OCM PROBLEM

Heuristics and solutions

March 25, 2024

Thibaut de Saivre

ÉCOLE POLYTECHNIQUE
INSTITUT POLYTECHNIQUE DE PARIS
TYPOGRAPHIX

CONTENTS

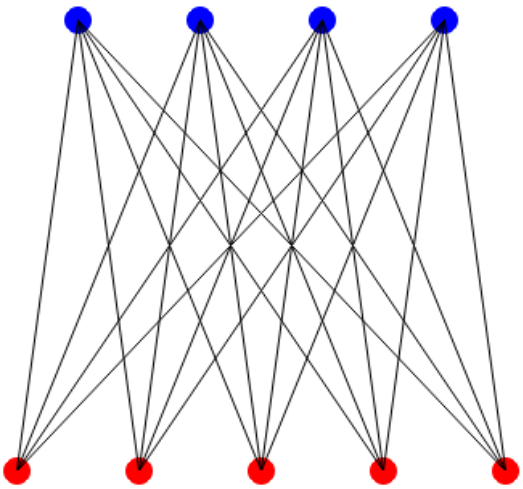

Figure 1: An example bipartite graph

1 ALGORITHMS USED

We present here the different algorithms used in this project, with the objective to solve the One Sided Crossing Minimization problem from the PACE 2024 challenge. They were implemented in Rust, with GTK windowed display support. See the README.md file for more information.

For all graphs, we will consider the following notations:

- V the number of vertices
- E the number of edges
- C the number of crossings
- *optimal* the number of crossings of the optimal solution

We implemented the following algorithms:

- Line Sweep crossing counting
- Barycentric heuristic
- Median heuristic
- Iterated Barycentric
- Iterated Median

1.1 Barycentric heuristic

This algorithm as well as the median heuristic one are described in [2].

- Runtime Complexity: $O(E+V)$
- Space Complexity: $O(E+V)$
- Solution Optimality: $K \times$ optimal with $K = \mathcal{O}(\sqrt{V})$

For this algorithm, we give every top and bottom node an abscissa depending on their order in their respective horizontal lines in the bipartite graph. We then update each node's abscissa by taking the average of the abscissas of the nodes it is connected to. If a node has no connections, it keeps its abscissa.

Algorithm 1: Barycentric algorithm

Input: A bipartite graph
Output: A crossing-minimized graph

```
forall nodes do
    abscissa ← order in the node's horizontal line
forall nodes do
    if connections > 0 then
        abscissa ← average of connected nodes' abscissas
    else
        nothing
forall nodes do
    abscissa ← order in the node's horizontal line
```

The algorithm iterates over all nodes and edges not in a nested way, thus it has a runtime complexity of $O(E+V)$. Note that if you want to associate a rank index to each node of the new graph, you will need to sort the nodes by their abscissa, which has a complexity of $O(V \log V)$.

1.2 Median heuristic

Same as the barycentric one, this algorithm is described in [2].

- Runtime Complexity: $O(E+V)$
- Space Complexity: $O(E+V)$
- Solution Optimality: $3 \times$ optimal

For this algorithm, we give every top and bottom node an abscissa depending on their order in their respective horizontal lines in the bipartite graph, but instead of updating node abscissas with the average of their neighbors, we take their median. If a node has no connections, it keeps its abscissa. If a node has an even amount of connections, we take the average of the two middle abscissas.

Algorithm 2: Median algorithm

Input: A bipartite graph
Output: A crossing-minimized graph

```
forall nodes do
    abscissa ← order in the node's horizontal line
forall nodes do
    if connections > 0 then
        abscissa ← median of connected nodes' abscissas
    else
        nothing
forall nodes do
    abscissa ← order in the node's horizontal line
```

The median of a list of edges can theoretically be computed in $O(n)$ time, but our implementation uses $O(n \log n)$ sorting for simplicity. However, this detail does not show in the benchmarks. The algorithm iterates over all nodes and edges not in a nested way, thus it has a runtime complexity of $O(E+V)$. Note that if you want to associate a rank index to each node of the new graph, you will need to sort the nodes by their abscissa, which has a complexity of $O(V \log V)$.

1.3 Line Sweep crossing counting

Before we present the next solving algorithms, we must present the Line Sweep crossing counting algorithm, which is used to count the number of crossings in a graph. It is especially useful for iterative algorithms as a termination criteria, although it is expensive.

- Runtime Complexity: Worst Case $O((E+V)\times E)$, average case $O((E+V)\log E)$
- Space Complexity: $O(E)$

The idea for this algorithm is to sort edges by order of apparition from to right. Assuming that top and bottom nodes are numbered starting from 0, this means sorting edges (i, j) in order of $\min(i, j)$. We then iterate continuously over the node indices, from 0 to max(top_index, bottom|-index), add appearing edges to an **active edge set** with which we check crossings, and remove disappearing edges from the set.

Algorithm 3: Line Sweep Crossing Counting

Input: A bipartite graph
Output: A crossing count

```
active_edges ← empty set of edges
sorted_edges ← sorted edges by min(i, j)
crossings ← 0
for index in range(0, max_index) do
    Remove disappearing edges from active_edges (edges with max(i, j) == current index)
    for each appearing edge in sorted_edges do
        crossings ← crossings + edges in active_edges crossing the new edge
        active_edges ← active_edges + appearing edge
return crossings
```

We iterate concurrently over all edges and nodes. In each iteration, we compare the current edge with all currently active edges. In the worst case, if all edges are active at the same time (very large and transversal crossing pattern), the active edges contain all edges. The global worst-case complexity thus is $O((E+V)\times E)$. However, in average simple cases, the complexity is closer to $O((E+V)\times$ average edge span), meaning $O((E+V)\log E)$ or even $O(E+V)$.

1.4 Iterated Barycentric

The iterated barycentric algorithm consists in repeating the barycentric algorithm multiple times and comparing the crossings count of each iteration to keep the best one.
We stop iterating once the number of crossings does not decrease anymore (or even starts increasing). This algorithm is described in [1]

- Runtime Complexity: $O((E+V)\times E)\times$ iterations
- Space Complexity: $O(E+V)$

One advantage of this method is to not give a worse solution than the initial one, because the barycentric method sometimes outputs a worse graph on the first iteration. The line sweep crossing counting algorithm used after each iteration gives its complexity to this algorithm.

Algorithm 4: Iterated Barycentric algorithm

Input: A bipartite graph
Output: A crossing-minimized graph

```
best_crossings ← line_sweep(current_graph)
new_crossings ← best_crossings - 1; best_graph ← current_graph
while new_crossings < best_crossings do
    graph ← barycentric_iteration(graph)
    new_crossings ← line_sweep(graph)
    if new_crossings < best_crossings then
        best_crossings ← new_crossings
        best_graph ← graph
return best_graph
```

1.5 Iterated Median

The iterated median algorithm consists in repeating the median algorithm multiple times and comparing the crossings count of each iteration to keep the best one.
We stop iterating once the number of crossings does not decrease anymore (or even starts increasing). This algorithm is described in [1]
It is almost the same as the iterated barycentric algorithm, but with the median algorithm instead of the barycentric one.

- Runtime Complexity: $O((E+V) \times E) \times$ iterations
- Space Complexity: $O(E+V)$

Algorithm 5: Iterated Median algorithm

Input: A bipartite graph
Output: A crossing-minimized graph

```
best_crossings ← line_sweep(current_graph)
new_crossings ← best_crossings - 1; best_graph ← current_graph
while new_crossings < best_crossings do
    graph ← median_iteration(graph)
    new_crossings ← line_sweep(graph)
    if new_crossings < best_crossings then
        best_crossings ← new_crossings
        best_graph ← graph
return best_graph
```

This method gives similar results and advantages to the barycentric one.

2 BENCHMARKS

2.1 Runtime Complexity

In this section we compare and analyze the runtime complexity of the presented algorithms.

- Barycentric vs Median

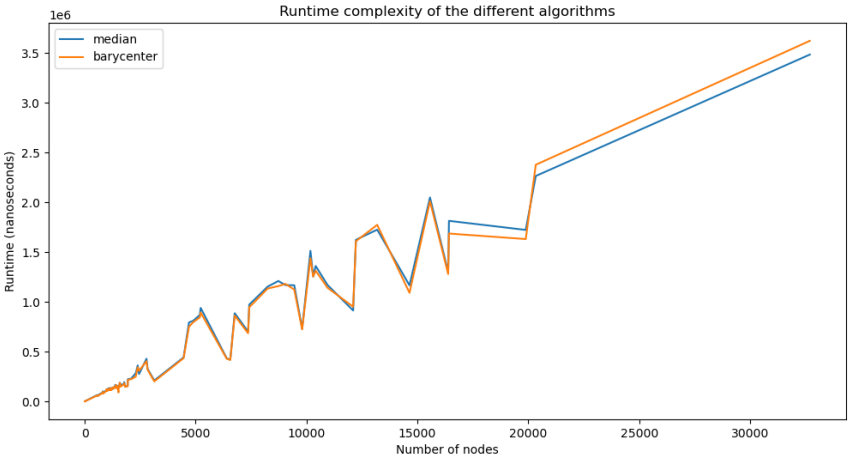

Figure 2: Runtime complexity relative to node count

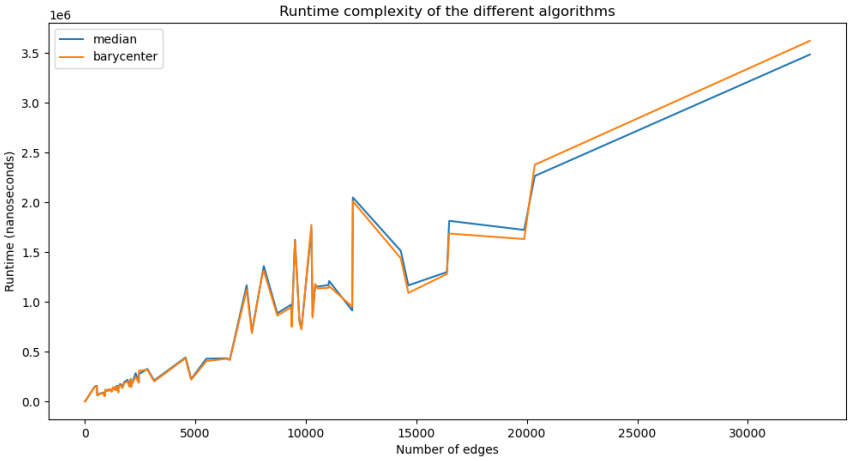

Figure 3: Runtime complexity relative to edge count

As expected, these figures show that the single iteration median and barycentric algorithms have a linear runtime complexity relative to the number of nodes and edges.

- Line Sweep Crossing Count

We compared the runtime complexity of the line sweep crossing counting algorithm to different combinations of edge and node counts. As expected from the analysis, the best fitting complexity is $O((E+V) \times E)$. You can find more complexity computations on the jupyter notebook.

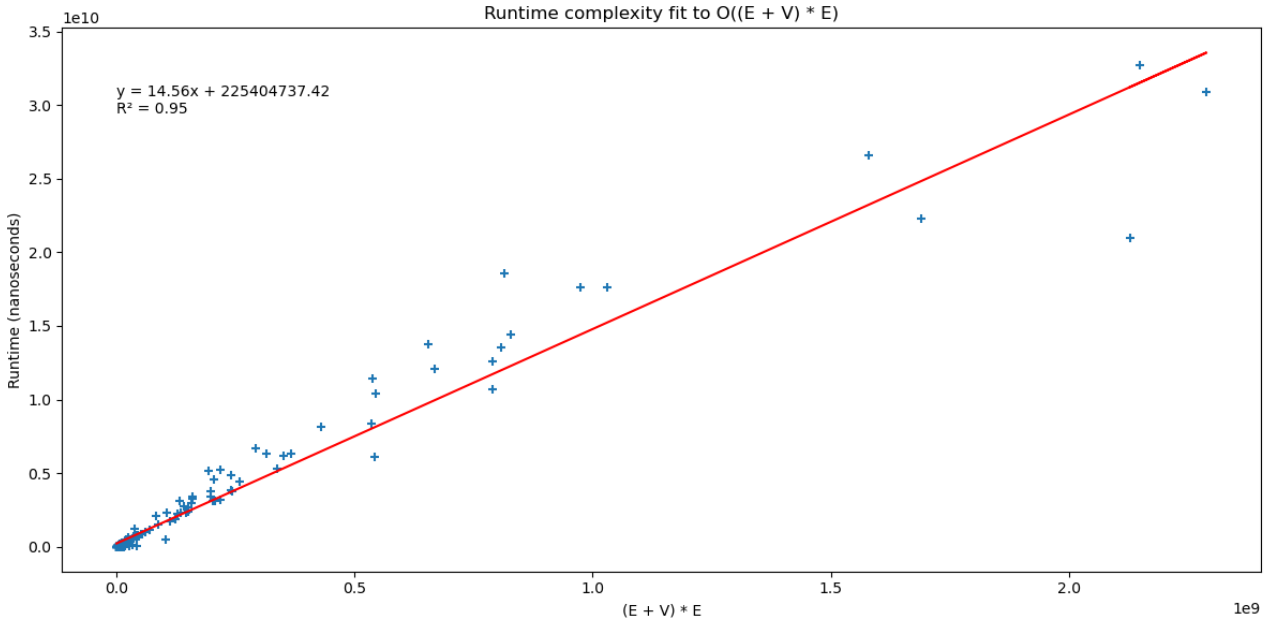

Figure 4: Line sweep runtime complexity

2.2 Performance

TODO

REFERENCES

[1] S. Tagawa K. Sugiyama and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Trans. Systems Man Cybernet.*, 11(2):109–125, 1981.

[2] Nicholas C. Wormald Peter Eades. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11(4):379–403, 1994.