Layouting Phylogenetic Networks using MIQP

Author: Jules Kreuer
Contact: jules.kreuer@uni-tuebingen.de
Date of submission: December 27, 2022
Project available on: Github

Introduction

In this technical report, we will discuss the layouting of phylogenetic networks using Mixed-Integer Quadratic Programming (MIQP). Our goal is to create an objective function and constraints that accurately capture the quality of a network and can be optimised to find a visually pleasing layout.
Proper positioning of nodes in a phylogenetic network is crucial for its readability and overall aesthetic appeal. Incorrect positioning can lead to long hybrid edges spanning across the whole network, making it cluttered and difficult to interpret.
To layout a phylogenetic network using MIQP, one first needs to define constraints that are describing the basic network properties. Furthermore an objective function, which represents the measure of goodness of the network layout is needed. The objective function can be defined based on various criteria, such as the number of crossings, the amount of overlap between branches, or the overall compactness of the network.
Once the objective function has been defined, the optimisation problem can be translated into code where the decision variables are the positions of the nodes in the network. The constraints of the problem can include various layout requirements, such as maintaining a minimum distance between nodes or to avoid overlapping branches.
However, this method does have its limitations, which will be tested and discuss in detail. Overall, our goal is to provide a comprehensive overview of using MIQP for visualising phylogenetic networks and its potential advantages and drawbacks. We will be using *IBM CPLEX 22.1.0* as our solver in combination with the *python 3.10 docplex* module.

Background

The layout and visualisation of networks is a longstanding problem in the field of graph theory and information visualisation. The goal of such visualisation is to effectively convey the structure and relationships within a network to the viewer, while also making the visualisation aesthetically pleasing [1]. Several heuristics are commonly used, including minimising the number of edges that are crossing, clustering similar nodes or closely related nodes, and maximising edge orthogonality [2].
Some of these heuristics will be implemented using Integer Linear Programming (ILP) / MIQP. ILP is a type of mathematical optimisation problem that involves finding the values of a set of decision variables that minimise or maximise an objective function, subject to a set of linear constraints. In an ILP, all of the decision variables are required to take on integer values, rather than continuous or fractional values. This can make ILP problems more challenging to solve, as the solution space is discrete and often larger than for continuous optimisation problems.
A MIQP is a type of optimisation problem that combines the features of an ILP with those of a Quadratic Programm (QP). In a MIQP, the objective function and constraints can include both linear and nonlinear terms, allowing for greater modelling flexibility. However, like in an ILP, the decision variables in a MIQP must take on integer values.
To create an ILP or MIQP program, one must first define the objective function and constraints. The objective function defines the measure of success for the optimisation, such as minimising cost or maximising profit. The constraints define the limitations or conditions that must be satisfied by the solution.
Once the objective function and constraints are defined, a solver can be used to find the optimal solution. There are various solvers available, such as IBM Ilog CPLEX [3], Gurobi, and GLPK.

Creating the MIQP

The are several ways to quantify the visual quality of a phylogenetic network. For trees, it is as simple as counting the edges that cross. For networks, it is important how cluttered they are or how long hybrid-edges are.

Terminology, Base Constraints and Objective

A tree is a data structure that consists of nodes organised in a hierarchy. Each node is connected to exactly one parent node by a directed edge and can have zero or more child nodes. In this work, trees are not oriented from top to bottom but rather from left to right. This unusual configuration allows for easier implementation of the constraints. The left-most node in the tree, which has no parent node, is called the root and is positioned at $x = 0$. The children of a node are placed one step to the right of their parents at $x = p + 1$. For example, the children of the root are at $x = 1$.
After loading a tree from a Graph Markup Language (GML) file, this positioning can be done by calling the `auto_posx()` method othe the `Graph` object.
This discrete positioning on the x-axis creates layers which will be important later on.
To ensure that a tree has certain basic properties, we add two constraints:

1. The y-position of every node must be greater than 0.
2. No two nodes can have the same position.

This second constraint was reformulated so that the vertical distance between any two nodes in the same layer must be greater than or equal to 1. The reformulation was necessary as the interface does not support "greater than" inequalities. When edges are added to a tree, resulting in nodes with in-degree ≥ 2, the resulting structure is no longer a tree, but rather a network. If the solver is able to find a y-positioning of the nodes that satisfies all constraints, it is said to have found a solution. An optimal solution is one that results in a visually pleasing tree or network.

Trees

Code of this sub-project can be found in: `01_tree_const`.
The main property of a visually pleasing tree is that its edges do not cross each other. This property can be represented using simple a constraint. For each pair of nodes the following property must hold:

$$N_A < N_B \wedge C_A < C_B \quad \vee \quad N_A > N_B \wedge C_A > C_B \tag{1}$$

With N_A, N_B nodes of a layer, $N_A \neq N_B$ and C_A, C_B a child of N_A and N_B respectively. Visualised in figure 1 the constraint is only fulfilled in the top two configurations.

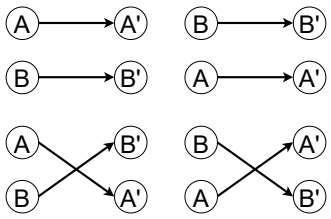

Figure 1: The 4 possible configurations. Two of them are crossing.

Fortunately, CPLEX and its python interface supports logical operations. This constraint represented as code would look like this:

```
model.add_constraint(
    model.logical_or(
        model.logical_and( mvar_node_a <= mvar_node_b, mvar_child_a <= mvar_child_b ),
        model.logical_and( mvar_node_b <= mvar_node_a, mvar_child_b <= mvar_child_a )
    ))
```

Here `mvar_node_a` and `mvar_node_b` are nodes of the same layer with `mvar_child_a` and `mvar_child_b` a children of this nodes respectively. Using the `itertools.combinations` package and a nested for loop this constraint is applied on every combination of nodes / children.
However, the time required to solve this problem is highly dependent on the balance of the tree. When the tree is unbalanced, the number of constraints required is small, as there are only a few nodes per layer. When the tree is balanced, the number of constraints required is significantly larger, as it is given by the total number of combinations. As a result, solving the quadratic growing number of constraints for balanced trees is slower.
An objective function is added in order to make the trees not only free of edge crossings but also visually pleasing. The function aims to position the parents as close to the centre of their children as possible.
In an effort to reduce the solving time, we implemented two constraints that aimed to limit the search space to a more feasible size:

1. Maximum height for every node
2. A maximum vertical distance between a parent and its child node.

To evaluate the effectiveness of these constraints, we measured the time to solve 200 random trees. Our results showed that both a lower and upper bound on the search space had a slight positive effect on reducing the average time and scatter. However, the second constraint had a negative impact on solving time. This was observed for both balanced and unbalanced trees. A detailed view can be found in figure 10.
To evaluate the capabilities of this constraint enriched ILP approach, we attempted to solve 1000 randomly generated trees ranging in size from 3 to 42. A maximum solving time of 20 seconds was imposed. The impact of balanced trees on the performance of the algorithm is depicted in figure 5. While almost every unbalanced tree up to 42 nodes was solved in less than a second, CPLEX started to struggle with balanced trees of size 21. Here the time increased and reached the cut-off of 20s.

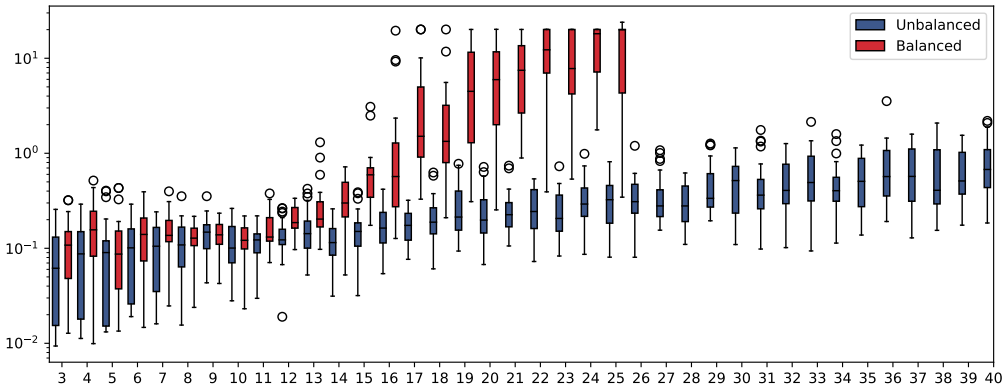

Figure 2: Time to solve trees of size n (x-axis) in seconds (y-axis, log-scale).

Extensions of Trees

In this section, we will extend standard trees with additional edges, and subsequently, generate modified constraints and an objective function.

Horizontal Gene Transfer

Code of this sub-project can be found in: `02_horiz_embedding_obj`.
In order to simulate horizontal gene transfer, edges are added to a tree such that they only connect nodes within the same layer. The goal of the objective is to find a configuration where nodes that share a hybrid edge are close. Other tree visualisation algorithms achieve this by flipping subtrees recursively in order to find the optimal position. In our case, the objective is to minimise the square of the length of the hybrid-edges.
It is expected that the solver will be able to find a solution quickly, as the constraints of the problem are not affected by the addition of hybrid edges. However, it is possible that it may take some time to find the optimal solution.
Similarly to the previous section, 1000 balanced and unbalanced trees were generated. A performance overview can be found in the appendix 11.

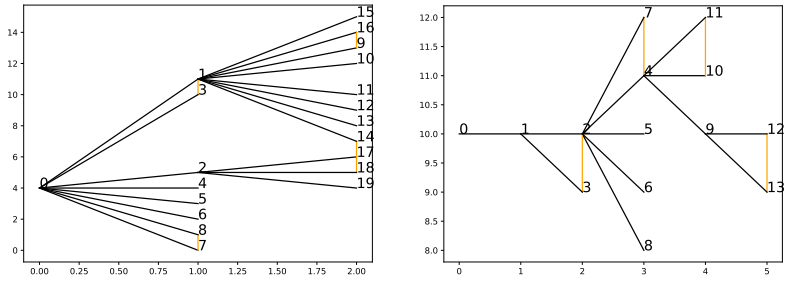

Figure 3: Example of an balanced and unbalanced tree with hybrid edges inside one layer.

Observations have shown that unbalanced trees with hybrid edges are solved effectively. Example trees are shown in figure 3 However, for balanced trees, the same problems that exist without hybrid edges may persist. This is particularly true for larger (size ≥ 20) problems, where the solutions obtained may not be optimal. Overall, the efficiency and effectiveness of the solver in finding solutions for tree structures with hybrid edges may depend on the balance of the tree and the size of the problem.

Hybrids from n to n+1

Code of this sub-project can be found in: `03_hybrid_embedding_obj`.
To the previous tree structure, additional edges were added from layer n to n+1. Having a total of three edge types (regular, horizontal, n to n+1), the previously mentioned constraints were only applied to the tree backbone with its regular edges. The hybrid edges are ignored entirely.
To optimise the tree structure, the objective function was extended with the goal of minimising the number of crossings. While this initially seemed like a good idea, there were several difficulties in implementing this objective. Specifically, it was not possible to include control-flow elements like "if/else". As a result, a workaround was developed using only max/abs functions:

$$low(a,b) = max((b-a) - |b-a|, \ -1) \tag{2}$$
$$q1 = low(\mathrm{N}_a, \mathrm{N}_b) + low(\mathrm{C}_a, \mathrm{C}_b) \tag{3}$$
$$q2 = |(low(\mathrm{N}_b, \mathrm{N}_a) + low(\mathrm{C}_b, \mathrm{C}_a)| \tag{4}$$
$$obj = \left(\frac{|q1+q2|}{2} - 1\right)^2 \tag{5}$$

The function `low(a,b)`, defined on the set of integers $\mathbb{Z}$, returns 0 if $a < b$ and -1 otherwise. If the edge connecting N_a to C_a crosses the edge connecting N_b to C_b, then the variable `obj` is equal to 1. Otherwise, if the edges do not cross, `obj` is equal to 0.

This objective is applied only on hybrid edges due to its computationally demanding nature. While the solver is able to effectively minimise the crossing using this objective, the maximum number of nodes is limited to around 20 for both balanced and unbalanced trees. The performance of this single objective can be observed in the figure 4 provided, where the objective `min_direct_crossing` is turned off and on exclusively.
The results indicate that the best outcome is achieved when the computational heavy objective is turned off. It took a total of 3.31 seconds to solve this network. The same result was computed when the objective `min_direct_crossing` was turned on and other objectives like the minimisation of hybrid length were active as well. Here the computation took 20 seconds. This may be due to the difficulty in optimising objectives with step-like properties (crossing: 1, not crossing 0) and the solver being designed to solve continuous functions. This problem of a "large objective shift" is also recognised during the solving process when such jump is made.

The performance of multiple trees was tested, and the results were depicted in the following graphs. It is evident from the graphs and the produced trees that the implementation of the `min_direct_crossing` objective increased the complexity for the solver.

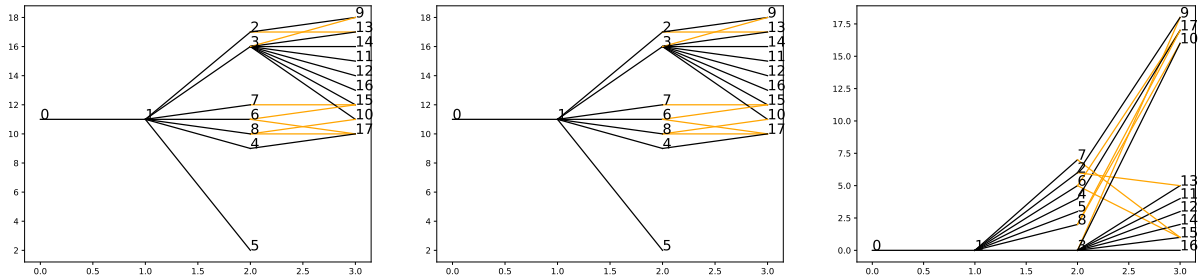

Figure 4: **Left**: `min_direct_crossing` tuned off. Other objectives turned on. The best possible positioning (4 crossings) was found in 3.31s. **Centre**: `min_direct_crossing` tuned on. Other objectives turned on. **Right**: Only `min_direct_crossing` active. Time-limit of 20s reached.

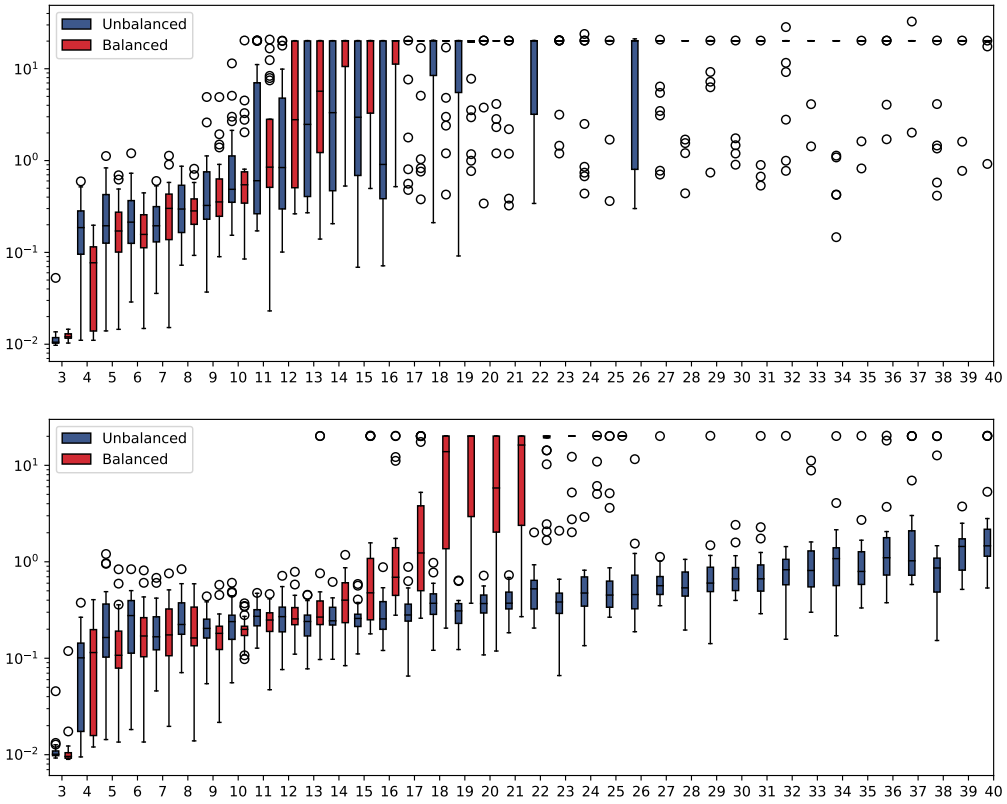

Figure 5: Time to solve trees of size n (x-axis) in seconds (y-axis, log-scale). **Top**: `min_direct_crossing` tuned on. **Bottom**: `min_direct_crossing` tuned off.

Multi-layer hybrids

Code of this sub-project can be found in: `04_multi_embedding_obj`.
As part of the investigation into the capabilities of using MIQP on increasingly complex structures, the next step is to allow for the use of hybrid edges in general, rather than just between adjacent layers (x to x+1).

To avoid the need for modifying the objective function, a method from the Sugiyama algorithm is employed [4]. This involves introducing dummy nodes with in-degree and out-degree of one into edges that span multiple layers, effectively converting the trees into those described in the previous section. Upon completion of the optimisation, these dummy nodes are removed.

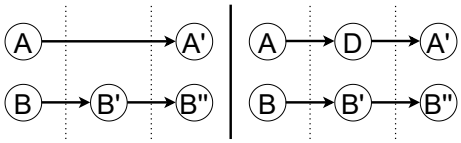

Figure 6: The dummy node `D` is inserted into the edge from `A` to `A'`

Similar results were observed due to the similarities with the previous problem. However, as there were slightly more nodes present in these problems, the time limit of 20 seconds was reached slightly earlier (see figure 12). We found that the objective function used did not adequately capture the quality of the trees. Specifically, we observed that long hybrid edges were crossing other edges, which was largely due to the objective of minimising the vertical hybrid edge length. This resulted in horizontal edges (vertical length = 0), but did not check for possible crossings. An example of such crossing can be found in figure 7

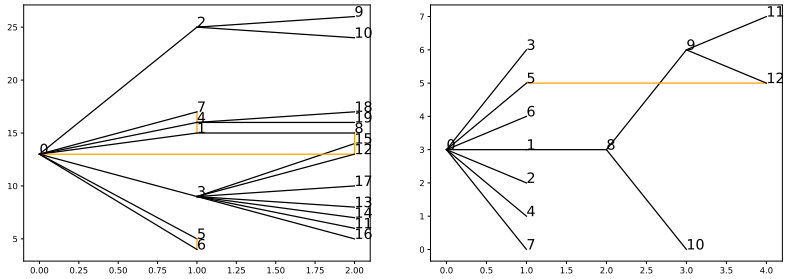

Figure 7: Long hybrid edges are crossing other edges even though the global minimum of the objective function was found.

One possible solution to this issue is only applying this objective to hybrid edges within the same layer. However, we also explored the use of additional objectives to improve the results, but none were found to be effective.

Networks Heuristic

Code of this sub-project can be found in: `05_network_heuristic`.
In the context of full phylogenetic networks, it can be challenging to identify which edges are hybrid edges. As a result, if a node has multiple parents, each of those edges is considered a hybrid edge. This leads to a sparse tree embedding, which reduces the effectiveness of the previously used tree constraints.
Direct optimisation through the minimisation of crossings has been demonstrated to be slow. As an alternative, an attempt was made to approximate the solution using multiple heuristic objectives. A total of six objectives were used: Nodes with the same parent (i.e. siblings) should be positioned as closely as possible. This can be achieved by minimising the pairwise distance between these nodes, resulting in the formation of groups. The parent node of each group should be centred within the group. This is sufficient to solve regular trees and find a positioning without edge crossings. To reduce the clutter of the network, the distance between groups should be maximised. These three objectives are visualised on the left side of figure 8.
To improve the placements of hybrid edges, the Last Common Stable Ancestor (LCSA) is determined for each node with multiple parents. The LCSA is the rightmost node that is an ancestor of all the parents of the node in question. The vertical distance between the node and the LCSA is minimised. This should position the node in between the multiple paths to its LCSA. Additionally, similar to the approach described in the previous section, the vertical length of hybrid edges is minimised. These objectives are visualised on the right-hand side of figure 8.

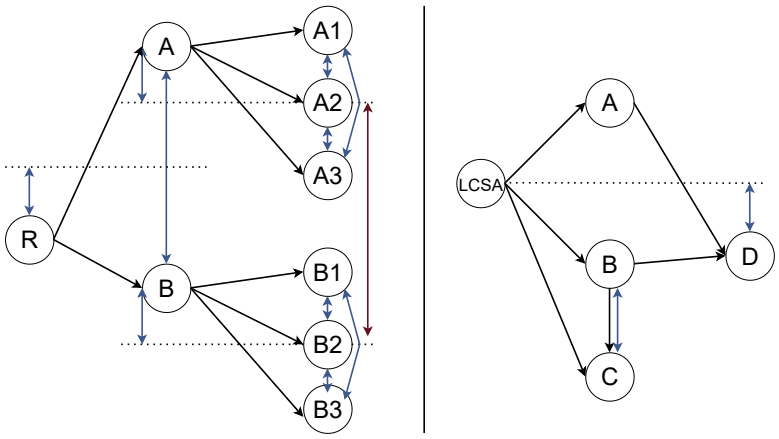

Figure 8: **Left**: Visualisation of the three first objectives. **Right**: Visualisation of two last objectives. The distance indicated by the blue arrows should be minimised. The red arrow maximised.

We found that the resulting trees were often not visually pleasing. In many cases, horizontal edges spanning multiple layers were crossing other edges. Turning the `min_hybrid_length` objective off solved that issue but introduces more crossing in other networks. Even for unbalanced trees with sizes greater than 20, the time limit was frequently reached before a satisfactory solution could be found (visualised in figure 13). This suggests that the chosen objective is not well-suited for producing high-quality layouts of larger phylogenetic networks.

Understanding the optimisation process

The solving process is highly dependent on the solver used. In this case, CPLEX by IBM ILOG is utilised [3].
Several factors can impact the solving process, including the search space, the satisfiability of the constraints, the computational complexity of the objective function, the search strategy and the optimality criterion.
As demonstrated in 10, restricting the search space through the use of upper and lower bounds on the positions of all nodes can reduce the runtime as well as the scatter. This is because the search space is limited to a smaller region. However, if the upper and lower bounds are set too closely, it may become impossible to find a solution.

The given constraints are creating a so-called satisfiability (SAT) problem where the goal is to find a solution that satisfies this set of constraints. When solving a tree, the main constraint was that edges should not cross 1. CPLEX is able to solve those SAT problems efficiently as shown in 5 by using techniques like branch-and-cut [5] [6].

The computational complexity of the objective function can also impact the speed at which a problem can be solved. If the objective function is large and difficult to compute, it slows down the solving process, as it must be recalculated for each potential solution that is explored. Furthermore, if the objective function contains functions that map multiple values to a single result (e.g., the absolute value function or the square function), the solver may need to explore both directions in order to determine the optimal solution. This can further increase the computational complexity. The presence of non-continuous functions, such as the count of crossing edges, also poses a challenge for solvers like CPLEX. As these functions do not have a gradient and will lead to a sudden shift of the objective, gradient-based algorithms such as simplex aren't that effective.

CPLEX employs two primary solving strategies: branch-and-cut and dynamic search [7]. If no specific strategy is specified, CPLEX will automatically choose the most appropriate one based on the characteristics of the problem. For the problems presented in this work, dynamic search is often the preferred method. Following the dynamic search strategy, CPLEX solves a series of continuous sub-problems. To manage these sub-problems efficiently, CPLEX constructs a tree in which each sub-problem is represented as a node. The root of the tree is the continuous relaxation of the original MIQP problem.

If the solution to the relaxation includes one or more fractional variables, CPLEX attempts to identify and add cuts, which are constraints that eliminate areas of the feasible region of the relaxation that contain fractional solutions. CPLEX can generate various local cuts that affect only a subtree, as well as global cuts affecting the whole problem.

If the relaxation solution still includes fractional-valued integer variables after the addition of cuts, CPLEX branches on a fractional variable, creating two new sub-problems with more restrictive bounds on the branching variable. For example, with binary variables, one node will fix the variable at 0, while the other fixes it at 1. If the sub-problems result in a fractional solution, the process is repeated. If the solution is infeasible or all-integer, the branch-and-cut algorithm terminates.

Determining when to stop the search is an important consideration in this MIQP optimisation.
CPLEX terminates under a number of conditions [8]. The most significant of these is the time limit set by the user. Additionally, CPLEX may declare integer optimality and terminate the search if it finds an integer solution and has found the global optimum. However, this scenario is only likely to occur for small or unbalanced trees, as it requires all parts of the search space to be searched. In the default mode, "balance optimality and feasibility," CPLEX uses tactics that aim to find a proven optimal solution quickly for a wide range of problem difficulties [9]. The branching strategy is designed to find high-quality feasible solutions without sacrificing too much time that could be spent proving the optimality of any solutions that have already been found. An alternate mode, "focus on feasibility," puts less emphasis on analysis and proof of optimality. As this mode was not selected, it could be the subject of further research.

It is interesting to examine the process of the solving steps and how the solution evolves over time. In the next figure 9, we visualise some intermediate results of one optimisation attempt.

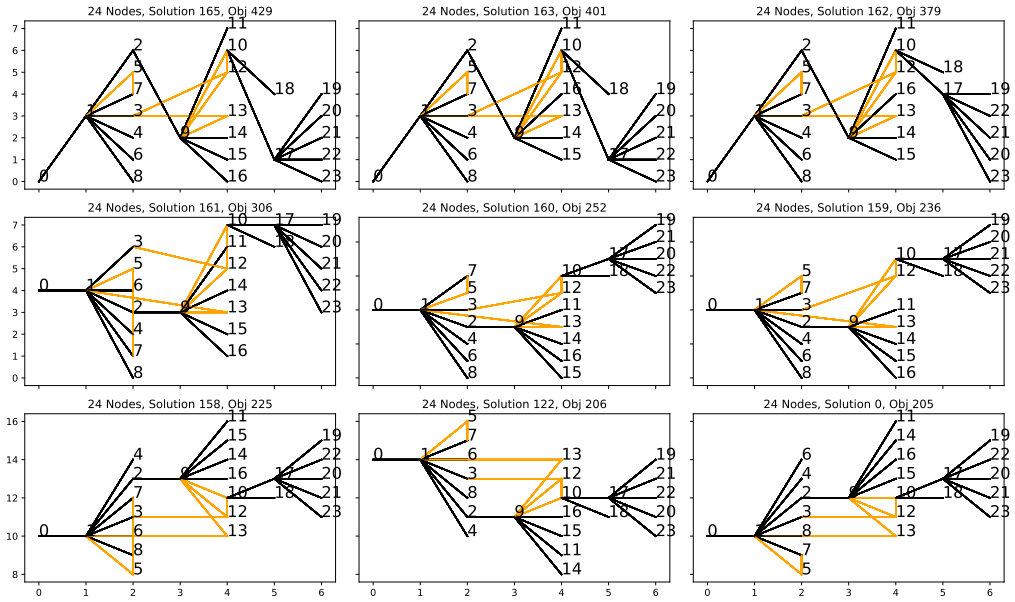

Figure 9: Intermediate solutions while solving a network. The objective score ranges from 429 to 205.

The first solution (top left) satisfies all constraints, but it is far from optimal. For example, it is clear that the objective of having parents centred between their children has not been fully optimised yet, as evidenced by the position of node 10 and its children 17 and 18. However, over the next solutions, it is clear that the children are being positioned more closely to one another.
Overall, the objective function values decrease from 429 for the first solution to 205 for the final solution, indicating that the solution is improving over time. Examining the intermediate results can provide insight into the optimisation process and help to identify areas for further improvement.

Summary and further research

In this work, we explored the use of mixed integer quadratic programming (MIQP) to lay out nodes in phylogenetic trees and networks.
We were able to formulate constraints to describe trees and successfully solve all types of them, including large, balanced, and unbalanced ones. However, as the complexity of the problem increased when applied to networks, we observed diminishing returns. Tree-embedded networks with horizontal hybrids were still solvable and produced high-quality trees. This was achieved by minimising the length of the hybrid edges. More complex structures like trees with hybrid edges spanning across multiple layers became very difficult to solve. In this case, only unbalanced trees produced visually pleasing results, likely due to the quadratic growth of constraints in balanced trees.
We also found that directly minimising the number of crossings was slow. This is due to the high computational cost and step-like properties of the objective. When no information about tree embeddings was available, we had to approximate the qualities of a readable network using multiple objective functions. This approach provided acceptable results for simple or small networks, but for larger ones (size ≥ 20), it was practically infeasible. For most smaller networks, it was possible to find a combination objectives that resulted in good results, but none was found that was applicable to all networks.
In conclusion, we can say that the use of MIQP to layout nodes in phylogenetic trees and networks may not be the most effective approach as already other good heuristics are available. However, there are still many avenues that could be explored, such as the use of different solvers or solving strategies, or the incorporation of alternative objectives that may lead to better results.

List of Abbreviations

Appendix

Code and Data

The complete code used, the complete data produced and every plot for this project can be found at: `https://github.com/not-a-feature/Layouting-Phylogenetic-Networks-using-MIQP`.

System Specifications

The system on which the runtime and speed were tested was a Framework Laptop with the following specifications:

- Device: Framework Laptop DIY Edition
- CPU: Intel i5-1240P (Up to 4.4 GHz, 4+8 cores)
- Memory: 16GB (1 x 16GB) DDR4-3200
- System: Linux Mint 21 Vanessa
- Kernel: 5.15.0-56-generic

Software Specifications

The following packages were used:

- IBM ILOG CPLEX Optimization Studio 22.1.0.0 [3]
- matplotlib 3.5.2 [10]
- numpy 1.23.1 [11]
- python 3.10.4 [12]

A complete conda environment can be found in file `enviroment.yml`

Additional figures

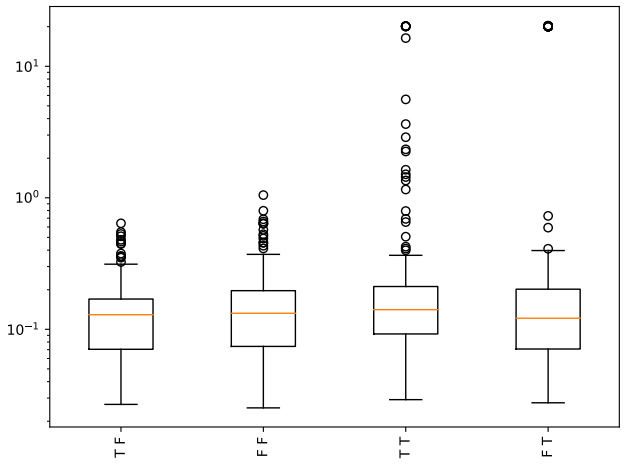

Figure 10: Time to solve trees using different constraints. Each run is repeated with 200 balanced trees with 7 - 15 nodes. First constraint is `max_height_by_nodes`, second one `c_withing_range`. An active constraint is represented with a `T`, an inactive one with a `F`.

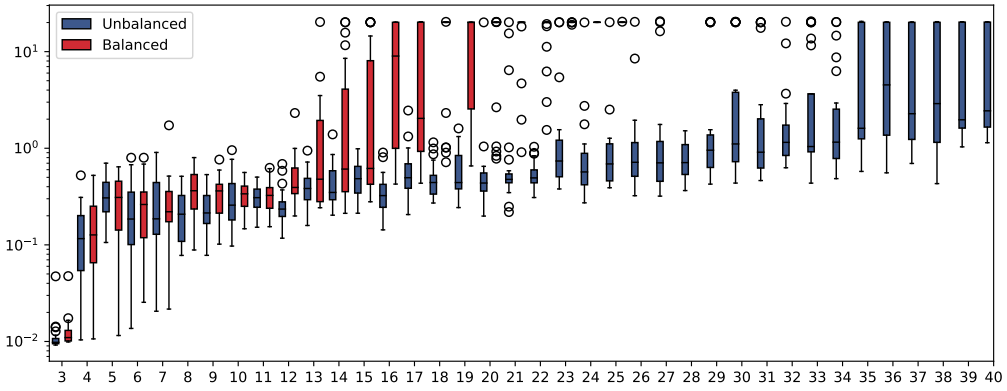

Figure 11: Time to solve trees of size n (x-axis) with hybrid edges inside the same layer. In seconds (y-axis, log-scale).

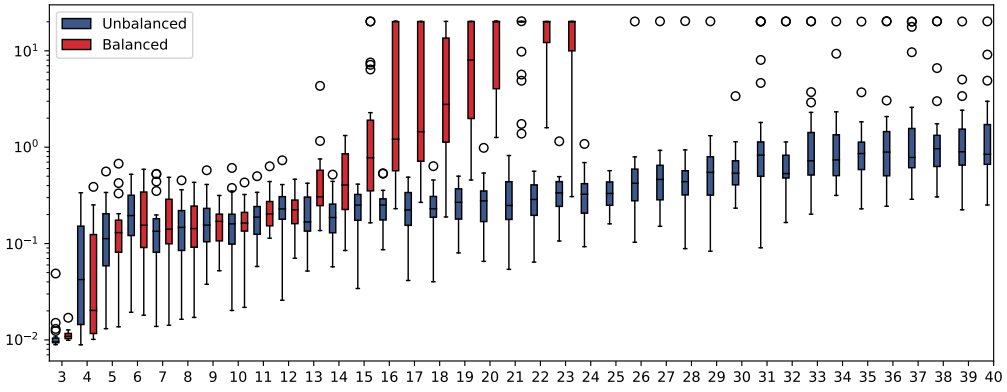

Figure 12: Time to solve trees of size n (x-axis) with hybrid edges spanning over multiple layers. In seconds (y-axis, log-scale).

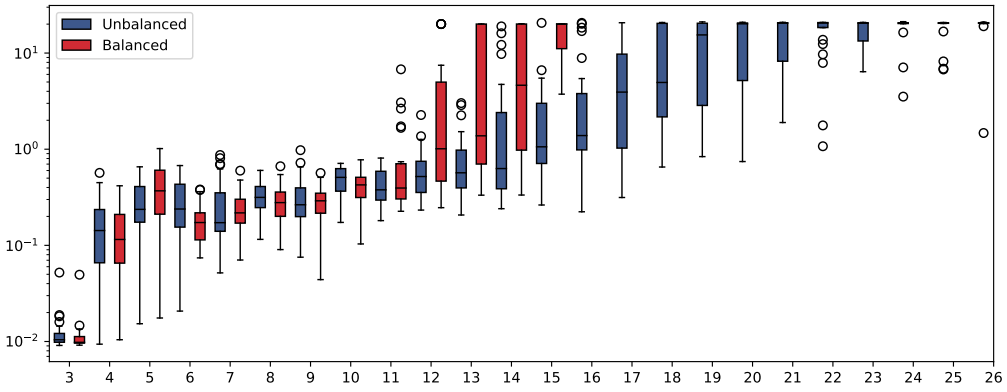

Figure 13: Time to solve networks of size n (x-axis) in seconds (y-axis, log-scale).

References

[1] R. M. Tarawaneh, P. . Keller, and A. . Ebert. "A General Introduction To Graph Visualization Techniques". In: *Visualization of Large and Unstructured Data Sets: Applications in Geospatial Planning, Modeling and Engineering - Proceedings of IRTG 1131 Workshop 2011* (2012), pp. 164–. DOI: 10.4230/oasics.vluds.2011.151.

[2] C. . Bennett, J. . Ryall, L. . Spalteholz, and A. A. Gooch. "The aesthetics of graph visualization". In: *Eurographics* (2007), pp. 57–64. DOI: 10.2312/compaesth/compaesth07/057-064. URL: https://doi.org/10.2312/COMPAESTH/COMPAESTH07/057-064.

[3] IBM. *ILOG CPLEX Optimization Studio - Overview*. 2022. URL: https://www.ibm.com/products/ilog-cplex-optimization-studio.

[4] N. S. Nikolov. "Sugiyama Algorithm". In: *Encyclopedia of Algorithms* (2016), pp. 2162–2166. DOI: 10.1007/978-1-4939-2864-4_649. URL: http://dx.doi.org/10.1007/978-1-4939-2864-4_649.

[5] IBM. *What does CPLEX do?* 2022. URL: https://www.ibm.com/docs/en/icos/20.1.0?topic=cplex-what-does-do.

[6] IBM. *What does CPLEX solve?* 2022. URL: https://www.ibm.com/docs/en/icos/22.1.0?topic=optimizer-what-does-cplex-solve.

[7] IBM. *Branch cut or dynamic search?* 2021. URL: https://www.ibm.com/docs/en/icos/20.1.0?topic=optimizer-branchcut-dynamic-search.

[8] IBM. *Terminating MIP optimization*. 2021. URL: https://www.ibm.com/docs/en/icos/20.1.0?topic=optimizer-terminating-mip-optimization.

[9] IBM. *Tuning performance features of the mixed integer optimizer*. 2021. URL: https://www.ibm.com/docs/en/icos/20.1.0?topic=mip-tuning-performance-features-mixed-integer-optimizer.

[10] T. . A. . Caswell. *matplotlib/matplotlib: REL: v3.5.2*. 2022. URL: https://zenodo.org/record/6513224.

[11] C. R. Harris, K. J. Millman, S. J. van der Walt, R. . Gommers, P. . Virtanen, et al. "Array programming with NumPy". In: *Nature* 585.7825 (2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: http://dx.doi.org/10.1038/s41586-020-2649-2.

[12] *The Python Language Reference — Python 3.10.9 documentation*. URL: https://docs.python.org/3.10/reference/index.html.