Assignment 3
COMP6741: Algorithms for Intractable Problems

Students: insert your names and zIDs here

1 Instructions

Assignment 3 is a group assignment. Each group consists of 3-4 members. Each group selects one alternative among the three alternatives described in section 2–4. By mutual agreement, students may form their own groupings. After the census date, the lecturer-in-charge partitions the remaining students into groups (see forum).
Due date. All deadlines are Sydney time. Submitting x hours after the deadline, with $x > 0$, reduces the obtained mark by $5 \cdot \lceil x/24 \rceil$ marks. No submissions will be accepted 5 days (120 hours) or more after the deadline. It is crucial to attempt to submit your report before the day of the group presentation.

How to submit. Before starting your work, create one GIT repository per group (i.e., Github, Bitbucket). The repository may be private or public. Invite the lecturer-in-charge to this repository. The Readme.md file in this repository contains your zIDs and names, followed by a description of the repository contents, instructions how to run the code, and all required software and packages clearly stated. The repository should contain a report which forms the basis for marking the assignment and answers the questions in section 5.
Grade. A part of the Assignment 3 mark is determined by peer grading where students are asked to grade the work of other students based on the collaboration within their own team and the presentation of other teams.

Report + code (LiC)	40%
Presentation (LiC)	20%
Presentation (peers from different groups)	20%
Peer evaluation (peers from same group)	20%

Your group has complete freedom in how to divide the work, but each student should be involved in the final presentation (pre-recorded video presentations are allowed).
Programming environment. All implementations will be done in SageMath (an extension of Python focused on graphs and other mathematical objects). Implementations for Alternative 1 may be written in C/C++ instead. In your group, decide which programming language and version you use and how you plan to collaborate. For example, in Ubuntu 22.04, `sudo apt install sagemath` installs SageMath 9.5 by default.

Make sure to write good quality code, following general conventions, with appropriate naming, spacing, testing procedures, comments and documentation.

Existing implementations and libraries may be reused, as long as their licenses allow unrestricted (academic) use.

2 Alternative 1 - PACE challenge

The PACE challenge is an annual challenge that brings theoretical ideas from parameterized complexity into practice.

In this alternative of Assignment 3, your task revolves around writing a solver for one of the tracks of this year's PACE, submit a report discussing your findings, and giving a presentation about your work.

2.1 Expectations

It is expected that you consult the literature and use sound theoretical underpinnings of your work. Design and discuss various algorithms and aim for several implementations or features that can be switched on and off, so that you can compare several solutions. Think about preprocessing (simplification rules) and postprocessing (once we have found a solution, can we make local modifications to try to get an improved solution), and randomization where we get a variety of solutions by running the algorithm multiple times.

Random numbers If you use randomization, it is recommended to use a built-in pseudo-random number generator as a proxy for random number generators. For your work to be reproducible (and to more easily debug and update results of a battery of tests), it is recommended to work with fixed seeds.

```
import random

random.seed(int(5787549218)) # using a seed helps with reproducibility
print(random.randint(1,100)) # prints the same integer each time we run this code
```

Experimental evaluation To test your implementations, use a small-scale test set of up to 100 instances (these should not all be random instances), describe the origin of these instances, and how you would scale up the testing for a more rigorous experimental analysis.

3 Alternative 2 - educational

Select one concept taught in the course and develop relevant visual learning material.

The implementation of this alternative needs to be in SageMath and develop an interact widget.

Submit both a report and a video demo of your widget to demonstrate how it can be used in teaching.

Examples includes a step-by-step visual execution of an algorithm or a visualization of the effect of graph modifications to the treewidth of the graph.

4 Alternative 3 - open source

Make a contribution to the open-source SageMath project.

Select a topic that is relevant to both the course (i.e., an algorithm solving an NP-complete graph problem) and the SageMath community (i.e., by checking open issues on Github).

Your implementation should be short, easy to understand and well-documented, so as to minimize any issues with code submission and publishing.

Past UNSW students have implemented nice tree decompositions and algorithms for maximum leaf spanning tree and connected dominating set.

5 Questions for all alternatives

Exercise 0. Describe how you collaborated, how you split up the work, and the contributions of each team member. [10 points]

Solution.
Our team engaged in a collaborative university project utilizing several key methodologies and tools. Firstly, we used **Github** for efficient source code version control, enabling seamless collaboration and tracking of code changes. Additionally, we organized both **online and in-person meetings** to facilitate thorough discussions on proposed solutions and to brainstorm ideas derived from academic papers read. We further enhanced our productivity through **pair programming sessions**, where team members worked together in real-time, sharing insights and problem-solving strategies. Moreover, we conducted regular **code reviews** to ensure code quality, identify potential issues, and promote knowledge sharing among team members.

After our initial meeting, tasks were distributed among team members as follows. Theo handled reading inputs and generating relevant data structures, while Deniz focused on developing the top-level minimization function to search for the minimal 'k' value. Collaboratively, the team worked on algorithm development for one-sided crossing minimization, and Erika processed the partially ordered set to output the linear order.

The first of the algorithms developed was a FPT algorithm as per [1]. The team collaborated together to read and process the paper. The implementation of the algorithm in python code was led by Sank and Deniz. Sank and Deniz collaborated on FPT implementation kernels, while Sank handled the branching algorithm, consulting with Theo and Erika where necessary. Deniz developed a brute force implementation, this was ultimately used for benchmarking test cases. Sank and Erika collaborated implementing a secondary FPT implementation as per [2], and Theo explored a randomized algorithm implementation.

Unit testing was employed to test each developed methods functionality. Every team member contributed by adding unit tests as they developed their assigned functionalities. This collaborative effort ensured that each component was thoroughly tested in isolation and enabled fast retesting if any implementation details were updated.
To evaluate algorithm performance, The algorithm was tested against a variety of publicly availble test instances. Deniz developed shell scripts that iteratively executed the algorithms against test instances and recorded benchmark times. This systematic approach to testing enabled us to validate the effectiveness of our algorithms and make informed decisions for further optimization and refinement.

Exercise 1. Describe the design of your project. Describe benchmark instances if relevant. Describe the design of your tests and testing environment. If relevant, how would your benchmarks scale to larger instances. [50 points]

Solution.
A high level process diagram depicting the design of our project is provided in 1. The description of each module is provided below:

- **Read Inputs**: This module reads in the input file, as per the pace2024 challenge specifications and produced the required data structures that were used for executing the algorithm.
- **Maximiser**: This module executes a binary search on an input function and returns the largest value for k for which the input function returns TRUE.
- **Algorithm** This module represents the Algorithm being executed (that is, the input function to the Maximiser). A concrete instance of the algorithm could be the FPT algorithm or the Brute Force Algorithm etc. This is configurable in code.
- **Linear Order Generator** This module transforms the final partial ordering into a single linear order.

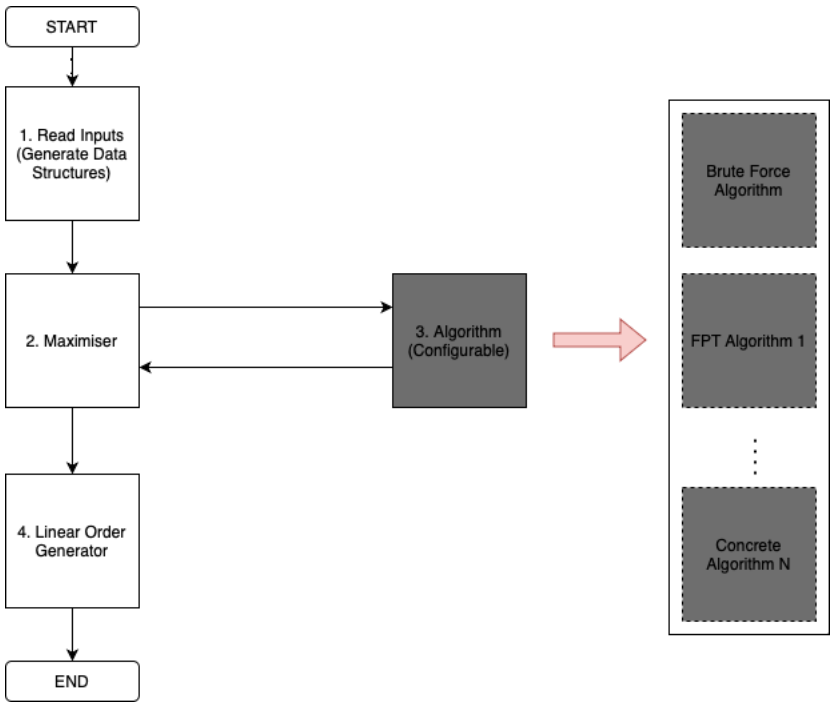

Figure 1: Project Design

A variety of benchmark instances have been provided by PACE2024 which we have used to test the efficacy and optimality of project. Two tests sets were provided, 'Tiny Test Set' as well as 'Medium Test Set'. The latter of which we selected a subset of instances as the set was very large. These instances are described in 1, 2. The tables include the number of vertices in the V_1 and V_2 layers of the bipartite graph, as well as the number of edges in E.

The test environment employed in the project was on two levels. Firstly, a multitude of unit tests were developed as the projects components were built out. This was facilitated using pythons `unittest` package. Unit tests enabled the continuous checking and retesting of small component methods when implementation details were changed by team members. A concrete example of one such unit test is provided in 5. The full range of unit tests deployed for the project can be found in repository under `test.py` and can be executed by `python3 test.py`.

Name	$\|V_1\|$	$\|V_2\|$	$\|E\|$
complete_4_5.gr	4	5	20
cycle_8_shuffled.gr	4	4	8
cycle_8_sorted.gr	4	4	8
grid_9_shuffled.gr	4	5	12
ladder_4_4_shuffled.gr	4	4	10
ladder_4_4_sorted.gr	4	4	10
matching_4_4.gr	4	4	4
path_9_shuffled.gr	5	4	8
path_9_sorted.gr	5	4	8
plane_5_6.gr	5	6	10
star_6.gr	2	6	6
tree_6_10.gr	6	10	15
website_20.gr	10	10	12

Table 1: Tiny Test Set - Benchmark Instances

Name	$\|V_1\|$	$\|V_2\|$	$\|E\|$
1.gr	136	122	257
2.gr	351	327	677
4.gr	26	26	139
5.gr	31	37	250
6.gr	143	152	294
7.gr	244	215	458
8.gr	346	308	653
9.gr	375	344	718
10.gr	393	333	725
11.gr	408	464	871
12.gr	108	124	231
13.gr	165	175	616
14.gr	288	286	1100
15.gr	54	54	162

Table 2: Medium Test Set - Benchmark Instances

Listing 1: Example Unit test

```
def test_computeCross2X2Bipartite(self):

#generate the test instance here
c = {}
V_1 = [1,2]
V_2 = [3,4]
G = {}
G[1] = [4,3]
G[2] = [3,4]
G[3] = [2,1]
G[4] = [1,2]

#method to test
helper.computeAllCrossings(c, G, V_2, V_1)
res = helper.getCrossings(3,4,c)
#assert result against expectation
self.assertEqual(res, 1, "crossings returned: " + str(val))
```

The test framework provided by the PACE2024 team enabled us to automate the testing for our algorithms over a variety of test instances. To do this, we first generated a bash script `solver.sh` 2 which would call our solution. This was then passed into the PACE2024 tester with the command `pace2024tester ./solver.sh` to test against the Tiny Test Set, more details on usage of `pace2024tester` is provided on their website. `pace2024tester` works by executing our solution and then using the resulting certificate (ordering of V_2 to calculate the crossings in G, it would then compare this to a existing certificate that is minimal in the number of crossings and output to the user whether the computed solution is correct or not.

Listing 2: solver.sh

```
#!/bin/bash
chmod +x $0
python3 algorithm.py $1 > $2
```

Additionally, we were interested in timing the execution times of each of the instances, as we wanted to compare times against a variety of different strategies. To do this, we produced a script `benchmark.sh` which would repeatedly call `time python3 algorithm.py` on a variety of instances and store the resulting times in a output file. This output file was then post-processed to generate the tables in 8 and 7

Listing 3: benchmark.sh

```
#!/bin/bash
# Usage
# bash ./benchmark.sh $1 $2
# $1: target parent directory with the test cases
# $2: name of the output file to write the benchmark times to.

echo "" > $2
# Check if the '$1' directory exists
if [ -d "$1" ]; then
    for file in $1/*; do
        echo "time␣python3␣algorithm.py␣$file" >> $2
                { time python3 algorithm.py $file; } 2>> $2
                echo "------------------------------" >>$2
    done
else
    echo "The␣'$1'␣directory␣does␣not␣exist␣in␣the␣current␣working␣directory."
fi
```

Exercise 2. Describe your implementation. Include screenshots, code, and testing examples. Describe challenges you faced, and how you overcame them. [20 points]

Solution.

Implementation Our main algorithm implementation was based off the FPT algorithm presented in [1] with some additional halting rules and simplifications rules. This algorithm solves the decision problem of whether there is a permutation of vertices such that the number of crossings is at most k. However, the problem we want to solve is not the decision problem but the minimization problem. Thus, we adapted this algorithm by conducting a logarithmic search for the minimal value for k.

The FPT algorithm we fully implemented has 3 main components: Kernalization, Simplification Rules and Branching. The reduction rules labeled as RR1, RR2 and RR3 (same as the paper) are applied exhaustively in the kernelization step, we also included early exits whenever a rule is applied such that we don't waste computation time. After applying the kernel, we arrive at our recursive branching function where we first exhaustively apply simplification rules RRL01, RRL02 and RRlarge. The function then branches to different possible orderings by committing a partial order relative to two vertices, for example either committing the order $a < b$ or $a > b$ for vertices a,b. This branching is run until every pair of vertices is comparable in the partial order. If every pair of vertices are comparable then we have a resulting linear order which has crossings at most k.

Listing 4: Implementation of RR1

```
def RR1(po, V_2, c, k, icp):
#any pair of vertices {a.b} in V_2 that form a 0/j pattern,
#commit a < b to the Poset.
n = len(V_2)
for a in range(n):
    for b in range (a, n):
        if(a != b):
            if not incomparable(po, V_2[a], V_2[b]):
                continue
            c_ab = helper.getCrossings(V_2[a],V_2[b], c)
            c_ba = helper.getCrossings(V_2[b],V_2[a], c)
            if((c_ab == 0) and (c_ba > 0)): # commit a < b
                k = helper.commitPartialOrdering(po, V_2[a], V_2[b], k , c, V_2, icp)
            elif((c_ba == 0) and (c_ab > 0)): # commit b < a
                k = helper.commitPartialOrdering(po, V_2[b], V_2[a], k, c, V_2, icp)
            if c_ab == 0 and c_ba == 0:
                k = helper.commitPartialOrdering(po, V_2[b], V_2[a], k, c, V_2, icp)
            if k < 0:
                return k


return k
```

The partial orders we commit to vertices are stored in a python dictionary where the keys and the values represent the following relation, $key < Values$. Whenever we commit a partial order we conduct a depth-first search to find any transitive relations we can commit resulting from the new relation. Once we arrive at a partial order where all pairs of vertices are comparable, we reverse a topological sort to arrive at our linear order.

We also implemented a Brute Force algorithm which utilised existing structures such as the logarithmic search and partial order functions. This Brute Force algorithm would simply commit a partial order and recurse on that state. For example, for a pair of vertices a,b it would recurse in the state where $a < b$ and the state where $b < a$. At worst we will need to commit $\binom{n}{2}$ different relations, where n is the number of vertices. Thus the overall running time of this algorithm is $O^*(2^{n^2})$.

We attempted to refine this brute force algorithm, as simply committing a partial order for each set of vertices would lead to our algorithm testing many invalid cases such as cases, where for example, $a < b$, $b < c$ and $c < a$.

Therefore, we looked for ways to filter out these impossible cases. We explored a method where, with n being the number of vertices in V_2, for each vertex v_k, a number between 0 and k-1 is associated to the vertex. For example, the first vertex can only have associated value 0, the second can have value 0 or 1, the third, 0,1, or 2, and so on. This allows us to define a partial ordering that will always be coherent. First we arbitrarily 'place' v_1, then there are 2 options: $v_2 < v_1$ and $v_1 < v_2$. If the number associated to v_2 is 0, we decide $v_2 < v_1$, and vice-versa. Now that we have 2 vertices placed, there are 3 possibilities for v_3: smaller than both, greater than both, or between one and the other. We choose one of these depending on the number associated to v_3, and so on. This allows us, once complete, to generate a total ordering of the vertices that will always be coherent. By cycling through every value associated to each vertex, we are able to run a smarter brute-force algorithm, that does not waste time on impossible cases. We were not able to fully complete this implementation due to time constraints.

We also made some progress on implementing a different algorithm from [2]. This algorithm approached the problem using dynamic programming and interval graphs. We have implemented the interval graphs which are list of tuples that store the left-most and right-most edge for each vertex. However, we have not been able to fully implement the dynamic programming component at the present time.

We also created new test cases to try and identify the reason behind long run times of our algorithm: for example, we created a case that is essentially a large cycle with a low number of crossings which our algorithm handled well, however another cycle with a larger amount of crossings resulted in very high runtime.

We implemented helper functions to facilitate the overall work: one helper function to read in input from the .gr file, one to write output, as well as helpers for the algorithm such as functions to get a vertex's closed neighborhood, a function to compute the overall number of crossings in the graph, ones to check if two vertices are incomparable, transitive, among others.

Listing 5: Some helper functions

```
def computeAllCrossings(c, G, V_2, V_1):
    n = len(V_2)
    #init 2d array storing crossings
    #c[a][b] indicates the number of crossings for c_ab (ie assuming a<b in Poset)
    for a in range(n):
        c[V_2[a]] = {}
        for b in range(n):
            if(a == b):
                c[V_2[a]][V_2[b]]= -1
            else:
                c[V_2[a]][V_2[b]]= 0
    total = 0
    for a1 in range(n): #vertex in V_2
        for b1 in range (a1, n): #vertex in V_2
            if (a1 == b1):
                continue
            for a2 in G[V_2[a1]]: #vertex in V_1 that has edge incident to a1 in V_2
                for b2 in G[V_2[b1]]: #vertex in V_1 that has edge incident to b1 in V_2
                    a1_i = a1
                    b1_i = b1
                    a2_i = V_1.index(a2)
                    b2_i = V_1.index(b2)
                    if(a2_i == b2_i):
                        continue
                    if (a1_i < b1_i) and (a2_i > b2_i):
                        c[V_2[a1_i]][V_2[b1_i]]+=1
                    elif(a1_i < b1_i) and (a2_i < b2_i):
                        c[V_2[b1_i]][V_2[a1_i]]+=1 #swapping them would force a crossing
                    elif(b1_i < a1_i) and (b2_i > a2_i):
                        c[V_2[b1_i]][V_2[a1_i]]+=1
                    elif(b1_i < a1_i) and (b2_i < a2_i):
                        c[V_2[a1_i]][V_2[b1_i]]+=1 #swapping them would force a crossing
        total += min(c[V_2[b1_i]][V_2[a1_i]], c[V_2[a1_i]][V_2[b1_i]])
    return total
```

```
35
36 def insertPartialOrdering(po, a, b, icp = None):
37     if a not in po.keys():
38         po[a] = []
39     if b not in po[a]:
40         po[a].append(b)
41         if icp:
42             icp[a].remove(b)
43             icp[b].remove(a)
```

Challenges

Our initial readings and investigations into the problem, by reviewing [1]. We faced challenges in our comprehension of the logic underpinning the branching component the FPT algorithm. We were able to overcome this through our group meetings and online discussions.

Time management during the course of the project presented as a challenge. At the outset of the project we committed a significant portion of our time to investigation and implementation of the algorithms,we consequently underestimated how long the instance-testing phase would take.

Instance testing involving our own custom instances was challenging as we had to balance creating larger instances without making them too complex (with obvious minimal k values so that we could check against the generated solution).

Instance Testing, debugging issues with large complex instances was challenging. We would often find that the algorithm would not return, requiring us to use the debugger tools to determine whether the algorithm was 'hanging' due to some kind of logical error, or simply executing slowly due to the size of the search tree.

Improving time complexity. After implementation, we spent significant time in pair programming environments to review code and improve execution speeds.

Exercise 3. Describe the outcomes of your project and draw conclusions about its merits. What future work is needed? [20 points]

Solution.
We received the following results running both the FPT algorithm and the Brute force algorithm on the benchmark instances. For the tiny set set the FPT algorithm and Brute Force ran with similar execution times, however, this is expected for small instances (small n). Although, in the website_20 instance we notice that in the brute force algorithm its execution time has significantly grown relative to the FPT algorithm. This due to the fact that the Brute Force algorithm grows faster with bigger n compared to the growth rate of the FPT algorithm with bigger k values.

Table 3: Parameterized

Test Case	**User Time**
complete_4_5.gr	0m0.022s
cycle_8_shuffled.gr	0m0.015s
cycle_8_sorted.gr	0m0.012s
grid_9_shuffled.gr	0m0.012s
ladder_4_4_shuffled.gr	0m0.011s
ladder_4_4_sorted.gr	0m0.010s
matching_4_4.gr	0m0.010s
path_9_shuffled.gr	0m0.010s
path_9_sorted.gr	0m0.010s
plane_5_6.gr	0m0.010s
star_6.gr	0m0.010s
tree_6_10.gr	0m0.014s
website_20.gr	0m0.014s

Table 4: Brute Force 1

Test Case	**User Time**
complete_4_5.gr	0m0.016s
cycle_8_shuffled.gr	0m0.014s
cycle_8_sorted.gr	0m0.012s
grid_9_shuffled.gr	0m0.015s
ladder_4_4_shuffled.gr	0m0.011s
ladder_4_4_sorted.gr	0m0.011s
matching_4_4.gr	0m0.010s
path_9_shuffled.gr	0m0.011s
path_9_sorted.gr	0m0.010s
plane_5_6.gr	0m0.010s
star_6.gr	0m0.010s
tree_6_10.gr	0m0.018s
website_20.gr	0m0.757s

Table 5: Brute Force 2

Test Set	**User Time**
complete_4_5.gr	0m0.000s
cycle_8_shuffled.gr	0m0.012s
cycle_8_sorted.gr	0m0.009s
grid_9_shuffled.gr	0m0.006s
ladder_4_4_shuffled.gr	0m0.012s
ladder_4_4_sorted.gr	0m0.008s
matching_4_4.gr	0m0.000s
path_9_shuffled.gr	0m0.012s
path_9_sorted.gr	0m0.008s
plane_5_6.gr	0m0.006s
star_6.gr	0m0.003s
tree_6_10.gr	0m0.010s
website_20.gr	0m0.015s

We really see the speed-ups of the FPT algorithm with larger instances that have small crossing numbers.

Table 6: FPT Algorithm

Test Case	User Time (s)
1	0m 4.789s
2	4m 6.36s
19_sorted_cylce	0.397
3star_15	0.008
shuffled_20cycle	0.133
star_20	0.015

Table 7: Brute Force 1

Test Case	User Time (s)
1	1hr
2	N/A
19_sorted_cycle	5.972
3star_15	0.007
shuffled_20cycle	6.375
star_20	0.027

Table 8: Brute force 2

Test Case	User Time (s)
1	30min
2	N/A
19_sorted_cylce	0.197
3star_15	0.013
shuffled_20cycle	2.296
star_20	0.013

Our algorithm is still quite slow for larger inputs. In the future we hope to create further speed ups and optimizations within our program. We have noticed that our current algorithm that extends the partial order due to transitivity is very slow, it is currently implemented via a recursive depth-first search. We hope to either find a faster algorithm or consider an iterative approach to extend these partial orders more efficiently.

Our program currently runs the FPT algorithm using a logarithmic search to find the minimal crossing number. Our search begins at k = 1 and doubles at every unsuccessfully instance, and once it finds a successful instances it conducts a binary search to find the optimal k value. We hope to implement further halting rules which would speed up false instances when k is low relative to the actual k value for the instance.

We also hope to fully implement the dynamic programming FPT algorithm which has a better running time overall than the current branching FPT algorithm we have implemented. This algorithm would also not require the logarithmic search to find the optimal k value, hence, we hope that it will be able to provide solution to instances which are currently running to slow.

Experiment Evaluation Theo constructed the tests shown on Table 6 - 8, by starting with known configurations and adding vertices individually to track the minimal crossing number. We did this with instances that

had up to 20 vertices. Many configurations were creating with low crossing numbers from crossing patterns and otherwise with higher crossing numbers were from complete and near-complete graphs. We would scale-up our current testing structure by creating automated testing architecture and programs that develop instances from known configurations.

References

[1] Vida Dujmović, Henning Fernau, and Michael Kaufmann. "Fixed parameter algorithms for one-sided crossing minimization revisited". In: *Journal of Discrete Algorithms* 6.2 (2008). Selected papers from CompBioNets 2004, pp. 313–323. ISSN: 1570-8667. DOI: https://doi.org/10.1016/j.jda.2006.12.008. URL: https://www.sciencedirect.com/science/article/pii/S1570866707000469.

[2] Yasuaki Kobayashi and Hisao Tamaki. "A Fast and Simple Subexponential Fixed Parameter Algorithm for One-Sided Crossing Minimization". In: *Algorithmica* 72.3 (2015), pp. 778–790. ISSN: 1432-0541. DOI: 10.1007/s00453-014-9872-x. URL: https://doi.org/10.1007/s00453-014-9872-x.