Project Report

Wiring Of Glass Fiber Switch Boxes

Kilian Bartz

Date of Submission:	December 23, 2025
Advisor:	Jun.-Prof. Dr. Philipp Kindermann

Universität Trier
Fachbereich IV - Informatik
Algorithmik

Acronyms

BFS	Breadth-First Search
CLI	Command Line Interface
CRP	Channel Routing Problem
DFS	Depth-First Search
GSR	Glass Fiber Switch Box Routing
ILP	Integer Linear Program
TI	text instance

Contents

List of Figures

1 Introduction

I worked on the topic of wiring grid-based glass fiber switch boxes for my research seminar and research project. The research seminar paper is an overview of algorithmic approaches described in literature to tackle similar problems. The research project, on the other hand, includes working implementations for two of the examined approaches to solve practical instances of the problem, as well as this accompanying report / documentation. The first approach solves the problem in an exact fashion (regarding minimal cable length) and uses an Integer Linear Program (ILP) formulation, implemented in Python. The second approach uses a flexible A* search to find good solutions (minimizing the number of bends) very efficiently in many cases, and is implemented in Rust. In terms of efficiency and quality of the solution, they can be compared as seen in Table 1. Due to its exponential runtime, the ILP program can only be applied to medium-size problems at most (see Section 7.3). A* on the other hand provides a good compromise in most cases, where it completely solves the problem, or most nets at least, very quickly.

My recommendation is using the A* solution and only trying the ILP if an optimal solution is required and the problem size permits it.

The entire source code can be found in my GitHub repository[1].

Feature	**A* Program**	**ILP Program**
Solution Quality	Heuristic (Good, but not guaranteed optimal)	Optimal (Guaranteed best solution)
Failing Behavior	Missing only a few nets	Solves problems completely or not at all
Time Complexity	Polynomial	Exponential
Application	Can tackle large problems	Impractical for grid size > 50

Tab. 1: Comparison of A* Algorithm and Integer Linear Programming (ILP)

2 Preliminaries and Problem Definition

The field of wiring problems is concerned with connecting active elements on a planar substrate with conducting wires while adhering to certain restrictions. Since the complexity of wiring problems, together with the integration density for integrated circuits, has grown rapidly over the last decades, designing efficient layouts requires computer assistance in the form of algorithms for automating the placement of components and the routing of connections [Spr88]. Traditionally, for a wiring problem the following inputs are given:

1. a set of m **terminals** $T = \{t_1, \dots, t_m\}$,

[1] `https://github.com/kilianbartz/forschungsproj`

2. a partition N of T, called the set of **nets** with $n := |N|$,
3. a model M,
4. (optionally) a **routing area** R.

A terminal t represents a wire endpoint that is to be connected with every other terminal in its net through wires. Often, the problem is discussed in the context of **two-terminal nets**, but depending on the specific problem, some, or all nets might be multi-terminal nets.

The model M dictates how wires are allowed to be routed and may govern details like minimum wire separation, allowing or prohibiting wires from overlapping or crossing, the number of layers, etc. To solve a wiring problem and produce a **valid wiring** or **layout**, all nets have to be realized while adhering to the model. If the model allows for multiple layers, wires may only switch layers at vertices called **vias**, which connect two layers. Their location can be either pre-defined or placed by the routing algorithm. Without a via, wires that lie on different layers do not interact and are independent of each other.

If an algorithm operates on a **grid** with height h and width w, wire segments and terminals have to be placed on grid lines and grid points, respectively. Horizontal grid lines are often called **tracks** and vertical grid lines are called **columns**. If the routing area is not given, the routing algorithm is supposed to produce a valid wiring with the smallest possible area, and we have an **optimization problem**. Otherwise, if the routing area is given, we are predominantly concerned with the **wirability problem**. However, once wirability is confirmed, a wiring that is optimal with regard to secondary optimization criteria (*e.g.* minimum total wire length) can be sought.

Literature is predominantly concerned with **channel routing** problems as representatives for the optimization problem. Here, all terminals lie on two opposite sides (most commonly the top and bottom side) of a rectangular routing area. A wiring with the least number of columns is searched. On the other hand, **switch boxes** permit the terminals to lie on all four sides of a rectangular routing area and are the usual representatives for the wirability problem.

It is known that the **channel density** $d_{\max}$ is a lower bound for the number of tracks [GZ88] which *area-optimal* channel routers are able to achieve for certain limited subproblems. The channel density is defined as $d_{\max} = \max\{d(c)\}$, where $d(c)$ represents the local density of a vertical cut c (*i.e.*, the number of nets that have to cross the vertical space between columns c and $c+1$). In total, for a given Channel Routing Problem (CRP), a layout with $d_{\max}$ tracks and $n+1$ columns is considered area-optimal [Wag91]. An example for CRPs that can easily be solved optimally is the class of instances where no top and bottom terminals lie on the same column. In this case, $d_{\max}$ tracks and $\mathcal{O}(m + n \log n)$ time are sufficient to route the channel using an *interval graph coloring* algorithm [AB10].

Regarding the model M, most routers assume the **Manhattan model** or the **knock-knee model**. The knock-knee model allows full orthogonal *crossings* of wires in grid points, as well as so called *knock-knees* where two wires may bend by 90 degrees at the same grid point and only touch “knee-to-knee” in this single vertex. This means the routing is only **limited by the edges**, each of which can only be assigned to a single net. In contrast, the Manhattan model only permits orthogonal crossings and does not allow knock-knees, reducing the number of valid wirings. An example of both types of wire interactions is highlighted in Figure 1.

The problem that I explore in this research project, Glass Fiber Switch Box Routing (GSR), is a single-layer switch box routing problem where the routing region is given by the size of its underlying grid (from 5×5 to 1024×1024). The grid confines the terminals (which are allowed to lie on the vertices on the edge of the grid) and the wires (which have to be made up of grid segments). The nets are two-terminal nets. Since the cables to be routed are glass fiber cables, the model permits crossings of cables, but no overlaps. Formally, this means a glass fiber switch box can be defined as a 3-tuple (s, T, N) with

- s the **side length** of the grid in $\{5, \dots, 1024\}$
- T the set of **terminals**; each terminal is represented by coordinates in $\{1, \dots, s\}^2$ and lies on the edge of the switch box
- N the set of **nets** with $N \subseteq \binom{T}{2}$.

For a given glass fiber switch box, the task is to find a **knock-knee wiring** of all nets with the following constraints:

1. The routing should be “as **optimal** as possible”, see below.
2. Every cable needs a “**splice point**,” which is a grid point that must not be crossed by any other cable.

The resulting wiring can be represented as a pair (P, S) with $P = (p_1, \dots, p_n)$ being the list of cable paths where $p_i = (p_{i_1}, \dots, p_{i_k})$ is the sequence of grid lines that connects the terminals of the *i-th* net together and $S = (s_1, \dots, s_n)$ the list of splice points, where s_i represents the splice point of the *i-th* net ($i \in \{1, \dots, n\}$).

Regarding the optimization metric, an obvious choice is the total length of the cables used, i.e.$\sum_{i=1}^{n} |p_i|$. However, when looking at the practical application, other metrics, such as the number of bends or the length of the longest cable (to minimize latency), may be more relevant.

An example for a solved glass fiber switch box can be found in Figure 1.

Notice Both of the programs presented in the following only try to solve the wiring without regard for splice points. As I have not found any literature concerned with strategies for optimal splice point placement, the only sensible algorithmic way to implement splice point placement would be to do so greedily. Because of this, I have opted to leave them out of my programs and leave splice point placement to the user’s discretion.

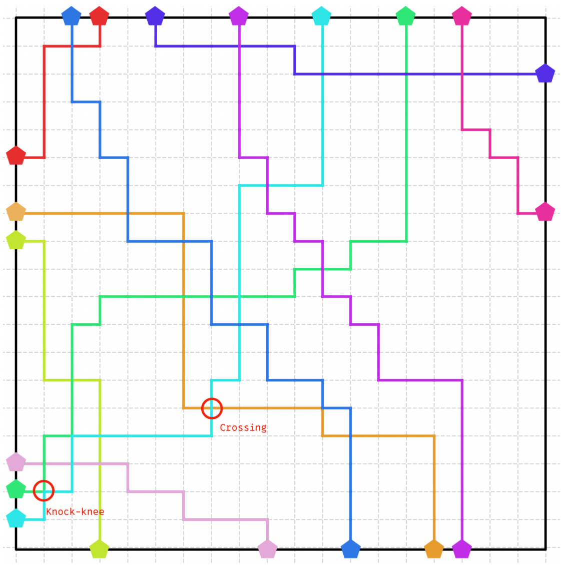

Fig. 1: Example of a glass fiber switchbox wiring instance, solved with minimal cable length. The terminals (represented by pentagons) abut the switch box from all four sides. Examples for crossings and knock-knees are marked with circles.

3 Input/Output of my Programs — Practical Instances

Both of my programs are terminal-based and work on `stdin / stdout`. For reproducibility, they should be launched with the contents of a text instance (TI), *i.e.*, a textfile describing the instance, piped to them like so: `./program < instance1.txt`. Both programs offer a Command Line Interface (CLI) which provides a help text, accessible through `./program -help`. The ILP program supports the `-p` (plot) flag to visualize the solution. More on this can be found in Section 6.

As explained in Section 2, an instance is defined by the size s of the grid, the set of terminals and the nets which should be routed. Every terminal is identified by a terminal number, starting at "1" in the top left and ending at $s \times s$ at the bottom right. For practical reasons (see below), the four corners of the grid are not allowed as terminals. An illustration and further rules can be found in Figure 2 and its caption. A TI to describe an instance should be constructed like this:

- First line: A single integer, indicating the side length s of the quadratic grid
- Each of the following lines describes a two-terminal net. The first integer is the terminal number of the starting terminal of the net; the second integer (separated by a single space) is the terminal number of the target terminal of the net.

Which terminal should be the starting and the target terminal of a net is an arbitrary decision. However, it may influence the solution found by the A* algorithm and dictates the order in which cable paths are described in the solution. An example for a TI can be found in Listing 1.

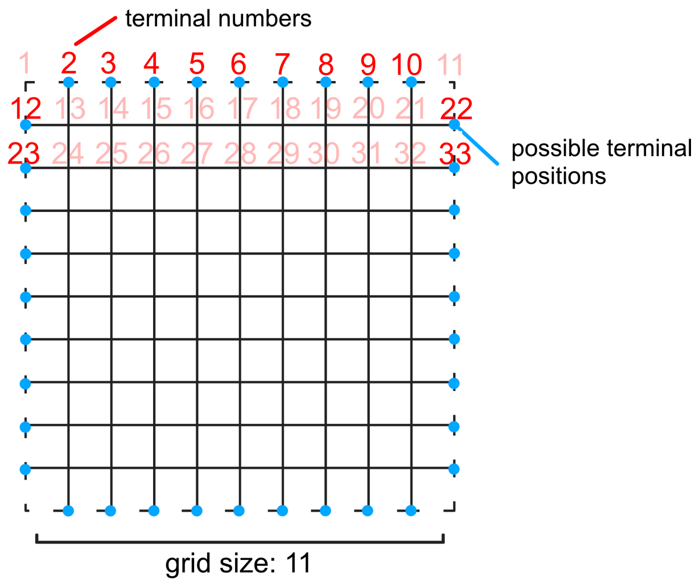

Fig. 2: Naming of important grid properties. Possible terminal positions are highlighted in blue, associated terminal numbers are shown in red. Pale red numbers indicate that this terminal position is only part of the numbering; placing a terminal at this position would be illegal. Only terminals from different edges of the grid are allowed to be connected to each other,*e.g.* forming a net $\{2, 5\}$ would be illegal. Each terminal only has a single possible first wire segment, which runs vertically if the terminal lies on a horizontal edge, or horizontally otherwise. Because of this, the four corners are also illegal terminal positions.

3.1 Output

The output of both programs is a JSON, including the following attributes:

- **type** (string). Can be “astar” or “ilp”.
- **solved** (boolean). Indicates whether the instance was solved completely, or not.
- **paths** (object). Describes the path that the algorithm found for every net as an array of edges.
- **time** (float). Total time needed to try and solve the problem.
- **grid_size** (int). Grid size of the problem.
- **missing** (array). Lists the nets which could not be solved.

Paths Object The path description object uses the net as a key (written as a pair of terminal numbers) and assigns a list of edges to it. Every edge is oriented from the starting terminal towards the target terminal. The list of edges starts with the edge

```
5
3 16
11 20
15 23
```

Listing 1: A simple TI for an instance of the wiring problem on a grid with size 5 and three nets.

```
{"type":"astar",
"solved":true,
"paths":{
   "(3, 16)":[[[1,3],[0,3]],[[1,2],[1,3]],[[2,2],
   [1,2]],[[2,1],[2,2]],[[2,0],[2,1]]],
   "(15, 23)":[[[2,3],[2,4]],[[2,2],[2,3]],
   [[3,2],[2,2]],[[4,2],[3,2]]],
   "(11, 20)":[[[3,3],[4,3]],[[3,2],[3,3]],[[3,1],
   [3,2]],[[2,1],[3,1]],[[1,1],[2,1]],[[1,2],[1,1]],[[0,2],[1,2]]]
   },
"time":0.000046539,"grid_size":5,"missing": []}
```

Listing 2: Example output of my A* program. The input was the TI from Listing 1.

incident to the target terminal and ends with the edge incident to the starting terminal. The order of edges for the ILP program is arbitrary. An example output can be found in Listing 2.

4 Exact Solution using an Integer Linear Program

The problem of routing two-terminal nets in the knock-knee model can be interpreted as a multicommodity flow problem [ZP93]. Each net has a starting terminal which wants to transport the net's commodity to the target terminal using grid edges. The single commodity flow problem can be efficiently solved in polynomial time using network flow algorithms [GTT89] and has been extensively discussed in literature. On the other hand, the multicommodity variant with integer flows has been shown to be NP-complete [EIS76]. Nonetheless, it can be modelled as an ILP in an intuitive way and then be inputted in a linear program solver. Nowadays, several different solvers are available, from open source (*e.g.* GNU Linear Programming Kit[2]) to commercial software (*e.g.* Gurobi[3], IBM CPLEX[4]).

[2]https://www.gnu.org/software/glpk/glpk.html
[3]https://www.gurobi.com
[4]https://www.ibm.com/products/ilog-cplex-optimization-studio

4.1 Advantages of the ILP Approach

Simplicity My ILP implementation separates the model, including the optimization criterion, from the solver. Additionally, the model and the optimization criterion are very easy to understand which enables simple and fast adjustments in case the solution has to be adapted to the problem.

Optimality As the ILP searches for an exact solution, it can be guaranteed that a found solution is optimal, and that an instance is unsolvable under the model's restrictions if no solution could be found.

Maintenance My ILP approach only describes the model and the instance to solve, the rest is handled by the solver. This simplifies maintenance greatly. Added to that, especially commercial solvers are actively maintained and offer support in case of a problem. This also means that fewer bugs have to be expected and many optimizations, such as computation on multiple CPU cores or even GPUs, can be relied on.

4.2 Structure of my ILP Program

My ILP program uses Gurobi and therefore also relies on the Gurobi Python bindings. After parsing the CLI arguments and the instance properties (*i.e.*, grid size and the nets to be routed), the vertices and edges of the underlying grid are generated. The **variables** for the ILP can be found in the flow dictionary. A variable `flow[k, i, j]` represents the binary decision whether the k-th net uses the edge (i, j) or not, and is created for every possible pair (n, e) with $n \in N$ and e representing a grid edge.

After that, the ILP **constraints** are created, demanding an outflow of exactly 1 at the starting terminal and an inflow of exactly 1 at the target terminal. All other grid vertices have to obey flow conservation. Additionally, the total flow across a grid edge is capped to 1 to ensure that every edge can be used by one net at most. The objective is set to minimize the total flow, *i.e.*, minimizing the number of total cable segments, which translates to minimizing cable length.

After the ILP is solved, the program iterates over all nets and collects the edges used and outputs the JSON.

4.3 Modifications of the Optimization Criterion

Instead of minimizing the cable length, depending on the concrete problem conditions, it might be necessary to optimize another metric. As seen in Section 7.1, this may even help the algorithms to find a viable solution in general.

An optimization criterion, which is important in many wiring scenarios, is **signal skew** s. It can be calculated as $s = |T_{\max} - T_{\min}|$ with $T_{\max}$ and $T_{\min}$ representing the longest and shortest path delays, respectively. For timing sensitive applications, s must not exceed some threshold. Therefore, it can make sense to incorporate s as a minimization criterion into the ILP. Replacing the minimal cable length with minimal signal skew is a trivial task and could be achieved using the following piece of code:

```
flows_per_net = [sum(flow[k, i, j]) for k in range(len(commodities))]
min_flow = min(flows_per_net)
max_flow = max(flows_per_net)
skew = max_flow - min_flow
model.setObjective(skew, GRB.MINIMIZE)
```

Another metric sensible for minimization is the total **number of bends** in the wiring. Adding this to my ILP formulation could be more challenging. An idea I want to sketch for this scenario would be to add new binary variables `bend[j, k]`, indicating if net k forms a bend at vertex j and to minimize the total bends in analogy to the total flow. In order to correctly set these variables through constraints, the following helper variables can be used:

- `horizontal[j, k]` to indicate that the k-th net takes a horizontal edge incident to vertex j
- `vertical[j, k]` to indicate that the k-th net takes a vertical edge incident to vertex j

`bend[j, k]` is now 1 iff both `horizontal[j, k]` and `vertical[j, k]` are 1. It should be noted, however, apart from the added conceptual complexity, adding more variables also makes the problem more complex. This results in longer practical solving times, even though the asymptotical runtime does not change.

5 Heuristic Solution using A*-Search

Some of the simplest and most well known approaches for wiring problems are Maze Routing algorithms, which go back to Lee's original publication [Lee61]. The basic idea is routing each net separately by performing a Breadth-First Search (BFS) (with unit costs) from the starting towards the target terminal. If the target terminal has been reached, the path can be retraced and blocked for all further nets. Using a BFS results in a shortest path for this net; however, the solution as a whole is not guaranteed to be optimal, because every net is considered in isolation.

In practice, exploring the grid using a BFS is not very efficient since the search perimeter expands in concentric circles, spending a lot of time searching in irrelevant regions, which is illustrated in Figure 3. Because of this, there have been several modifications to this basic algorithm, *e.g.* by Soukup [Sou78] who replaced the BFS with a Depth-First Search (DFS) or Hadlock [Had75] who used an A*-search.

When using BFS, the time a grid cell has to wait before it is explored is solely governed by its distance to the starting terminal. In a greedy best-first search, the order of grid cells explored is solely governed by the predicted distance to the target terminal. The A*-search combines both ideas by exploring grid cells x in ascending order of the value $f(x) = g(x) + h(x)$ where g represents the real distance from the starting terminal to the grid cell along the exploration path and h represents a heuristic prediction of the distance to the target terminal.

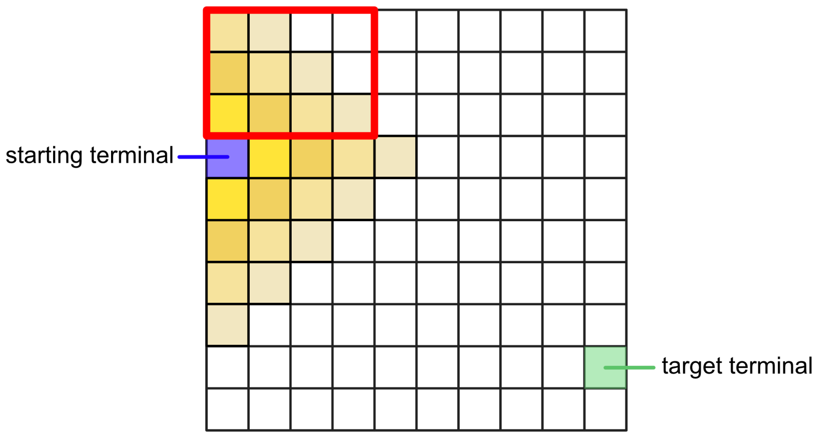

Fig. 3: Illustration of the propagation of the search perimeter of BFS in concentric circles. In general, one would like to prevent searching in the area marked in red, as the distance to the target terminal only increases in each iteration and the optimal path towards the target cannot run through this area.

5.1 Advantages of A*-Search

Efficiency In practice, A*-search is very efficient, exploring only a limited number of irrelevant grid cells, and can still provide optimal solutions, as long as the heuristic is admissible (optimistic[5]).

Even though the asymptotical run time remains $\mathcal{O}(s^2 \times n)$ with s the grid size and n the number of nets, this is much more favorable compared to the exponential run time of the ILP and enables the capabilities to solve even big grids as can be seen in Section 7.3.

Utility and Simplicity Apart from its optimality (as with BFS this applies only to the routing of a single net, not the whole solution), the use of the heuristic h provides a lot of utility, as it can be used to encode further design criteria by rewarding desirable behavior through a discounted value of h. In contrast to the ILP program, these desirable behaviors can be assessed locally instead of globally, making certain optimization criteria like (heuristic) bend minimization a lot easier (see Section 5.4).

5.2 Structure of my A*-Program

The A*-program is implemented in Rust, which significantly improved the runtime compared to my early Java implementation. The code is distributed over three locations:

1. `main.rs`. This file is predominately concerned with configuring the CLI, parsing the stdin inputs, creating the instance and calling `process_pq` to run the A*-search. If the initial search could not solve the problem entirely, missing nets and

[5]This means that the heuristic function provides a lower bound and never overestimates the distance to the goal.

critical edges are identified and a solution optimization strategy based on flipping bends (see Section 5.5) is called.

2. `cabling::base`. This module contains the basic data structures for Terminals, Edges and Nets, as well as a helper function to compute the Manhattan Distance.

3. `cabling::astar`. This module contains data structures for the A*-search, such as GridPoint (representing an explored grid cell, specific to a net), a PriorityQueue (based on a MinHeap), the CablingProblem and the Solution. Beside important methods on these data structures, the functions `process_pq` for performing a step of the A*-search and functions for identifying and solving problems through flips are found here.

5.3 Process Flow During the Initial A*-Search

As part of the initialization, the initial GridPoints, corresponding to the starting terminal, for every net are created and added to the PriorityQueue q. At the start of each iteration of `process_pq`, the GridPoint g with the lowest f-value is extracted. It is then checked whether the grid should be further explored from g, or not. Reasons to stop exploration are:

- Another net has finished in the meantime and claimed the edge needed to reach g from its predecessor.
- g's net has already been finished (GridPoints are lazily skipped instead of removing them actively when a net is finished).
- g was generated during a previous exploration phase (smaller net version) and is therefore invalid (see next point).
- g is the target terminal, finishing the net. In this case, the path p to g is retraced and all edges within p are tested for legality, *i.e.*, that they have not been assigned to finished net in the meantime. If all edges are legal, the path is stored in the solution and every edge within p is blocked for other nets. Else, the path obtained is illegal and g's net has to be routed from scratch. All of the GridPoints for g's net in q are invalidated through incrementing the net version of g's net and the exploration of the net has to restart at the starting terminal.

If none of the above conditions hold, g's legal neighboring GridPoints are computed and added to the priority queue.

The A*-search terminates once all nets have been solved or q is empty. In the first case, a complete solution has been found and can be outputted. Otherwise, the initial A*-search was not able to solve the entire cabling problem. This means that due to the order of completion, at least one net was blocked by another net and could not be routed. In this case the program tries to solve the remaining nets by flipping bends of completed nets. This is detailed in Section 5.5.

5.4 First Optimization: Bend Minimization

There are certain situations where trying to minimize the number of bends can help with finding a complete solution. This is because solely minimizing the cable length can lead to stair-like patterns with many bends, which leads to a local bottleneck, illustrated in Figure 4. In order to implement (heuristic) bend minimization, I extended the heuristic h used in the A*-search. Concretely, a constant reward of 10 is discounted from h when the edge direction from the parent GridPoint matches the edge direction between the parent and its predecessor.

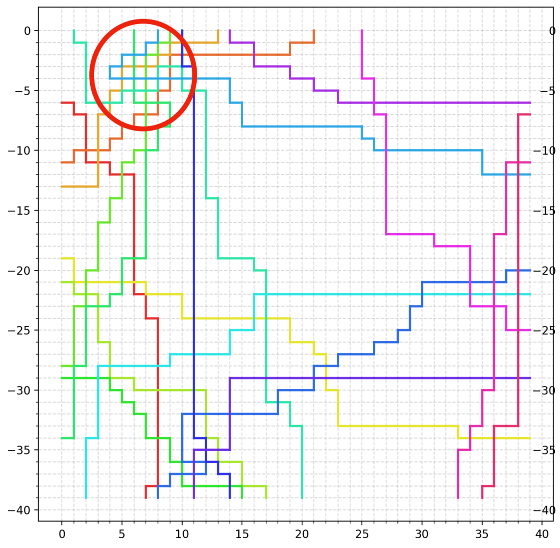

Fig. 4: When optimizing solely for minimum cable length, stair-like patterns (highlighted by the red circle) are often produced. This produces a local bottleneck, which would prevent routing of new nets from terminal number 5 to 10.

5.5 Additional Solution Optimization Strategies

When using a sequential algorithm, it cannot be prevented that some nets might not be routed due to blocking by previous nets. To deal with this, a *clean-up phase*, which tries to route the remaining (blocked) nets, can follow the routing phase. Automatic clean-up approaches, which rearrange or remove existing wiring, fall under this category.

One option for automatic wiring clean-up is **Rip-up and Reroute** [DK82], where previous nets are removed to create room for a new connection path. The idea is that this enables routing of the most problematic nets and a new path can be found for the removed nets. Advantages of this strategy are the guaranteed improved routability for the previously blocked nets and the simple implementation. One has to simply find and remove the finished nets in conflict with the missing nets and then delete them from the solution. However, if the deleted nets cannot be routed afterwards, this might result in a poorer result compared to the original solution.

Alternatively, there are **Shove Aside** techniques, which move existing wire segments into adjacent routing spaces [DK82] in order to clean up local bottlenecks. This leaves all previously finished nets intact, making it more stable, but does not guarantee that the missing nets can be routed. As early experiments during implementation indicated that Shove Aside leads to more solved instances (tested grid size 5, with 2 to 3 nets) than Rip-up and Reroute, I decided to implement it in my A*-program. Concretely, I implemented flipping of bends to free edges for previously blocked nets.

General Bend Flipping Procedure In Figure 5, a simple example for a blocked net, which can be solved by flipping the finished net, can be seen. To perform such a flip, I

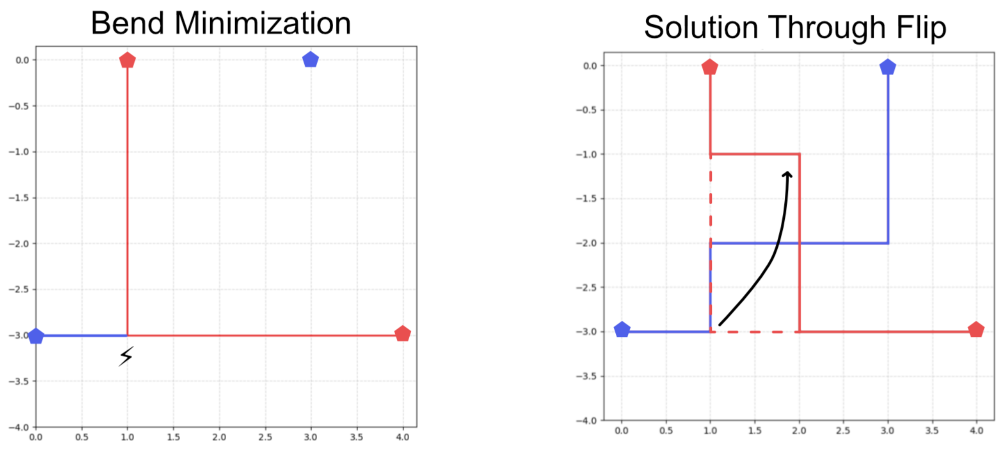

Fig. 5: Using bend minimization (left), the most optimal path for the red net makes exactly one bend and therefore blocks the blue net. By flipping this bend upwards (right), which introduces a new bend, enough space can be freed to enable routing the blue net.

employ the following strategy:

1. Select one net $m \in M$, with M the set of missing nets.
2. Select one net $b \in S_N \backslash F$, which blocks m and the associated conflicting edge e, with S_N the set of finished nets, F the set of filtered nets (see below) and e an edge, which lies on both b's path and a possible path for m. To find conflicting nets, a further A*-search from the starting terminal of m is performed with the previously found solution as initial state. e is selected to be the edge which lies on a path for m and is closest to b's target terminal [6] (`find_conflicting_net`).

[6]This works better than selecting an edge closer to the starting terminal since it increases the chance of a bend between e and the starting terminal.

3. Check if b has a bend. If not, b will not be modified and is added to F for the next iteration and the algorithm restarts at step 1.

4. Select the first bend $x = (v, w)$ (`find_bend`) between e and the starting terminal of b. Also, compute the list of edges between e and v on b's reverse path, called E_r. Steps 2 to 4 are explained in Figure 6.

5. Flip x if this is legal (`fix_bend`). If the flip is illegal, b is added to F and the algorithm restarts at step 1. Otherwise, m can be extended by at least one grid cell in the next A*-search.

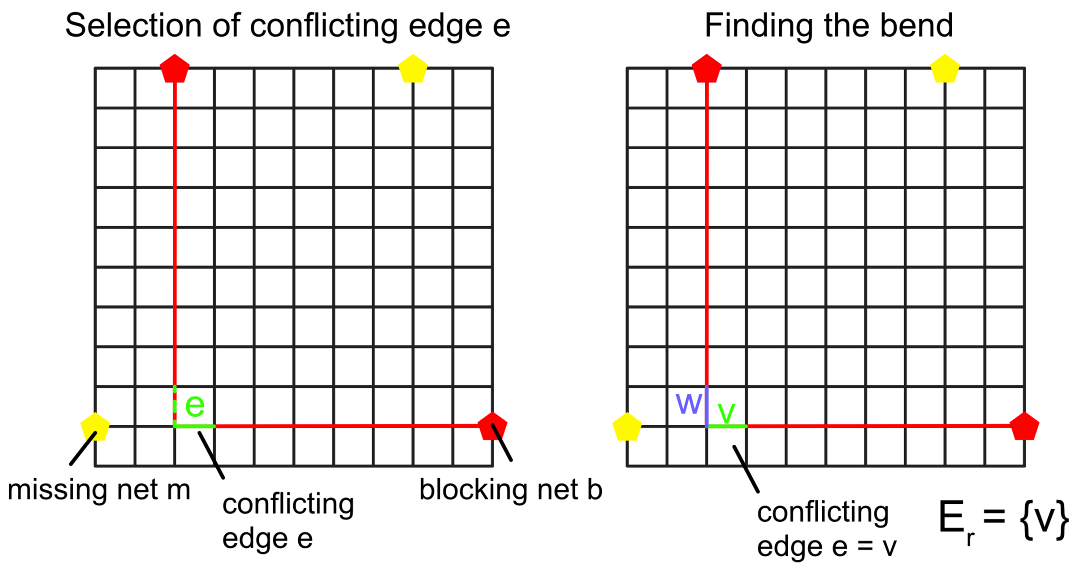

Fig. 6: By performing an A*-search from the starting terminal of the yellow net (on the left), the fully-colored and the dotted green edge are candidates for the conflicting edge e. The fully-colored edge is selected, because it lies closer to the target terminal (on the right) of the red net. The bend $x = (v, w)$ can be found by walking through the reversed red net (*i.e.*, sorted from target to the starting terminal), starting from e until the first edge with a different direction is encountered. This edge is then called w. v is the edge between e and w that is adjacent to w. In this case $e = v$. Every edge between e and v (both inclusive) are collected in E_r.

In order to perform the actual bend flip itself, I explored two different methods, static and dynamic flips, detailed below. The static variant is faster, but helps in less cases than the dynamic version. This enables another trade-off between speed and quality of the solution. To combine both advantages, the "dynamic" strategy of my program uses a hybrid approach, trying to perform the flip statically if possible and falling back to the dynamic algorithm otherwise.

Static Flips A bend is flipped statically by first relocating w to w' and then every $e_r \in E_r$. This process is explained in Figure 7. If one of these relocations results in an edge incident to a terminal, the entire flip is illegal.

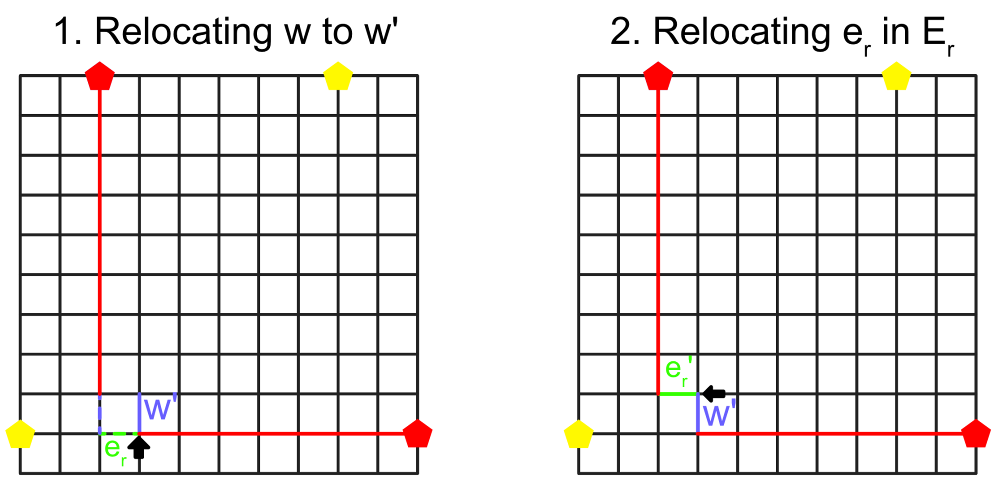

Fig. 7: Hardcoded flip procedure. First, edge w is relocated to the ending coordinate of e (marked by an arrow). Then in step 2, every $e_r \in E_r$ is relocated to the starting coordinate of w' (again marked by an arrow). In this case, even after flipping the bend in the red net, the yellow net still cannot be completed and a second flip has to be performed.

Dynamic Flips A bend can be flipped dynamically by first relocating w to w' exactly like in the static case. In order to connect the grid vertex, which was previously incident to w, and the grid vertex, which is now incident to w', a new A*-search is performed (`route_new_subnet`). A nice side effect of this is that only the legality of relocating w has to be explicitly tested and every other edge is governed by the already implemented rules of the A*-search.

Implementation Notes Here, the selection of the candidate nets for clean-up is somewhat arbitrary, since it depends on the order in which the program iterates through the solved nets. If there are two nets b_1 and b_2 blocking a net m, whether b_1 or b_2 is selected to be flipped depends only on the arbitrary order of nets in the `solution.paths` data structure. Using a heuristic to select a net to shove aside instead may improve the number of instances that can be fully solved.

That said, selecting nets for rip-up and reroute is a lot more precarious, as removing too many nets results in a greater re-routing effort and can cause routers to fail on nets where feasible routes were found earlier. This also makes rip-up and reroute difficult to assess in terms of computational effort [DS81] and might also be the reason, why shove aside performed better than rip-up and reroute in my initial experiments.

Another important question when implementing solution optimization strategies is how many times the strategy should be applied, before conceding that the instance is (probably) not solvable with the applied algorithm. In my implementation, I chose to try a maximum of $n + c$ iterations, with n the amount of nets, and c a user-configurable parameter. The parameter should include a prediction of the length of the missing net(s), because it might happen that through every flip, the missing net only reaches one grid vertex further, if at all. When experimenting on instances of grid size 5 with 2 to 3 nets, more than 2 iterations did not result in any improvement. In general, setting the number of tries to a fixed value will, most likely, result in unsolved instances that would be solvable, given more iterations. However, in order to control the worst-case runtime, a hard limit has to be set.

6 Extras

Included in my project are several useful tools that I used during development of the programs and as part of my experiments. Their function is described in this section.

6.1 Plot Solution

Both of my programs produce a JSON output that includes the produced cabling. However, in many cases visualizations provide much faster means to inspect and verify a solution. Because of this, I included the `plot_solution.py` script. It reads from *stdin* and expects a JSON-Output from one of my programs as input (verbose output works too, as long as the JSON string is at the end of it). The easiest way to use it is to pipe the output from the programs directly to it, like so:
`./program < instance1.txt | uv run plot_solution.py`. This script opens a Matplotlib window displaying a solution, similar to Figure 5.

6.2 Random Instance Generator

To create single test instances, the `gen_instance.py` script is included. It takes the grid size s and the number of nets n as CLI arguments and outputs a TI that can be piped to a file or to a solver program.

6.3 Algorithm Tester

To run my experiments in Section 7, I needed to test my programs' outputs on every possible instance with a specified grid size and number of nets. I did this using the `algo_tester` program included in this project. It provides two commands: The first command ("generated") is for generating all possible instances within a provided specification (determined by grid size, minimum and maximum number of nets) and runs them through the provided command. The second command ("folder") iterates over all TIs within the provided directory and runs them through the given command. To speed up the execution, the workload is automatically parallelized over all available CPU cores using *rayon*[7] for both commands.

The algorithm tester program provides a CLI with a help page (`./algo-tester --help`) which can be consulted for more information.

6.4 Results Evaluation

The JSONL output files can be evaluated and compared using the Jupyter Notebooks found in `final_val`.

7 Experiments

To evaluate the behavior of my programs over a wide spectrum of test instances, I decided to test them on all legal instances within these limits:

1. Corpus *small*: Grid size $s = 5$, minimum number of nets $n_{\text{min}} = 2$, maximum number of nets $n_{\text{max}} = 3$. Total number of instances: 8703. For A*, these were tested with a maximum of 3 bend flipping iterations (`-i 3`).
2. Corpus *large*: Grid size $s = 6$, minimum number of nets $n_{\text{min}} = 2$, maximum number of nets $n_{\text{max}} = 4$. Total number of instances: 628416. For A*, these were tested with a maximum of 10 bend flipping iterations (`-i 10`).

Concretely, I wanted to measure how effective my programs and the implemented optimizations are in tackling the test instances,*i.e.*, how many of the solvable instances were being fully solved and how good the results were in case the solution was incomplete. The ILP with its exact solution provides the baseline for these experiments. Secondly, I wanted to investigate the efficiency of my programs, especially regarding the trade-off between runtime and effectiveness.

Additionally, for the tests in Section 7.2, I created a dataset of 1700 random instances with grid size 20 and 3 to 19 nets (100 instances per number of nets) with the help of the random instance generator to measure the impact of increased problem density.

To measure the impact of increasing the grid size in Section 7.3, I performed further experiments on 5 randomly generated instances with grid sizes $20, 30, 40, 50, 60$ and 3 nets each.

The experiments were run using the following configuration:

- CPU: AMD Ryzen 7700X (8 Cores / 16 Threads)
- RAM: 32GB
- OS: Arch Linux (Cachy OS)
- Build based on commit #b9b4c8c

[7]`https://docs.rs/rayon/latest/rayon/`

The times reported were taken from the output JSONs of the respective programs and aggregated if necessary. Because the results were obtained through the Algorithm Tester (Section 6.3), they were produced under full CPU load conditions. The execution of the experiment took place in a normal Linux environment (no prioritization of benchmark, no changes to the scheduler etc.), meaning that especially the short durations measured for A* were certainly influenced by external factors, making times difficult to compare. However, the results should be sufficient for broad comparisons and represent a real scenario pretty well.

7.1 Solution Optimization Strategies

The results of testing my A* program on the small instances corpus can be seen in Table 2.

Tab. 2: Comparison of my solution optimization strategies. Failed instances are those which are solvable through ILP, but were not fully solved through the strategy. Times for the base algorithm and Bend Minimization were not measured.

Method	Solved	Failed instances (%)	Time (s)
A* (base)	7648	1055 (12.0%)	–
Bend Minimization	8150	553 (6.4%)	–
Bend Flips (static)	8465	238 (2.7%)	0.372
Bend Flips (dynamic)	8469	234 (2.7%)	0.380
Bend Flips (hybrid)	8469	234 (2.7%)	0.374

Implementing bend minimization roughly reduced the number of failed instances by a factor 2 over the base algorithm, while bend flipping improved the bend minimization results by another factor 2. When comparing the bend flipping implementation, we can see that the static method performs the fastest, but cannot solve every instance solvable via the dynamic version. This leaves the hybrid approach as the best strategy for flipping bends, as it combines the effectiveness of dynamic bend flipping with the efficiency of the static version.

The ILP as reference was able to solve all 7855 solvable instances (0 failed instances), but took 136.7s compared to roughly 0.4s for the A* variants (ca. 340 times increase).

Testing on the Large Corpus On a considerably larger corpus, the behavior from Section 7.1 can be seen at a larger scale. The results of static bend flipping can be found in Figure 8, the results of dynamic bend flipping in Figure 9. The A* program is able to solve about 84% of all instances. About 40% of the unsolved instances are unsolvable, meaning that A* only fails on about 10% of solvable instances. The difference between static and dynamic bend flipping is quite small with 1129 instances and 1.14s. Hybrid bend flipping is again the best variant and solves as many instances as dynamic flipping while staying very close to the runtime of static flipping. Compared to A*, running the ILP took about 716 times longer.

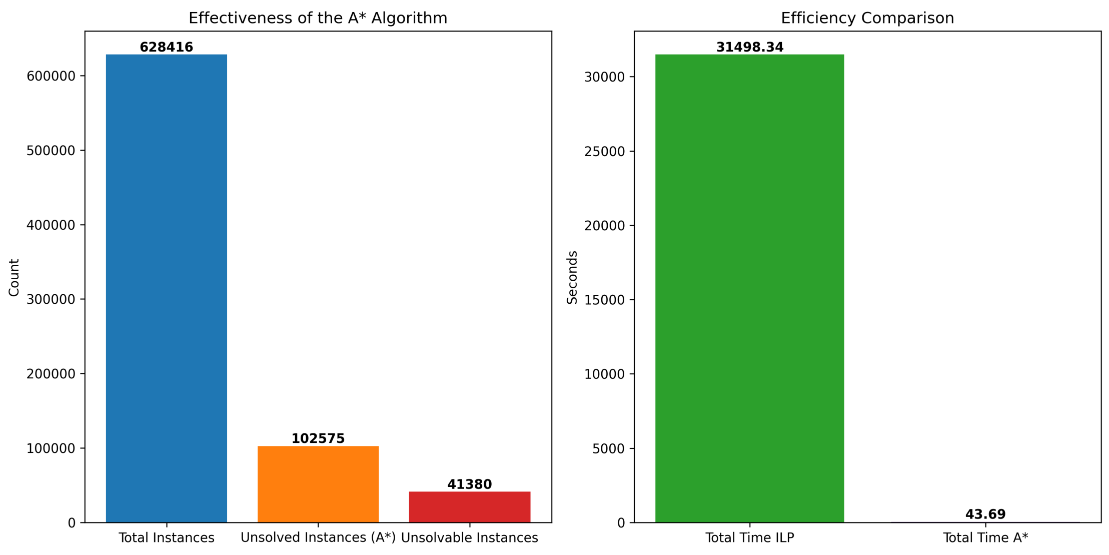

Fig. 8: Effectiveness and efficiency of dynamic bend flipping on the large corpus.

7.2 Density Tests

To better understand the behavior of the ILP and A* programs on more or less dense problem instances, I ran an additional test where the grid size was constant at 20 and the number of nets was increased from 3 to 19. The results are visualized in Figure 10 and Figure 11.

The ILP is able to solve all of these instances and exhibits linear time scaling in regards to the number of nets, as expected.

When looking at the A* results, it can clearly be seen that the solution quality diminishes with increased problem density, leading to fewer completely solved instances and more missing nets. Additionally, the scaling behavior is quite unpredictable. Even though an upwards trend is discernable, there are many outliers in the graph. One reason for this is certainly that external load on the test system has a far bigger influence on the small time quantities measured for A*, compared to the ILP times. However, it also suggests that the information of grid size and the number of nets do not suffice to reliably predict the A* runtime and instead, factors like the order in which nets are processed and other nuances have a large impact as well.

That said, the average runtime for A* reaches a maximum of about 11 **milliseconds** while the ILP reaches a maximum of 206.14 **seconds** (increase by a factor of 18740). Furthermore, there seems to be a threshold up to which A* is able to solve instances pretty well. For a grid size of 20 this seems to be 12 nets, after which the average quality of the solution noticeably drops. However, even though only half of the dense instance with 19 nets can be completely solved using A*, many of the failed instances only miss few nets, which should minimize the effort when manually altering these instances. Concretely, over the dense instances with 19 nets, 70 out of 1900 nets were not able to be routed (solving rate of 96% per net), leading to 0.7 missing nets on average.

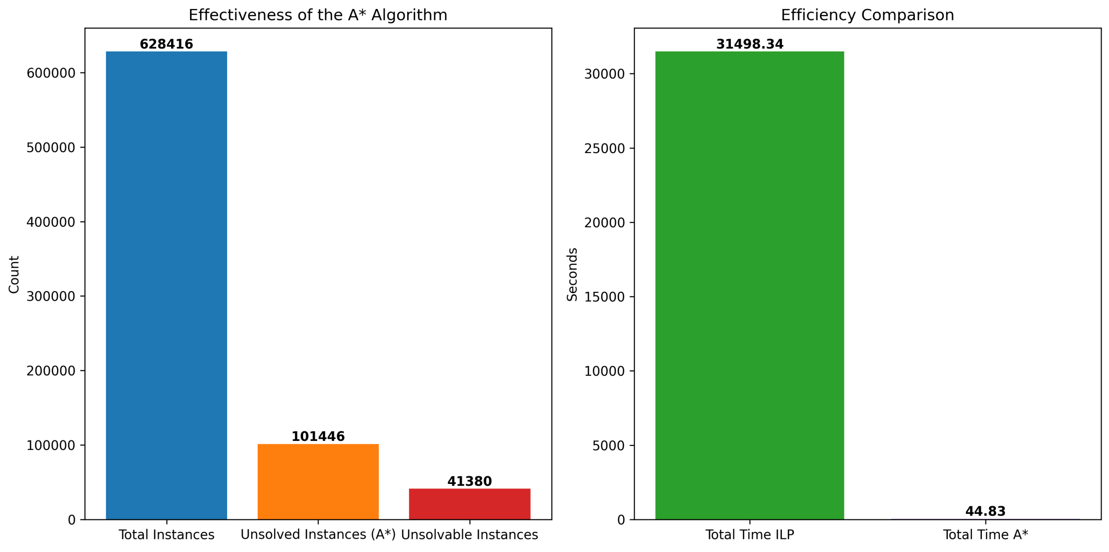

Fig. 9: Effectiveness and efficiency of static bend flipping on the large corpus.

7.3 Scaling With Grid Size

To assess how both programs deal with increased grid size, I measured the time needed to solve the five scaling behavior test cases mentioned above. The results can be seen in Figure 12. The instances are all simple enough that both programs are able to solve all of them completely. However, while increasing the grid size leads to an increase in runtime by a factor of ca. 2 for A* (from smallest to largest measurement), the ILP time increases by a factor of ca. 793. This problematic scaling is further compounded by the fact that increasing the grid size without increasing the number of nets only makes an instance *less* complex, as it allows the routing algorithm more freedom.

Two notes on the results:

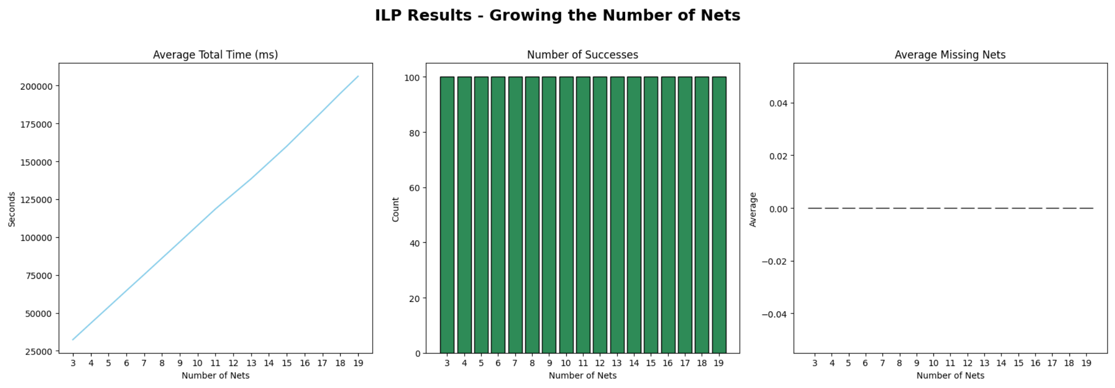

Fig. 10: Impact of growing problem density on the effectiveness and efficiency of the ILP program.

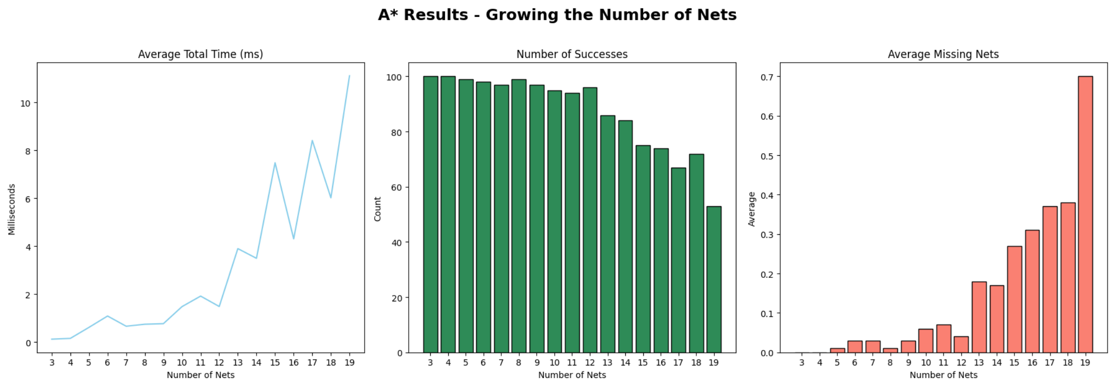

Fig. 11: Impact of growing problem density on the effectiveness and efficiency of the A* program.

1. Due to the very long ILP runtimes, the reported times are not an average over 100 instances, but a single runtime for a single instance. This of course impacts their representativity, but should suffice to infer the general behavior on instances with larger grid sizes.

2. The tested instances are supposed to represent sparse instances where the routing area is much larger than would be necessary for the amount of nets to be routed. The instances tested here are very trivial, because they are limited to 3 nets. This means that no solution optimization has to be performed and the instances could most likely be solved just as well using a greedy approach. However, as seen in Section 7.2, A* seems to also perform well up to medium density, which indicates that these results might be generalizable to realistic cabling problems.

8 Conclusion

I implemented and presented two programs to solve grid-based glass fiber switch box wiring problems. While the optimal ILP program is able to solve all solvable problems within the model's boundaries, it exhibits exponential runtime scaling and increased the runtime by a factor of 340 to 18700, compared to A*, in my experiments. When the grid size increases to about 50, the ILP takes more than an hour to solve a single instance which makes it impractical for any larger problems. On the other hand, A* provides very fast runtime and shows promising solution quality, as long as the instances are not too dense, and can still produce solutions that solve a majority of nets in dense cases, which might significantly reduce the amount of manual clean-up effort needed. This makes my A* program the more practical choice in many scenarios.

Regarding optimizing A* solutions, I implemented and tested two ideas, namely bend minimization and bend flipping, which together were able to reduce the failure rate of A* by a factor of 4. Due to its flexibility, the A* algorithm can also be extended, either to alter the optimization criterion or to increase the quality of solutions and decrease its failure rate further. While the optimization criteria depend heavily on the specific

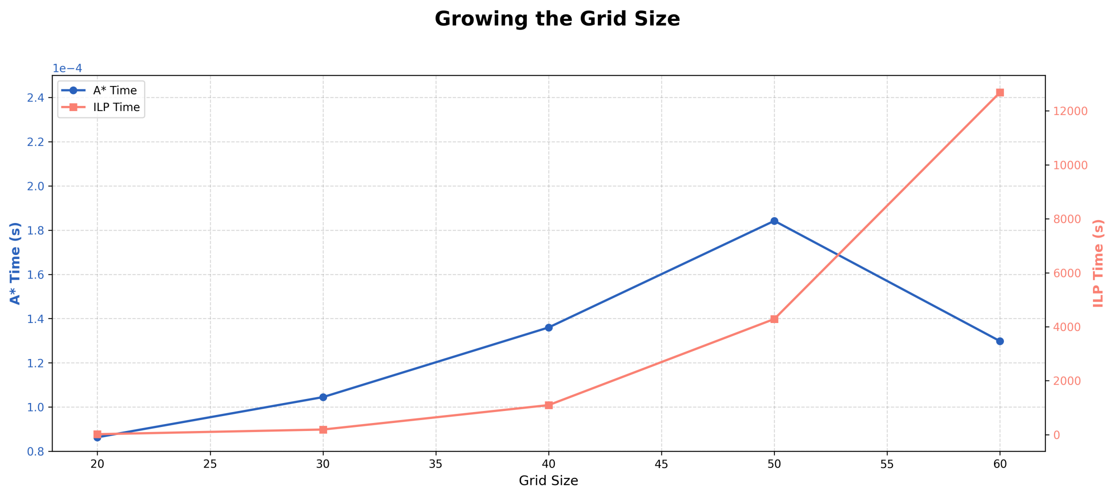

Fig. 12: Impact of a growing grid size on the runtime of the ILP and A* programs. The ILP times are plotted from 0 to 12000 seconds, while the A* times are plotted from 0.08 to 0.24 milliseconds.

application domain, one can think of many solution optimization strategies that are generally beneficial. An idea I mentioned previously is heuristically determining the order in which nets are solved (*e.g.* it might be possible to determine a net-processing order that minimizes the number of blocked nets in a pre-processing step). Furthermore, other rip-up and re-route and additional shove aside strategies could be explored. Here, extra heuristics might be used to determine which of the already routed nets should be altered (in my implementation the choice is arbitrary when there are multiple routed nets which block a missing net).

Regarding performance, my A* program seems fast enough for many practical use cases. However, there are definitely optimizations that can be implemented. Without breaking the current structure of the A* program, the priority queue, based on a Min-Heap, could be replaced with a Fibonacci Heap, which at least theoretically improves its runtime. Additionally, a new algorithm for the Single-Source Shortest Paths (SSSP) problem has been found, which reduces runtime from Dijkstra's $\mathcal{O}(m + n \log n)$ (using a Fibonacci Heap) to $\mathcal{O}(m \log^{2/3} n)$ for sparse graphs [DMM+25]. Implementing A*-search based on it might also be worthwhile in certain cases.

References

[AB10] Mikhail J. Atallah and Marina Blanton: Algorithms and theory of computation handbook. 2: Special topics and techniques. Chapman & Hall, 2. ed edition, 2010, ISBN 978-1-58488-820-8.

[DK82] William A Dees and Patrick G Karger: Automated rip-up and reroute techniques. In *19th Design Automation Conference*, pages 432–439. IEEE, 1982.

[DMM$^+$25] Ran Duan, Jiayi Mao, Xiao Mao, Xinkai Shu, and Longhui Yin: Breaking the sorting barrier for directed single-source shortest paths, 2025. `https://arxiv.org/abs/2504.17033`.

[DS81] William A Dees and Robert J Smith: Performance of interconnection rip-up and reroute strategies. In *18th Design Automation Conference*, pages 382–390. IEEE, 1981.

[EIS76] S. Even, A. Itai, and A. Shamir: On the complexity of timetable and multicommodity flow problems. *SIAM Journal on Computing*, 5(4):691–703, 1976, 10.1137/0205048. `https://doi.org/10.1137/0205048`.

[GTT89] Andrew V Goldberg, Éva Tardos, and Robert E Tarjan: Network flow algorithms. Technical report, 1989.

[GZ88] Teofilo Gonzalez and Si Qing Zheng: Simple three-layer channel routing algorithms. In John H. Reif (editor): *VLSI Algorithms and Architectures*, volume 319, pages 237–246. Springer-Verlag, 1988, ISBN 978-0-387-96818-6, 10.1007/BFb0040391. `http://link.springer.com/10.1007/BFb0040391`, visited on 2025-04-22, Series Title: Lecture Notes in Computer Science.

[Had75] Frank Hadlock: Finding a maximum cut of a planar graph in polynomial time. *SIAM Journal on Computing*, 4(3):221–225, 1975.

[Lee61] Chin Yang Lee: An algorithm for path connections and its applications. *IRE transactions on electronic computers*, (3):346–365, 1961.

[Sou78] JIRI Soukup: Fast maze router. In *Design Automation Conference*, pages 100–101. IEEE Computer Society, 1978.

[Spr88] Alan P Sprague: *Problems in VLSI layout design.* The Ohio State University, 1988.

[Wag91] Dorothea Wagner: A new approach to knock-knee channel routing. In *International Symposium on Algorithms*, pages 83–93. Springer, 1991.

[ZP93] D Zhou and Franco P Preparata: On the manhattan and knock-knee routing models., 1993.