 Both repair operators reinsert the nodes in the order in which they are passed to the operators.

43 2.2 Initialization

44 We compute an initial solution by inserting all the nodes of B in increasing order of their
45 indices in an at the beginning empty solution with help of the greedy insertion operator.

46 2.3 Main Loop

47 In the main loop, we randomly select between random and block destroy for removing nodes
48 from the current incumbent solution. Random destroy is selected with a probability of 20%
49 and block destroy is selected with a probability of 80%. If random destroy is selected, the
50 number of nodes to remove is sampled uniformly random between 100 and 150. If block
51 removal is selected, the number of nodes to remove is fixed to 50 and the removed nodes
52 are randomly shuffled before they are passed to the repair operator. For the repair, greedy
53 insertion is used. If the repaired solution has a lower or equal number of edge crossings than
54 the incumbent solution, it is accepted as new incumbent and otherwise it is rejected.

55 2.4 Prevention of Premature Convergence

56 LNS is a rather local search heuristic and especially in combination with a greedy repair
57 strategy, it is prone to getting stuck in a local optimum. A common countermeasure is to add
58 random noise to the objective function in order to make the repair less greedy [2]. However,
59 since we found this strategy to be not beneficial for the given problem, we employ another
60 approach for preventing premature convergence: We do not only keep track of one incumbent
61 solution, but two - a global incumbent and a local incumbent. The global incumbent is
62 the best solution encountered so far. The local incumbent is the solution, the main loop is
63 currently working on and it might differ from the global incumbent. At the beginning of the
64 search, the global and local incumbents are identical. Whenever the number of consecutive
65 iterations of the main loop, which did not yield an improvement of the local incumbent
66 solution, exceeds a certain threshold (set to 2000 in the submitted solver), a new local
67 incumbent is computed by slightly worsen the current global incumbent and the counter for
68 unsuccessful iterations is reset to zero. More precisely, the new local incumbent is computed
69 by removing 20 nodes from the global incumbent with the random destroy operator and
70 reinserting the removed nodes with the random repair operator. Hence, whenever the search
71 makes no more progress, it is guided away from the current local optimum without having to
72 restart completely from scratch.

73 2.5 Performance Improvements

74 In order to be able to construct and to evaluate a high number of solutions in a short
75 time, we applied different techniques to improve the performance. One of these performance
76 improvements is that we precompute a crossing matrix C , where element c_{ij} of the matrix
77 is the number of crossings between edges of nodes r_i and r_j if node r_i appears before node
78 r_j in the permutation. With help of the crossing matrix, a solution can be evaluated very
79 quickly and it allows to determine quickly how the number of crossings changes if a node is
80 shifted from a position k to the next position $k + 1$ in the greedy insertion operator.

81 Another performance improvement is that we do not evaluate a new solution, which is
82 created in the main loop, from scratch. Instead, we update the objective value in the destroy
83 and repair operators based on the operations, which are performed by the operators. If, for
84 example, a node r_j is removed, the number of crossings is reduced by the number of crossings
85 between edges of node r_j and edges of all other nodes of the current solution.

86 Furthermore, we applied different improvements on the code level, like loop unrolling
87 or storing and using a transposed version of the crossing matrix in addition to the original
88 matrix in order to make memory accesses more efficient.

89 2.6 Handling of Large Problem Instances

90 For large instances, we cannot compute the crossing matrix since this is too time and memory
91 consuming. Without crossing matrix, the evaluation of solutions and the greedy insertion
92 operation becomes time consuming. Thus, we employ a special strategy to handle large
93 instances with 15,000 or more nodes in B : If the number of edges exceeds 300,000, we
94 compute an initial solution with the Barycenter heuristic [1] and otherwise an initial solution
95 is computed via greedy insertion similar to the smaller instances. We then further optimize
96 the solution segment-wise. We first divide the solution into contiguous segments of 2000
97 nodes and optimize the order of the nodes in each segment analogous to smaller instances
98 as described in the previous subsections. Per segment, 1000 iterations of LNS are executed.
99 The segment-wise optimization is repeated until the time limit is reached (or a solution with
100 zero crossings is found) and in each repetition, the segment size is increased by 1000 up to a
101 maximum of 5000.

102 ——— References ———

- 103 1 Erkki Mäkinen and Harri Siirtola. The barycenter heuristic and the reorderable matrix.
104 *Informatika (Slovenia)*, 29(3):357–364, 2005.
- 105 2 Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the
106 pickup and delivery problem with time windows. *Transportation Science*, 40(4):455–472, 2006.
107 doi:10.1287/trsc.1050.0135.
- 108 3 Paul Shaw. A new local search algorithm providing high quality solutions to vehicle routing
109 problems. Technical report, University of Strathclyde, 1997.