






An Automated Router With Optical Resource Adaptation

Ferre Vanden Kerchove , Xiangfeng Chen , *Student Member, IEEE*, Didier Colle , *Member, IEEE*, Wouter Tavernier, Wim Bogaerts , *Fellow, IEEE*, and Mario Pickavet , *Senior Member, IEEE*

Abstract—Photonic Integrated Circuits are rapidly becoming more reconfigurable using tunable waveguide elements, coming closer to realizing ‘general purpose’ programmable waveguide meshes. To utilize the full potential of such circuits, special software routines need to be developed to determine the optical paths inside the mesh. Right now, current methods either scale exponentially in problem size or are severely lacking performance-wise, largely unable to find solutions, especially in recirculating waveguide meshes with square or hexagonal unit cells. We present an algorithm that computes an efficient configuration that correctly routes all given signals. Whereas similar papers look at meshes containing 7 to 20 hexagonal cells, in this article, meshes of up to hundreds of hexagonal cells are considered. We compare the results of our algorithm to an earlier proposed algorithm and to an optimal solution. Several parameters are introduced in the algorithm. These are studied and an optimizer is implemented to determine effective values for them.

Index Terms—Graph theory, optical routing, photonic integrated circuits, programmable photonics.

I. INTRODUCTION

OVER the last decade, numerous advancements have cleared a path for programmable photonics to advance from a mere concept to a practical platform for new advancements and applications, especially in neuromorphic computing and machine learning [1]. The flow of light is controlled through electronics and software in order to project a set of optical ports at the input to a set of ports at the output. While this demonstrates large-scale integration and programmability, most of these circuits are still very much application-specific photonic integrated circuits (ASPIC). Like most other photonic integrated circuits (PIC), these have to be custom designed and fabricated. In turn, this leads to long design times and high development costs,

Manuscript received 18 November 2022; revised 15 March 2023 and 27 April 2023; accepted 4 May 2023. Date of publication 11 May 2023; date of current version 19 September 2023. This work was supported in part by the Flemish Research Foundation (FWO-Vlaanderen) under Grants G020421 (GRAPHSPAY) and 1100923N, and in part by the European Research Council (ERC) under Grant 725555 (PhotonicSWARM). (*Corresponding author: Ferre Vanden Kerchove.*)

Ferre Vanden Kerchove, Didier Colle, Wouter Tavernier, and Mario Pickavet are with the IDLab, the Department of Information Technology, Ghent University - IMEC, 9052 Ghent, Belgium (e-mail: ferre.vandenkerchove@ugent.be; didier.colle@ugent.be; wouter.tavernier@ugent.be; mario.pickavet@ugent.be).

Xiangfeng Chen and Wim Bogaerts are with the Photonics Research Group, Department of Information Technology, Ghent University - IMEC, 9052 Ghent, Belgium (e-mail: xiangfeng.chen@ugent.be; wim.bogaerts@ugent.be).

Color versions of one or more figures in this article are available at <https://doi.org/10.1109/JLT.2023.3275385>.

Digital Object Identifier 10.1109/JLT.2023.3275385

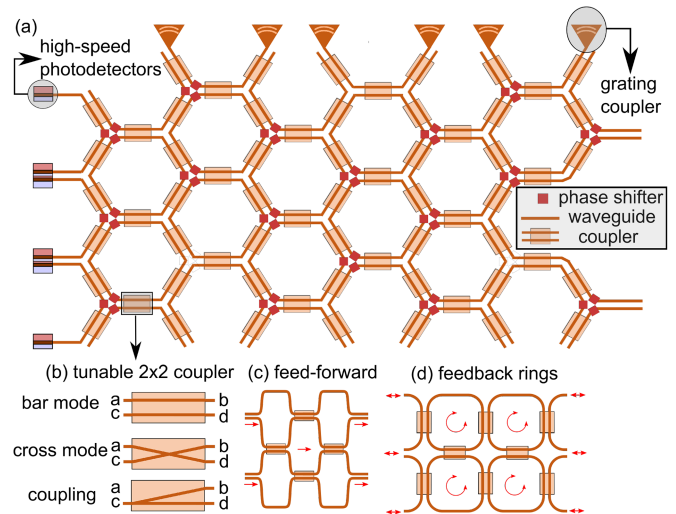


Fig. 1. Programmable photonic circuits: (a) the circuit connects inputs, outputs, and functional blocks (modulators, detectors). (b) internally it has couplers that can be put in different states. The topology can be feed-forward (c) or use feedback rings, these can be organized and tiled in different shapes, e.g., squares (d) or hexagons (a).

slowing down the exploration of novel optical applications [2]. As a result, general-purpose programmable photonic chips are now touted as the next logical step in the development of PICs, providing a chip where the flow of light can be arbitrarily programmed to perform a variety of functions, which can accelerate prototyping and developments. The programmable flow of light enables dynamic manipulation over the course of light and thus a programmable control of the functionality of the circuit. The most commonly used architecture is a mesh where light flows in one direction [3] as seen in Fig. 1(c). A forward-only mesh is easy to understand but does not completely utilize the full potential of programmable photonics. Alternative architecture tiles the chip with waveguides organized in triangular, square, or hexagonal loops, the latter two can be seen in Fig. 1(a) and (d). Here, light can be folded onto itself, creating resonances. This enables the implementation of a broad range of different wavelength filters. These meshes can also contain specialized function blocks inside the mesh, with dedicated input and output ports.

Right now, a popular and potent mesh topology uses hexagonal tiling, offering significant advantages over other regular tiling [4], as an example, only a hexagonal mesh can implement Sagnac loops. In this article, we consider only this architecture.

However, the proposed algorithm does not depend on the exact architecture, and can easily be used with other architectures.

The functionality of a waveguide mesh that offers recirculation is twofold. Firstly, it provides the connections between all possible pairs of ports, including the different high-performance photonic blocks, essentially acting as an all-to-all switch. Secondly, it enables the synthesis of the aforementioned optical filters inside the mesh, such as interferometers and ring resonators. This article only focuses on the first functionality, but the proposed algorithm is left as much room as possible to allow for an extension to the placement and synthesis of filters.

Given a set of signals with their source ports and corresponding destination ports, the main problem is deciding the exact path for every signal. These paths cannot share waveguides, severely limiting what is physically possible. On smaller meshes, a path for every signal can still be manually constructed. On larger meshes, this becomes a challenging task to perform for humans. A logical step is to use specialized routing algorithms. However, for photonic purposes, these algorithms are still largely in their infancy. They either assume one-directional flow in forward-only meshes [3], or focus on small-scale recirculating meshes [5], [6]. Correctly modeling all constraints imposed by the physics of light proves to be a real hurdle, which leads to algorithms that do not manage to properly explore all possible configurations.

We base our work on [5], [6], [7], focusing on meshes that contain an order of magnitude more unit cells. While the aforementioned papers look at meshes that consist of 7 to 20 hexagonal cells, in this article, meshes of hundreds of hexagonal cells are considered. This article is an in-depth extension of [8].

Section II defines the problem and gives the relevant physical constraints. We then translate this problem into a problem on graphs. Section III provides a general overview of our algorithm. Section IV gives a complexity analysis and Section V describes the integer program that is used to obtain an optimal solution. In Section VI, we briefly talk about the test data that is used. Afterward, in Section VII, we gauge the quality of the results. The results are compared to solutions produced by an algorithm similar to [6] and an optimal solution, the latter being only obtainable on smaller meshes. We justify design and parameter choices by showing performance data in Section VIII. At last, in Section IX, we summarize our work and give areas where future work is possible.

II. PROBLEM STATEMENT

Firstly, a broad description of the problem is given with the relevant physical constraints and optimization goal. Then, the problem is reformulated to a problem on graphs, and an equivalent way to realize the constraints is given. This second formulation is the basis of the Automated Router with Optical Resource Adaptation (Aurora) and ensures that it is fully compliant with the constraints.

A *commodity* in a mesh is a source-target connection pair, where light needs to flow from source to target. The main problem of this article is the following: given a mesh consisting of waveguides and couplers, and a set of commodities, find a

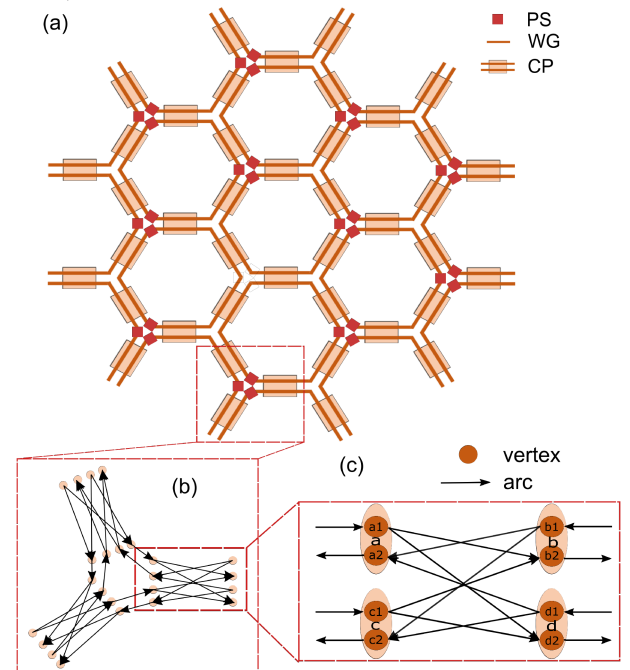


Fig. 2. (a) A hexagonal mesh consisting of waveguides (WG), couplers (CP), and phase shifters (PS). (b) a set of interconnected couplers. (c) a coupler as a graph.

way to connect each source with its target through a path for the light in the mesh. A path can only change from one waveguide to another by utilizing a *coupler*. Here, this acts as a switch as seen in Fig. 1(b). Either it lets the light continue in the waveguides, or it switches the light from one waveguide to the other, and vice versa. This is called bar and cross mode respectively. There is a third mode, coupling mode, but this is only relevant if there are multiple targets per source, or for the realization of other optical functions like wavelength filtering. Neither is considered in this article. The specific design of couplers leads to the following physical constraints. A path using a coupler cannot make a U-turn in this coupler, nor can a coupler be used in both bar and cross mode at the same time. The following set of constraints is an equivalent way to capture the physical restrictions.

- 1) U-turns are not allowed in couplers.
- 2) No waveguide can be used by two different paths.
- 3) A single path cannot use a waveguide twice

Now, graphs are a natural framework to tackle this problem. Recently, new developments [5] have shown an interesting graph representation of couplers, enabling an abstraction of physical constraints that aids algorithm design. Graphs have been extensively studied and a rich field of algorithms can now be leveraged. Fig. 2(c) shows this graph representation.

Logically, a coupler could be modeled by 4 vertices where light enters and exists. Doing so proves cumbersome because this creates paths in the graph that are not physically possible (such as back-coupling light from one input to the other input). This simple model does not represent the underlying problem well. As a consequence, the routing algorithm would need to take care of this, adding additional complexity.

Instead, a directed graph is used. This type of graph has directed edges, called arcs. Besides this, every vertex is separated into an incoming vertex and an outgoing vertex, this immediately disallows U-turns. See [5] for a more in-depth treatment. This translation comes with drawbacks, as every waveguide and coupler arm is now modeled by two arcs, each in one direction. For every arc, there is a *reverse arc* that represents the same waveguide or coupler arm, but going in the opposite direction. Of the three earlier listed constraints, the second constraint, which states that no waveguide can be used by two different paths is thus not translated to “no arc can be used by two different paths”. Instead, this becomes “no arc or its reverse arc can be used by two different paths”. The third constraint is adapted similarly. Notice that each node has either at most one incoming arc or one outgoing arc.

Because of production imperfections, couplers differ in physical properties such as insertion loss and power consumption. Waveguides have similar imperfections. To account for this, couplers and waveguides are given a base weight connected to their inherent physical properties. Since couplers and waveguides are represented by arcs in a graph, every arc is given a base weight corresponding to the characteristics of the physical structure it represents. Weights are further explained in Section III-C.

The problem is now stated more rigorously as follows. Given a weighted, directed graph $G = (V, A)$ of vertices (or nodes) and arcs, and a set of pairs of vertices, $(s_1, t_1), \dots, (s_n, t_n)$ called commodities, find a set S of paths $\{P_1, \dots, P_n\} = S$, such that P_i is a directed path from s_i to t_i and that no arc is *congested*.

An arc a is congested in the following situations.

- 1) Two different paths use a
- 2) A path uses both a and its reverse arc
- 3) A path uses a and another path uses its reverse arc

If a path contains a congested arc, this path is said to have a *conflict* or to be *congested*. This definition is slightly broader than what one would normally consider congested in similar algorithms, but a necessary adaptation nonetheless. If the congestion definition was not extended with reverse arcs, there would be no problem with one path using an arc and another path using the reverse arc since these are distinct arcs. However, remember that an arc and its reverse arc model the same physical waveguide, thus violating the constraint that no waveguide can be used by two different paths. A set of paths is a *legal routing* or a *solution to the problem instance* if every source is connected to its target by a conflict-free path. A problem instance is feasible if there exists a solution.

The total weight of a solution S is equal to the sum of the weights of the paths P in S . In turn, the weight of a path is the sum of the weights b_a of the arcs a in P .

$$\text{Total weight of solution } S = \sum_{P \in S} \sum_{a \in P} b_a.$$

Now, the problem is finding a solution where the total weight of the solution is minimal, i.e., no other solution exists with a lower total weight. Given two solutions, we call a solution *better* if it has a lower total weight than the other solution. An example of a problem instance and a legal routing is shown in Fig. 3. Here, on the left of the figure, you can see 10 colored nodes and arrows,

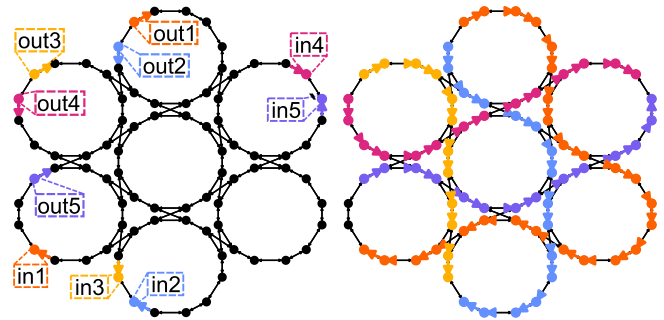


Fig. 3. An example problem instance on the left and an optimal solution regarding the total length of all paths on the right. There are ten colored nodes, and of each color, there is a node with an outgoing and incoming colored arc. These are the source and destination, respectively. The solution displays a path for every commodity such that all physical constraints are respected.

with each color having one node with an outgoing arrow and one with an incoming arrow. These represent the source and target nodes respectively. The right figure displays a legal routing that is optimal, in regard to the fact that every arc has a weight of 1.

III. THE ALGORITHM

We propose a negotiation-based algorithm, partially based on Pathfinder [9] and additional improvements [10], [11], [12]. This is a logical choice, given the fact that modern routing algorithms in electronic FPGA design tools are also based on these principles [13]. Several modifications are incorporated, some to take the physical constraints into account, and others to improve performance. One example of such a modification is the expanded definition of congestion, as mentioned in the previous section.

A. Sequential Routing

A popular and straightforward way to tackle this problem is with sequential routing [6]. This follows the principle that one commodity is routed first along its shortest path. Then the following commodity is routed but this path cannot use the routing resources that are already used in the previous commodity’s path. The path for the next commodity cannot use the resources of the two paths that were routed before. This continues until all commodities are routed, or until there does not exist a path anymore between a source-target pair. When there is a large number of commodities, the latter is quite likely. This type of routing fails to account for the fact that certain commodities could easily use an alternative path that does not need crucial resources, whereas others cannot. As an example, in Fig. 4, commodity A has as source A_s and as target A_t , and commodity B has B_s and B_t . For this problem, the path between the commodities can share nodes, but cannot share edges. Neither commodity can have its shortest path, since that would completely cut off the target of the other commodity, and both commodities need to compromise.

If there are n commodities, then there are $n!$ possible permutations to route the commodities sequentially. Firstly, it is infeasible to fully explore this search space even for n as small

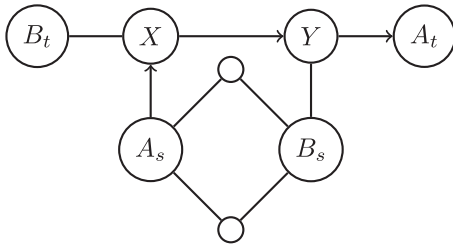


Fig. 4. The shortest path for commodity A is $A_s - X - Y - A_t$, indicated by the arrows. However, this uses both edges (A_s, X) and (X, Y) . There is now no longer a path for commodity B possible. The situation is analogous when B is routed by its shortest path. Instead, both need to take a longer path.

as 15. Secondly, as was just shown, there might not even be a solution that is obtainable in this way.

This motivates us to use an iterative routing algorithm that does not route sequentially, but simultaneously. The main idea of iterative simultaneous routing is that, in every iteration, all commodities are routed independently of each other. Instead of being disallowed to use certain routing resources that others use, it is possible to temporarily share them. To decide which routing resources the paths use, the algorithm is guided by continuously changing weights that represent the artificial cost to use these resources. These weights cause shared routing resources to become increasingly more expensive. This discourages the usage of heavily sought-after routing resources by making them prohibitively expensive, causing commodities to use a cheaper detour. Commodities that do not have another option will still use these expensive routing resources.

If sequential routing finds a solution, that solution is often fairly good. This stems from the fact that everything is routed through the shortest path that is still possible, which generally leads to short solutions. As will be explained in Section III-E, our algorithm sometimes switches between sequential and simultaneous routing, but the majority of the time it uses simultaneous routing.

B. Aurora: An Automated Router With Optical Resource Adaption

We now give an overview of our algorithm, accompanied by the pseudo-code in Fig. 5. Basic variables are initialized on line 1 to 3 such as the number of solutions sol it has found so far and the current best solution $best$. Besides this, the variable $paths$ stores the current path for each commodity, starting with having no path for each commodity. At first, a round of preprocessing is conducted which slightly modifies all weights. This is further explained in Section III-D.

Given a graph G and a set $coms$ of commodities containing the source and target nodes of the commodities, the first round of routing is conducted (line 8) with Dijkstra's algorithm [14]. This routing is based on the weights of the arcs. After the initial round of routing, every commodity has now a least-weighted path, independent of the other commodities.

The main loop starts. Firstly, if there are congested arcs, the *congestion weight* of these arcs is updated (line 10). This is further specified in Section III-C and in general, depends on

Require: commodities $coms$ and graph G

```

1:  $sol \leftarrow 0$ 
2:  $best \leftarrow \emptyset$ 
3:  $paths \leftarrow \emptyset$ 
4: for  $arc$  in  $G$  do
5:    $weights[arc] \leftarrow base\_weight(arc)$ 
6:  $weights \leftarrow PREPROCESSING(G, coms)$ 
7: for  $co$  in  $coms$  do
8:    $paths[co] \leftarrow DIJKSTRA(G, co, weights)$ 
9: for  $iter \leftarrow 1 \dots max\_iters$  do
10:   $UPDATE\_WEIGHTS(G, paths, weights)$ 
11:  if  $CONFLICTS(paths)$  then
12:    for  $co$  in  $coms$  do
13:      if  $paths[co]$  has conflict then
14:         $paths[co] \leftarrow DIJKSTRA(G, co, weights)$ 
15:      if  $REPETITION\_AVOIDANCE(paths, iter)$  then
16:         $SEQUENTIAL\_ROUTING(G, coms, weights)$ 
17:    else
18:       $best \leftarrow CHOOSE\_BEST(paths, best)$ 
19:       $sol \leftarrow sol + 1$ 
20:      if  $sol = max\_solutions$  then
21:        break
22:       $RESET\_CONGESTION(G, weights)$ 
23:       $LENGTH\_BASED\_RIP\_UP(G, paths)$ 
24: return  $best$ 

```

Fig. 5. Pseudo-code for the Automated Router with Optical Resource Adaption (Aurora).

how many paths use this arc and how many iterations this arc has been congested so far. Now, if there are congested paths, the algorithm rips up these paths and reroutes them (line 12–14), with the routing now based on the updated weights. On line 15, our algorithm avoids repeating the same configurations too many times, as explained in Section III-E. Thus, if deemed necessary, a round of sequential routing is then conducted with all commodities that have conflicts. This sequential routing is based on the weights that the arcs have at that time, but after every commodity, the weights are immediately updated, instead of only being updated after all the conflicted commodities are rerouted.

If a solution is found, it is recorded (line 18). This event is called a convergence. Note that if another solution has already been found, the choice of which solution is kept is made based on the lowest total base weight. Now there are two possibilities, either the algorithm immediately returns the solution and terminates, or it continues to search for better solutions, i.e., solutions with a lower base weight. For the latter, the algorithm is said to be multi-convergent. If this is the case, the congestion term of the congestion weight of the arcs is reset to 0 (line 22), and a cost-based rip-up is initiated (line 23). For this, only the base weights are considered. For every commodity, the base cost of their path in the solution is compared to the cheapest (in terms of base weights) possible path for this commodity. All paths that are a certain percentage more expensive than their cheapest possible path are artificially marked as congested so that they

are rerouted when the next iteration starts. See Section VIII-C for an exact description. The congestion reset ensures that these more expensive paths will try to take better routes than the route they had in the last solution while a complete restart is avoided in order to keep the good parts of the previous solution. This continues until a certain number of iterations has passed or a fixed number $max_solutions$ of solutions are found (line 20). When this happens, the algorithm either returns the best solution it found or reports that it has failed to find a solution. This failure is an indication that there might not be a solution at all, but not a guarantee. In Section VIII-E, we study a good choice for this maximum number of iterations.

C. Weights

A major difference from Pathfinder is the fact that we give weights to the arcs, instead of the vertices. This enables us to model more directly the intrinsic differences of waveguides and couplers, such as insertion loss (IL), power consumption (PC), and basic unit length (BUL). Every arc a has a *base weight* b_a which is a linear combination of these three properties, $b_a = c_1 \cdot IL_a + c_2 \cdot BUL_a + c_3 \cdot PC_a$, where c_1, c_2, c_3 are scaling coefficients which can be chosen according to need. For simplicity's sake and without loss of generality, in this article, we only focus on the basic unit length and assume that all arcs have a BUL of exactly 1. Thus we put $c_2 = 1$ and $c_1 = c_3 = 0$, hence the base weight of every arc is 1. Since the base weight of every arc is the same as its length, we minimize the total length of the solution. The total length of a solution is the sum of the lengths of the arcs in the paths of that solution. From here, when we describe a solution as being longer than other solutions, we mean that it has a higher total base weight compared to these other solutions. Given solution a with a total length of 204 and solution b with a total length of 200, solution a is said to be 2% longer than solution b . In general, if l_a is the total length of a and l_b the total length of b , then $\frac{l_a - l_b}{l_b}$ calculates how much longer solution a is than solution b . Negative numbers imply that a is shorter instead.

The *congestion weight* w_a of an arc a is now the following:

$$w_a = (b_a + \varepsilon_a + h_a \cdot HI) \cdot (p_a + 1).$$

Here b_a is the base weight. ε_a is either 0 or a small value ε , depending on preprocessing, see Section III-D. h_a keeps track of the number of iterations that this arc was congested before, whereas HI is a constant called history increase. Different values for the history increase lead to varying performance which we study in Section VIII-B. h_a starts at 0 and increases by one for each iteration this arc or its reverse arc is congested. This makes the arc progressively more expensive and paths will avoid arcs that are often congested. Instead, they will choose detours.

For every path, p_a is the number of other paths that used this arc or its reverse arc in the previous iteration. The exact definition of p_a is motivated by the following observation. Firstly, when an arc is heavily congested and used by many paths, it immediately becomes more expensive, causing commodities to change their

path the next round. Besides this, it discourages other commodities to use this arc in the next round. We specifically add “+1” in $p_a + 1$ because otherwise, all unused arcs would cost 0. All congested paths would always want to use unused arcs, causing many commodities to completely change their path, irrelevant of how much worse this new path is. Now p_a is taken as the number of other paths using this arc or reverse arc, noticeably not counting the current path for which we are calculating the weights. This follows similar reasoning. If we took all paths into account, then a commodity would make its own path more expensive as well. If there is any congestion on that path, it will completely reroute and avoid the arcs that it uses in the previous iteration, even though many of these might not be used by any other path.

D. Preprocessing

Right now, the mesh architectures under investigation often have many equal-cost least-weighted paths between two vertices. The algorithm makes use of this property by calculating all the least-weighted paths between a sink and its corresponding target of a commodity. If two least-weighted paths between two different source-target pairs use the same arc or reverse arc, the cost of that arc is slightly increased by a value ε . If a commodity has many least-weighted paths, it will prefer the path that has no arcs shared with a least-weighted path of another commodity.

E. Configuration Repetition Avoidance

The algorithm sometimes gets stuck, where one iteration it suggests routing R_1 , the next iteration R_2 and it keeps alternating between these two. On one problem instance, a cycle of 4 configurations involving 3 different paths was found. Seemingly this does not appear often, but enough to warrant a modification that helps the algorithm avoid useless iterations consisting of exploring the same sequence of configurations over and over again. It is even vital to combat this issue, otherwise, on certain problem instances, no solution is found at all. Various schemes to resolve this issue have been tested, but the performance data favors a basic rule in the end. After a fixed number of iterations, one round of sequential routing is conducted in random order. Schemes that try to detect when configuration repetition is happening were outperformed both in time and solution quality by this rule.

F. Cost-Based Rip-Up

Once a solution is found, the algorithm does not terminate. Instead, it resets the congestion history term h_a and looks at the base weight of the paths. Paths that are $R\%$ more expensive than their cheapest possible path are artificially marked as conflicted. Now the algorithm restarts by immediately ripping up these paths and the history increase is scaled as well by a factor v . This continues until a maximum number of solutions, $max_solutions$, is reached. We investigate the exact values for these parameters in Section VIII-C.

TABLE I
COMPLEXITY ANALYSIS

In terms of	E, V and C ,	r and C .
Dijkstra	$O(E + V \log V)$	$O(r^2 \log r)$
Single loop	$O(C (E + V \log V))$	$O(C r^2 \log r)$
Total	$O(C ^2(E + V \log V))$	$O(C ^2 r^2 \log r)$

IV. COMPLEXITY

In this section, the complexity of the algorithm is studied. This analysis comes with the caveat that a large part of the total time complexity is hidden by constants and that the “difficulty” of the problem instance has a large influence on the total time needed. This is expected because this algorithm is a heuristic for an NP-hard problem [15] which makes it generally hard to give meaningful complexity analysis.

We characterize the complexity in two ways, firstly for general graphs, and in terms of the number of edges $|E|$, vertices $|V|$, and commodities $|C|$. And secondly, for the specific hexagonal tiling that we use, which is in terms of the radius r and the number of commodities $|C|$. Our implementation of Dijkstra’s algorithm has a time complexity of $O(|E| + |V| \log |V|)$. Now, in the graph we use, for every node, the sum of the indegree and outdegree is at most 3, and $|E| \leq 3|V|$, thus this time complexity simplifies to $O(|V| \log |V|)$. The number of nodes is a multiple of the number of hexagons which, in turn, scales quadratically with the radius r . This gives us a complexity $O(r^2 \log r)$ for Dijkstra’s algorithm, which is part of the main loop. Dijkstra’s algorithm is called at most once¹ for every commodity in the main loop. This happens only for commodities that have a conflict. Hence, the complexity becomes $O(|C|r^2 \log r)$. As discussed in Section VIII-E, the maximum number of iterations of the main loop is a multiple of the number of commodities and thus a total complexity of $O(|C|^2 r^2 \log r)$. See Table I for the description in terms of $|E|$ and $|V|$.

V. INTEGER PROGRAM FOR AN OPTIMAL SOLUTION

We describe the integer program that we utilize to obtain an optimal solution. This integer program is strongly dependent on the graph representation that is used. As mentioned before, every arc a has an opposite arc, a^{op} . Now we introduce variables $x_{ca} \in \mathbb{B}$ for every commodity c and every arc a , indicating that the path for this commodity uses this arc. A commodity c has a vertex c_s and c_d denoting the source and destination of that commodity. The set of all commodities is denoted by C and the set of all arcs by A . We use the set of all vertices V and let $V^*(c)$ be the set of all vertices, without the source and destination of c , thus $V^*(c) := V \setminus \{c_s, c_d\}$. Let $\delta^-(v)$ be the set of all incoming arcs of a vertex v , and $\delta^+(v)$ be the set of all outgoing arcs. The integer program now becomes the following:

$$\text{Minimize } \sum_{c \in C} \sum_{a \in A} b_a x_{ca}.$$

¹Technically twice, because it can be called again in the sequential routing, however, this only happens a fixed number of times. This does not change the time complexity analysis.

$$\begin{aligned} \text{Subject to } \sum_{a \in \delta^-(n)} x_{ca} &= \sum_{a \in \delta^+(n)} x_{ca}, & \forall c \in C, \\ & & \forall n \in V^*(c), \\ \sum_{c \in C} x_{ca} + x_{ca^{\text{op}}} &\leq 1, & \forall a \in A. \end{aligned}$$

The objective function is as stated before, to minimize the total base weight of the used arcs. The first constraint is Kirchhoff’s law which makes sure that in every vertex, if a path enters the vertex, it also leaves the vertex, except in the case that this is the source or destination vertex for this commodity. The second constraint states that for every arc, at most one commodity can use this arc or its reverse arc.

For every commodity c , the following 4 constraints are also added. This ensures that there is a path starting in the source c_s and going to the destination c_d .

$$\begin{aligned} \sum_{a \in \delta^-(c_s)} x_{ca} &= 0, \\ \sum_{a \in \delta^+(c_s)} x_{ca} &= 1, \\ \sum_{a \in \delta^+(c_d)} x_{ca} &= 0, \\ \sum_{a \in \delta^-(c_d)} x_{ca} &= 1. \end{aligned} \quad (1)$$

Note that this fully captures the definition of congestion, proposed in Section II. This integer program completely represents the problem, but largely because of the underlying graph representation. To give an example of this dependence, we do not need to specify that a path cannot pass through the source of another commodity. This follows from the fact that a source node is always chosen to have exactly one outgoing arc, and thus that arc has to be used for the path of that commodity.

VI. TEST SETS

We use multiple test sets to benchmark our algorithm. One test set consists of multiple related problem instances with the same radius r , which is first explained. A mesh of radius r has $1 + 3r(r + 1)$ hexagonal cells. One single hexagonal cell is considered a mesh of radius $r = 0$. In Fig. 3, a mesh of radius $r = 1$ can be seen. Most test sets are on a mesh of radius $r = 8$, which consists of 217 hexagons. This radius is chosen because it is the largest on which optimal solutions can still be found systematically. For every problem instance in a test set, it is exactly known if it is feasible or not. If it is feasible, the optimal solution is also known. This is realized by the integer program described in Section V.

Now, every test set is based on a radius r and a list of commodity candidates $(s_1, t_1), \dots, (s_n, t_n)$. The first problem instance, P_1 , consists of routing the first commodity (s_1, t_1) on a mesh of radius r . The second problem instance, P_2 , is routing the first and second commodity $(s_1, t_1), (s_2, t_2)$. Now at a certain index m , it is possible to route commodity 1 through m , but

routing commodity 1 through $m + 1$ is infeasible. Remember, this infeasibility is shown by an integer program and thus proven infeasibility. Then, commodity $m + 1$ is not included in the next problem instance, and the next commodity is added instead. If this is feasible, problem instance P_{m+1} becomes routing $(s_1, t_1), \dots, (s_m, t_m), (s_{m+2}, t_{m+2})$. If this is again infeasible, the process repeats itself and the next commodity is tried. This gives a chain of related problem instances, where each problem instance is more difficult than the previous one. Another advantage is that the problem instances are ‘randomly’ generated. In a solution of the last problem instance of a test set, the mesh is often quite full, with little space for more commodities. We say that the mesh is *densely* used then. As a consequence of this way of generating problem instances, the number of commodities can differ between problem instances, and having more commodities does not necessarily mean that a problem instance is more difficult. Nor that it is more densely used. Instead, there might be a couple of commodities where the source and target are located close to each other. This is often easier to route than when the commodities are always located far away from each other and many paths cross each other. The list of commodity candidates is generated with the sources and destinations randomly distributed through the mesh, but an emphasis is put on populating the outer layers of the mesh, simulating the functionality of the mesh as a switch box.

VII. COMPARISON OF AURORA TO SEQUENTIAL ROUTING AND AN OPTIMAL SOLUTION

Here, we look at the performance of the algorithm, Aurora, with optimized parameter values. To avoid the fact that the parameters are overfitted to the problem instances that are used to obtain the parameters, there are two sets of test sets. One on which the optimized parameter values are obtained, and another, larger one. All plots here are created from the performance of Aurora on the second set of test sets, thus ensuring that the parameters are not specifically adapted to the exact test sets that were used to obtain the parameter values. The optimizer and the exact parameter values are described and studied in the section after this one, Section VIII.

We compare Aurora to a sequential routing algorithm, as described in [6]. To improve the results of this algorithm, instead of only trying one order, 50 random orders are tried, and the best result is kept. As shown in Fig. 6, sequential routing fails on some smaller instances and on almost all medium and larger instances. This is not unsurprising as explained in Section III-A. Depending on the exact location of commodities, it might not be possible that a solution can be found, and the possible number of different orders to try grows very quickly. However, this does not make it impossible, but rather unlikely. For example, sequential routing finds a solution at 21 and 22 commodities a couple of times. This explains the fluctuations in the number of unsolved feasible instances. In comparison, Aurora manages to solve 100% of the feasible problem instances, thus outperforming sequential routing by a wide margin when considering the number of solved instances. If we only look at the subset of problem instances that sequential routing can solve, its performance is similar to Aurora

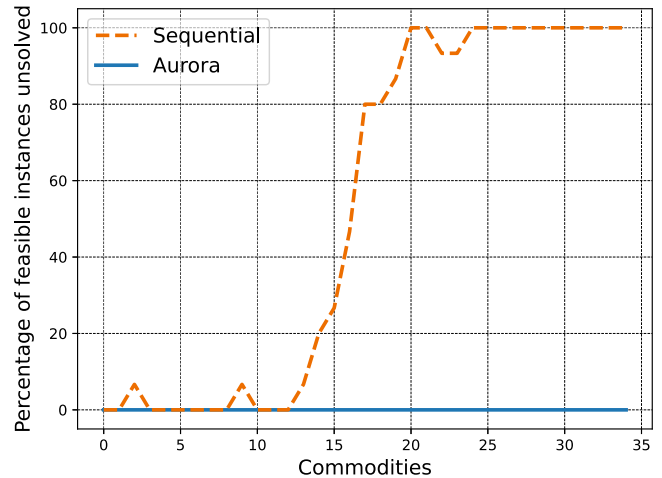


Fig. 6. Percentage of unsolved feasible instances of sequential routing over all test sets with radius $r = 8$ by the number of commodities.

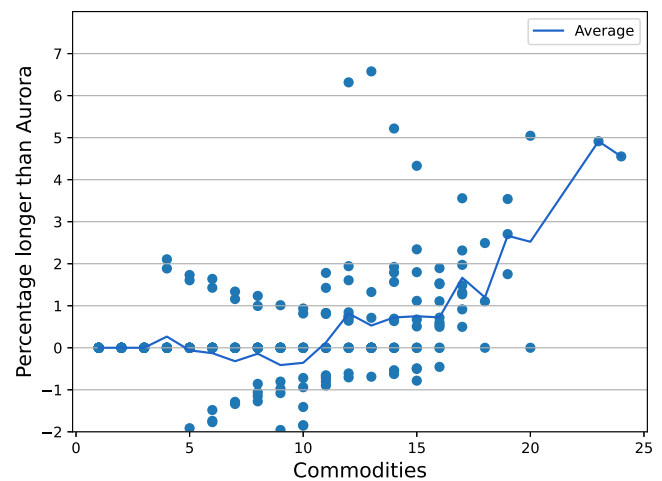


Fig. 7. A comparison of sequential routing versus Aurora on all instances for which sequential routing found a solution. They perform similarly for a small number of commodities, but this degrades quickly with a rising number of commodities.

on smaller instances. On the few larger instances that it manages to solve, it has around 2–5% longer path results than Aurora, see Fig. 7. In Fig. 8, the time needed for Aurora and sequential routing is compared on the instances where sequential routing finds a solution. Towards the higher number of commodities, this happens only on a few instances. Aurora is much faster on the entire range. Sequential routing only finds solutions on ‘easier’ problems with few commodities for the size of the mesh, hence Aurora quickly finds solutions here as well.

Looking at the general performance of Aurora in Fig. 9, on average, it stays well within 2% of the optimal length. Even at the ninety-fifth percentile, the solutions are at most 4% longer than optimal. The time needed for Aurora in comparison to an integer program can be seen in Fig. 10 where Google OR-tools [16] is used as the solver. Here we can see that Aurora scales significantly better, unsurprisingly given the exponential nature of a solver for integer programs. We can see that it is several orders of magnitude faster, which only becomes more apparent with

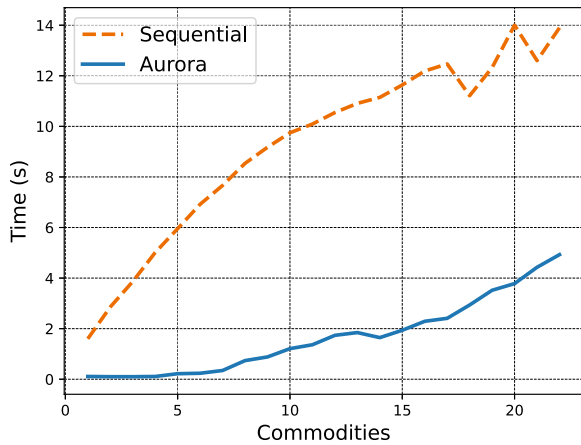


Fig. 8. Average time needed of Aurora and sequential routing on all feasible problem instances where sequential routing finds a solution. Sequential routing barely finds solutions towards the higher number of commodities, hence the jagged line towards the end.

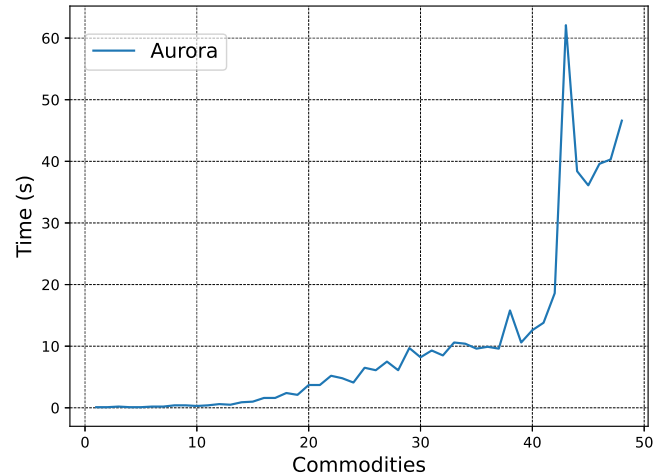


Fig. 11. Number of commodities versus the time needed on a mesh of radius $r = 13$.

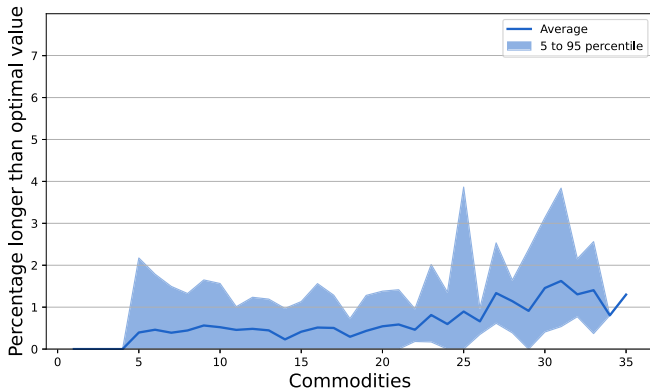


Fig. 9. Performance of Aurora with parameter values according to differential evolution, see Section VIII. History increase: 0.053, ϵ : 0.335, maximum convergence: 6, rip-up: 17.5%, history reduction: -0.27 (Lower is better.).

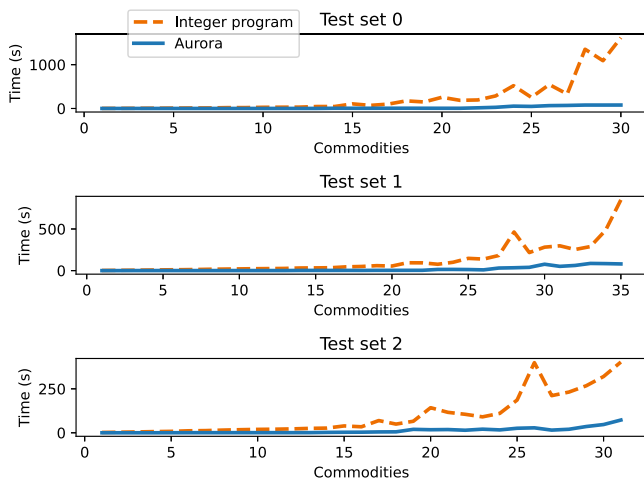


Fig. 10. Time comparison of an integer program and Aurora on 3 test sets. Other test sets paint a similar picture. We have plotted single test sets because of the large variation in the time needed between test sets.

larger mesh radii and more commodities. For radius $r = 8$, the integer program solver often needs multiple hours to days to find all solutions to the problem instances of one test set. For higher radii, this is increased by another order of magnitude. Lastly, we look at the time needed on a mesh of a larger radius as can be seen in Fig. 11. The radius here is $r = 13$ and the mesh contains 546 hexagons. Besides a spike at 43 commodities, the algorithm quickly finds a solution to all 48 problem instances in less than 50 seconds.

VIII. PARAMETER TUNING

A. Performance Metrics

In this section, we study more extensively the effect of parameter values and elaborate on how we optimize the parameter values. In this article, we limit our scope to meshes that are based on hexagons. However, this limitation is not used anywhere in the algorithm. The mesh's exact topology might impact the optimal parameter values, but then these values can again be found by the parameter tuning in this chapter. We do not expect the effect of parameters such as history increase and restart to change drastically.

The quality of a specific parameter setting is measured by 3 key factors. These are listed in order of importance.

- 1) Number of feasible problem instances solved
- 2) Percentage longer than the optimal length
- 3) Time

To explore the parameter space smartly, we use differential evolution as described in [17]. Differential evolution spreads agents in the parameter space, and each agent represents a specific choice for every parameter. These agents then explore the space by choosing to either keep their own values or change to a linear combination of the parameter values of other agents. Differential evolution was chosen for its simplicity in design and the convincing performance as mentioned in [17]. We use the recommended hyperparameter settings but limited the number of agents to 10 and the number of differential evolution iterations to

8. This is a low number of agents and iterations and is chosen to reduce computation time. Besides this, as seen in various plots, there is a degree of randomness in the result for each parameter value. There is little reason to try to find if the optimal value for a parameter is 0.338 or 0.339 since this is more depending on the exact problem instances we use to run differential evolution on. We are more interested in general trends and good parameter ranges, which we study here. The overall data used for the plots that follow are taken from the data gathered during differential evolution.

There are several parameters we study. These are history increase (Section III-C), history vary (Section III-F), ε (Section III-D), rip-up percentage and the maximum number of convergences (Section III-F). Given a value for these, Aurora runs on test sets each comprising several problem instances. We then get the results produced by this version of the algorithm on one test set and the fitness is calculated. This fitness is computed with the following fitness function f : if u is the number of unsolved feasible problem instances, l_{opt} the sum of the optimal lengths of all the problem instances in one test set, and l_{sol} the sum of the solution lengths in that test set:

$$f(u, l_{\text{opt}}, l_{\text{sol}}) = 10 \cdot u + 100 \cdot \frac{l_{\text{sol}} - l_{\text{opt}}}{l_{\text{opt}}}.$$

Note that for this fitness function, lower is better, with fitness of 0 meaning optimal. This convention is chosen to agree with the literature on differential evolution. This fitness function largely focuses on solving all instances, and then afterward on finding solutions close to optimal. We investigate the effect of the parameters on the runtime, but it is not one of the objectives we explicitly minimize for.

B. General Trends for History Increase

We extensively compare different parameter choices and their effect on performance. Firstly, we look at the effect of history increase. For history increase, we studied values ranging from 0.01 to 1. This coincides with being 1% to 100% of the base weight of an arc. A couple of general trends are clear. Too small of a value often leads to the algorithm not finding a solution as seen in Fig. 12. It runs out of iterations since too little happens with each iteration. For most values of history increase, there are some unsolved feasible instances, indicating that a single value for history increase might not be preferable. Fig. 13 is a plot of how much longer the solution is than the optimal value in function of history increase. An abundantly clear trend can be seen, a higher history increase leads to longer and thus worse solutions.

Secondly, we look at the effect of history increase on time. Different test sets have a varying number of problem instances and a different number of commodities for each problem instance. Thus to have an insightful comparison, it is necessary to normalize the time of each test set. Given a test set T , a specific choice of parameter values p_{sp} and the set of all parameter values P , the normalized time for the parameter choice p_{sp} on T is:

$$\text{normalized time}(T, p_{sp}) = \frac{\text{time}(T, p_{sp})}{\min_{p \in P} \text{time}(T, p)}. \quad (2)$$

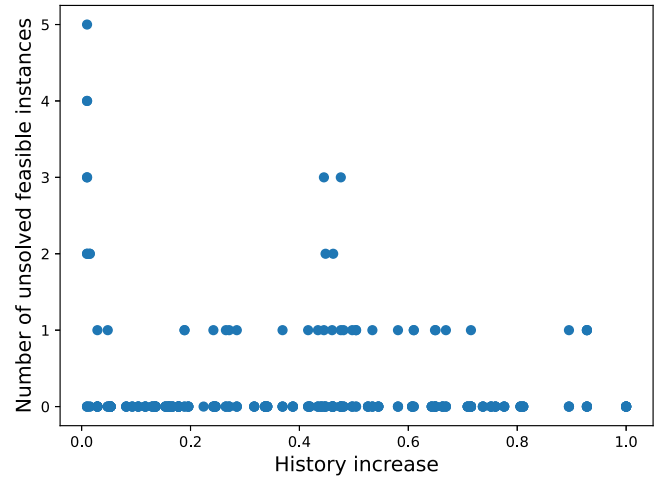


Fig. 12. Number of unsolved feasible instances, one dot represents a result on a test set, and every test set contains between 25 and 35 problem instances. (More unsolved instances is worse.).

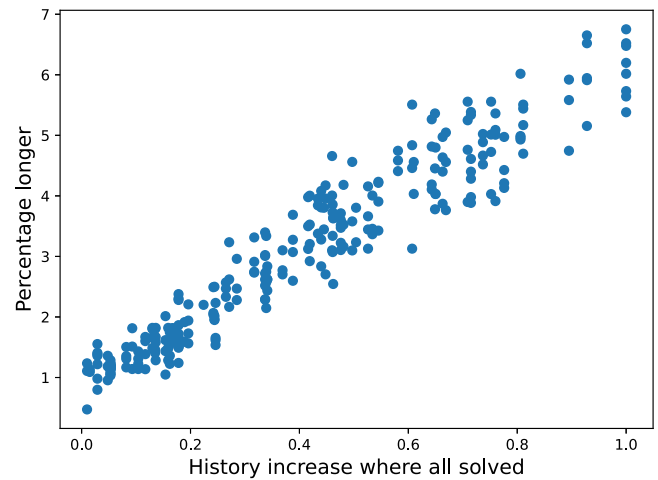


Fig. 13. Percentage longer than the optimal length versus history increase, every dot represents one run of the algorithm for a fixed parameter choice on a fixed test set. (Longer is worse.).

The normalized time expresses how much longer this parameter choice needed in comparison to the shortest time needed for this test set. For example, a value of 2 indicates that this parameter choice needs twice as long as the fastest parameter choice on the same test set.

When plotting the normalized time versus history increase as seen in Fig. 14, it is striking that small values have high running times. This agrees with the earlier observation that too small of a value for history increase leads to either not finding solutions or only finding them after many iterations. This leads to high runtimes. Then a less clear trend emerges, where the fastest time seems to occur for values of history increase going from 0.17 to 0.47. Values outside this range seem undesirable timing-wise.

If we look at both solution quality and time, we can see that there is a trade-off. Small values (0.02-0.10) for history increase give high-quality solutions but at the cost of high runtimes. Vice versa, higher values (0.10-0.47) result in faster solutions and a loss of solution quality.

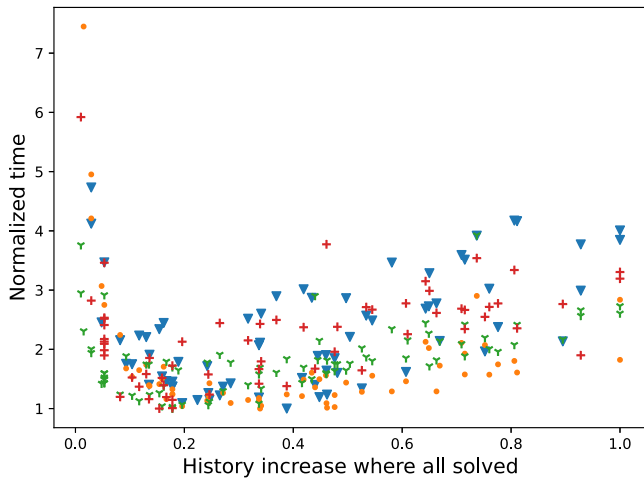


Fig. 14. Normalized time versus history increase, different markers indicate different test sets. Time is normalized per test set, see (2).

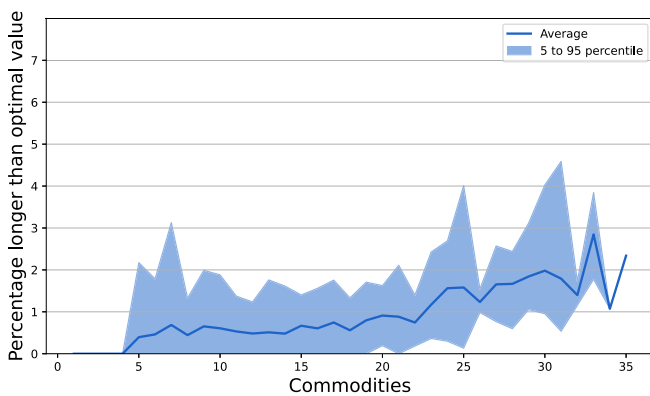


Fig. 15. Performance of single-convergence with parameter values according to differential evolution. History increase: 0.053, ε : 0.335. Note that, whereas this is similar to Fig. 9, their multi-convergence performance is shown.

Now, we use the tuned parameter value of history increase from differential evolution and plot the total performance on all test sets that have optimal solutions. This is the performance of single-convergence. A solution is found for all of the approximately 450 different feasible problems. The resulting performance is plotted in Fig. 15. Above 30 commodities, there aren't many test sets where an optimal solution can be found within reasonable time, hence the narrowing 5 to 95 percentile band. A solid performance is shown, with an average performance that stays below 3%. Looking at the time needed to get these solutions, our algorithm scales a lot better than the exponential scaling of the integer linear program, as shown in Fig. 16.

C. Effects of Cost-Based Rip-Up

We now look towards multi-convergence. After one solution is found, the algorithm does not terminate. Instead, it rips up certain paths and resets all congestion. Two parameters are an obvious choice to tune for this restart: at which percentage $R\%$ does the algorithm rip up and how many times n does it look for a solution? We call n the maximum convergence number. Looking at the previous section, we want to tune the history increase HI

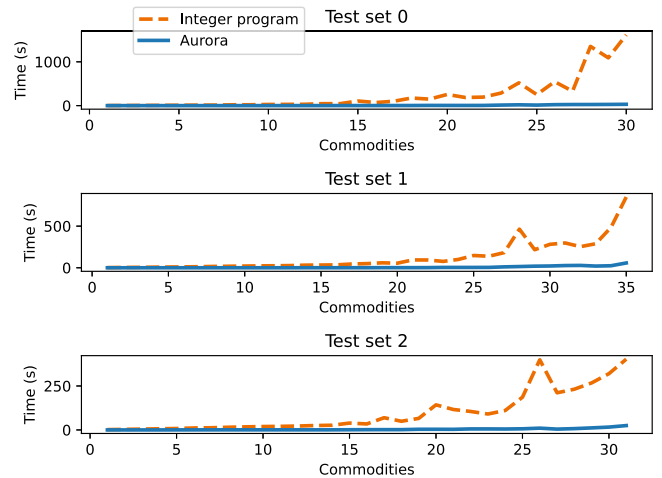


Fig. 16. Time comparison of an integer linear program solver to a single convergence of Aurora on different test sets. Similar to Fig. 10, but towards the higher number of commodities, the time needed is even lower by an order of magnitude.

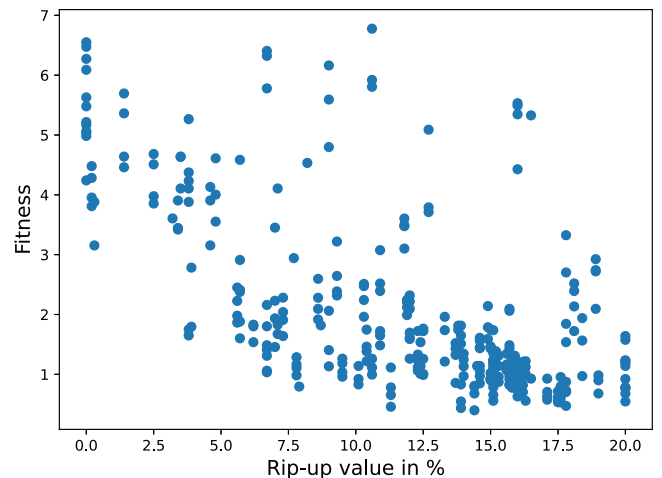


Fig. 17. A downward trend can be seen between rip-up value and fitness with 12% to 18% giving good values. (Lower fitness is better).

and introduce a fourth parameter. It seems logical to slightly scale the history increase by a value v after each solution.

The effect of multi-convergence is now the following: after one solution is found, the algorithm restarts by ripping up all paths that are $R\%$ longer than their shortest path in the mesh. It changes the history increase HI to $v \cdot HI$ and does this until n solutions are found. The first time HI is used, after the first solution $v \cdot HI$, after the second solution $v^2 \cdot HI$, etc.

Firstly, we look at the rip-up value. The idea of ripping some parts up, but not all, is that the previous solution contains good parts and bad parts. The good parts consist of paths that are optimal or close to optimal, whereas the bad parts can still be improved. As seen in Figs. 17 and 18, choosing a rip-up value that is too low both causes worse fitness and greater time usage. A value of around 15% gives the most consistent performance.

Restarting has a clear effect on performance, and also the number of times matters. As seen in Fig. 19. Restarting only once already improves fitness on average by 33%. Restarting

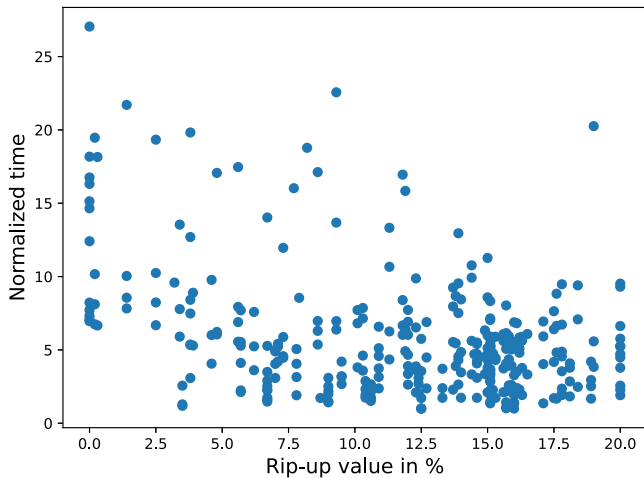


Fig. 18. Rip-up value versus normalized time. No clear trend emerges here, but a value of 0%, which means ripping up everything, seems to be undesirable. Time is normalized per test set, see (2).

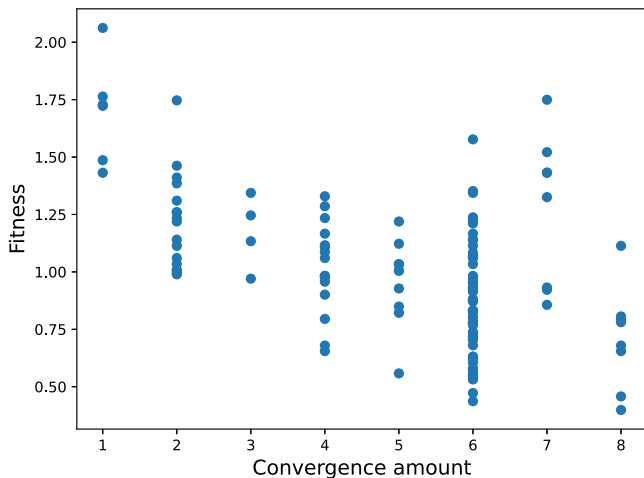


Fig. 19. Effect of the number of restarts on fitness. Restarting more is better but with diminishing returns.

more keeps improving fitness but with diminishing returns. Differential evolution ended with a maximum convergence value of 6, thus for the allotted number of iterations, there is no meaningful value in restarting more than 6 times.

D. Preprocessing Effects

Preprocessing changes the initial weights by a small value ε , see Section III-D. This stimulates paths to not use the shortest paths that share edges with the shortest paths of other commodities.

The effect of preprocessing seems to be less outspoken, however, whereas only a fourth of all runs are on a test set without preprocessing, it does account for a third of all situations where not all feasible instances were solved. Besides this, there are noticeably more unsolved instances when not using preprocessing. As depicted in Table II, using preprocessing reduces the chance of not finding a solution by around 25%. Not using

TABLE II
NUMBER OF TEST SETS WHERE AT LEAST ONE FEASIBLE PROBLEM INSTANCE REMAINS UNSOLVED

	Unsolved	Total	% unsolved
With preprocessing	28	230	12%
W/o preprocessing	15	88	17%

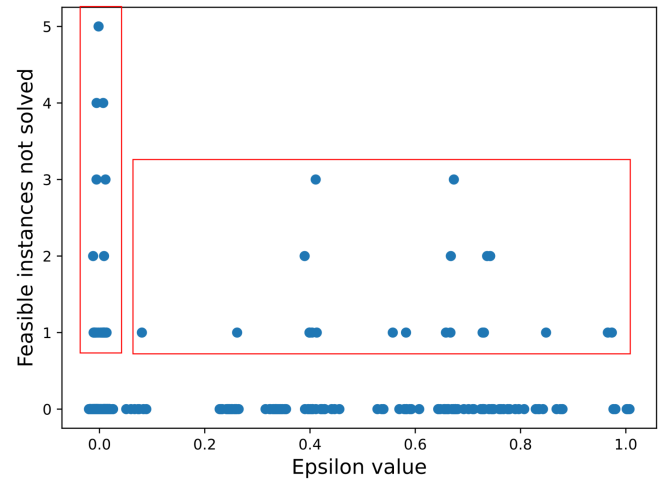


Fig. 20. All test sets and the number of unsolved problem instances mapped out by value for ε . The left box is all test sets where no preprocessing was done, a noticeably higher number of instances remains unsolved.

preprocessing not only increases the chance of not finding a solution to a problem instance in a test set, but it also increases the number of problem instances it did not find a solution for in a single test set, see Fig. 20.

E. Number of Iterations

The maximum number of iterations is an important parameter to tune. With the current fitness function, a higher value for the number of iterations is always better. More iterations can only improve the number of feasible instances solved and the quality of solutions. Because of this, a second goal is put forward that the maximum number of iterations should strive for. This should be an estimation by which the algorithm should have found a solution. If it does not find a solution by then, the algorithm concludes that no solution exists. This estimation will depend on the number of commodities. Fig. 21 plots the maximum number of iterations the algorithm needs over all problem instances by the number of commodities. Here, at most, 65 iterations are needed per commodity. This peak of 1618 iterations is reached for one problem instance with 25 commodities. This motivates the following rule: if there are n commodities and Aurora has not found a solution after $80 \cdot n$ iterations, it halts. This rule should provide a high enough safety margin to find a solution while being proactive in deciding when there is no solution. If Aurora finds a solution, then the maximum number of iterations is increased by 50%, giving it plenty of time to improve its results through multi-convergence.

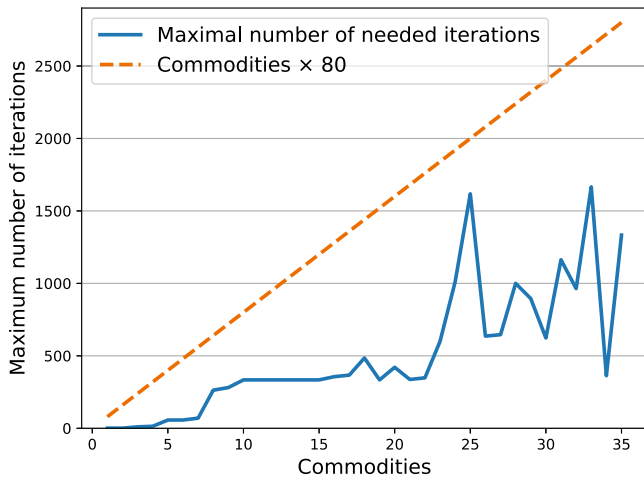


Fig. 21. The maximum number of iterations that is needed to find a solution by number of commodities over all problem instances.

IX. CONCLUSION

We have demonstrated a high-performance routing algorithm specifically adapted to the unique topology of photonic integrated circuits. Aurora produces close-to-optimal results in reasonable time and does this so consistently over a wide variety of different problem instances. On average, the results are within 2% of the optimal solution and it finds these several orders of magnitude faster than an integer linear program solver. Starting from hexagonal meshes with radius $r = 8$, the runtime of these solvers easily goes up to hours. On higher radii, it is infeasible to expect results in reasonable time on normal hardware, whereas Aurora still produces good solutions within seconds to minutes. The effect of the different parameter choices was studied and more interesting values were highlighted. We implemented an optimizer to attain good parameter values and validated the obtained parameter values on a wide variety of test sets.

There are still some aspects where future work can be done. Right now, the algorithm only terminates once it reaches the maximum number of iterations. However, by analyzing the progress of the algorithm, for example, in terms of available routing resources and current iteration, it might be possible to detect early if there is no feasible solution. This could greatly reduce runtime on unroutable problem instances.

Couplers also have a coupling mode, meaning that they can split the light over both waveguides. This allows for single-source multi-target commodities. The proposed algorithm is designed to be fairly easily adaptable to accommodate this problem, but further research would be required to devise a strategy to properly implement this.

REFERENCES

- [1] D. Pérez, I. Gasulla, and J. Capmany, "Programmable multifunctional integrated nanophotonics," *Nanophotonics*, vol. 7, no. 8, pp. 1351–1371, 2018.
- [2] W. Bogaerts and A. Rahim, "Programmable photonics: An opportunity for an accessible large-volume PIC ecosystem," *IEEE J. Sel. Topics Quantum Electron.*, vol. 26, no. 5, Sep./Oct. 2020, Art. no. 8302517.
- [3] N. Harris et al., "Linear programmable nanophotonic processors," *Optica*, vol. 5, no. 12, pp. 1623–1631, 2018.

- [4] D. P. López, "Programmable integrated silicon photonics waveguide meshes: Optimized designs and control algorithms," *IEEE J. Sel. Topics Quantum Electron.*, vol. 26, no. 2, Mar./Apr. 2020, Art. no. 8301312.
- [5] X. Chen, P. Stroobant, M. Pickavet, and W. Bogaerts, "Graph representations for programmable photonic circuits," *J. Lightw. Technol.*, vol. 38, no. 15, pp. 4009–4018, Aug. 2020.
- [6] A. López, D. Pérez, P. DasMahapatra, and J. Capmany, "Auto-routing algorithm for field-programmable photonic gate arrays," *Opt. Exp.*, vol. 28, no. 1, pp. 737–752, 2020.
- [7] X. Chen and W. Bogaerts, "A graph-based design and programming strategy for reconfigurable photonic circuits," in *Proc. IEEE Photon. Soc. Summer Top. Meeting Ser.*, 2019, pp. 1–2.
- [8] F. V. Kerchove, X. Chen, D. Colle, W. Bogaerts, and M. Pickavet, "Adapting routing algorithms to programmable photonic circuits," in *Proc. Eur. Conf. Opt. Commun.*, 2022, Paper We5.19. [Online]. Available: <https://opg.optica.org/abstract.cfm?URI=ECEOC-2022-We5.19>
- [9] L. McMurchie and C. Ebeling, "Pathfinder: A negotiation-based performance-driven router for FPGAs," in *Proc. 3rd Int. ACM Symp. Field-Programmable Gate Arrays*, 1995, pp. 111–117.
- [10] Y.-J. Chang, Y.-T. Lee, and T.-C. Wang, "NTHU-route 2.0: A fast and stable global router," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2008, pp. 338–343.
- [11] K. E. Murray, S. Zhong, and V. Betz, "AIR: A fast but lazy timing-driven FPGA router," in *Proc. IEEE 25th Asia South Pacific Des. Automat. Conf.*, 2020, pp. 338–344.
- [12] M. Pan and C. Chu, "FastRoute 2.0: A high-quality and efficient global router," in *Proc. Asia South Pacific Des. Automat. Conf.*, 2007, pp. 250–255.
- [13] K. E. Murray et al., "VTR 8: High-performance CAD and customizable FPGA architecture modelling," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, no. 2, pp. 1–55, 2020.
- [14] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.
- [15] T. Eilam-Tzoref, "The disjoint shortest paths problem," *Discrete Appl. Math.*, vol. 85, no. 2, pp. 113–138, 1998. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S0166218X97001212>
- [16] L. Perron and V. Furnon, "Or-tools," Google, Aug. 11, 2022. [Online]. Available: <https://developers.google.com/optimization/support/cite>
- [17] M. E. H. Pedersen, "Tuning & simplifying heuristical optimization," Ph.D. dissertation, Univ. Southampton, School of Engineering Sciences, Southampton, U.K., 2010.

Ferre Vanden Kerchove received the M.Sc. degree in mathematics from Ghent University, Ghent, Belgium, in 2021. He is currently working toward the Ph.D. degree in computer science with IDLab, Ghent University - imec, Leuven, Belgium. His research interests include algorithms, graph theory, logic, and computability.

Xiangfeng Chen (Student Member, IEEE) received the M.Sc. degree from the Center for Optical Materials Science and Engineering Technologies, COMSET, Clemson University, Clemson, SC, USA, in 2018 by carrying out research on array waveguide gratings for III-V on silicon nitride hybrid integration. He is currently working toward the Ph.D. degree with the Photonic Research Group, Ghent University - IMEC, Ghent, Belgium. His research interests include large-scale programmable photonic circuits at both circuit and component levels. He enjoys the interdisciplinary nature of photonic engineering.

Didier Colle (Member, IEEE) received the M.Sc. degree and Ph.D. degree in electrotechnical engineering from Ghent University, Belgium, in 1997 and 2002 respectively. Since 2022, he has been a Senior Full Professor with Ghent University. He has been an Associate Professor since 2011 and a Full Professor since 2014 with the same university. He is co-responsible for the research cluster on network modeling, design, and evaluation (NetMoDeL) inside the IMEC IDlab research group. This research cluster deals with fixed internet architectures and optical networks, green ICT, the design of network algorithms, and techno-economic studies. He has authored or coauthored in more than 500 international journal and conference articles and has resulted in more than 20 Ph.D. degrees in his research field, which include inside international (mainly European), national, and bilateral research projects together with the industry.

Wouter Tavernier received the B.S. and M.S. degrees in computer science from Ghent University, Ghent, Belgium, in 2002, and the Ph.D. degree from Ghent University, in 2012, on reliable routing and switching. He joined the Internet-Based Communications Networks Group (which became part of IDLab in October 2016) of Ghent University in 2006 as a Researcher on Carrier Ethernet. He is currently a Professor with Ghent University, where he teaches courses on computer networks. He has authored or coauthored in more than 100 scientific publications in his research field, which include performance and resource optimization aspects of network function virtualization and deterministic networking. This work is performed in the context of European projects such as H2020 5G-CHAMPION, NGPAAS, SONATA-NFV, and 5G TANGO.

Wim Bogaerts (Fellow, IEEE) received the Ph.D. degree in the modeling, design, and fabrication of silicon nanophotonic components from Ghent University, Ghent, Belgium, in 2004. During this work, he started the first silicon photonics process on imec's 200 mm pilot line, which formed the basis of the multi-project-wafer service ePIXfab. In 2014, he co-founded the spin-off company Luceda Photonics to further develop unique software solutions for silicon photonics design, using the IPKISS design framework. Since 2016, he has been again a full-time Professor with Ghent University, looking into novel topologies for large-scale programmable photonic circuits, supported by a consolidator grant from the European Research Council (ERC). His research interests include the challenges for large-scale silicon photonics: Design methodologies and controllability of complex photonic circuits, telecommunications, information technology, and applied sciences. He is a Senior Member of Optica and SPIE.

Mario Pickavet (Senior Member, IEEE) received the M.Sc. and Ph.D. degrees in electrical engineering from Ghent University, Ghent, Belgium, in 1996 and 1999, respectively. Since 2000, he has been a Professor with Ghent University, where he is teaching courses on discrete mathematics and network modeling. He is co-leading the research cluster on network modeling, design, and evaluation (NetMoDeL). He has authored or coauthored about 500 international publications, both in journals and proceedings of conferences. He is coauthor of the book *Network Recovery: Protection and Restoration of Optical, SONET-SDH, IP, and MPLS*. His main research interests include fixed internet architectures and optical networks, green ICT, and the design of network algorithms. In this context, he is currently involved in several European and national projects. He is Holder of a Bronze Medal at the International Mathematical Olympiad (Sweden, 1991).