Solving the ACM ICPC Problem 'Wiring Assistant'

A Reduction Approach for Huge Grid Graphs

Samuel Füßinger
samuel.fuessinger@student.uni-tuebingen.de
Eberhard Karls Universität
Tübingen, Baden-Württemberg, Germany

ABSTRACT

In this paper, we address the 2006 *ACM International Collegiate Programming Contest* problem 'Wiring Assistant', which centers on finding the minimum path cost in potentially enormous grid graphs. We present an approach that can solve this problem efficiently by reducing the input grid graph through the removal of identical neighboring columns or rows, leading to a much more manageable size where pathfinding algorithms can be run in milliseconds. We prove that our reduction algorithm guarantees a worst-case grid size of 301×303. Additionally, we provide a detailed analysis of both the space and time complexity of our approach and examine its scalability limitations for the generalized problem without constraints on the maximum grid size and the number of wires.

Our full implementation is publicly available at https://github.com/Samsu-F/Wiring_Assistant.

KEYWORDS

Pathfinding, Lattice path, A*-algorithm, Grid graph, ACM ICPC, Wiring Assistant

1 INTRODUCTION

1.1 About ACM ICPC

The *ACM International Collegiate Programming Contest* (ICPC) is a global programming contest that challenges college students to efficiently solve algorithmically complex problems. Hosted annually by the *Association for Computing Machinery* (ACM) since 1970, it aims to foster "collaboration, creativity, innovation, and the ability to perform under pressure" [4].

1.2 Problem

In 2006, the ACM ICPC included a problem called 'Wiring Assistant' [8]. In this problem, a circuit board composed of a grid of horizontal and vertical tracks is given, as well as a number of pre-existing wires running on these tracks. Additionally, two points that must be connected by a path on the grid are given (see Figure 1 for an exemplary visualization).

Wires are strictly horizontal or vertical, running only on a single track. Horizontal and vertical wires may overlap in a single grid point. Two horizontal wires or two vertical wires may only overlap in their endpoints.

The tracks are numbered, starting from zero for the leftmost column and the bottom row. The wires are identified by the coordinates of their lower left endpoint followed by the upper right one. For example, Figure 1 shows a possible scenario with the light green wires described by `4 0 4 6  0 5 6 5  9 1 9 6  7 4 7 7  8 6 8 8  4 6 6 6  8 8 10 8`. The light blue points that need to be connected are described by the coordinates `7 8` and `6 1`.

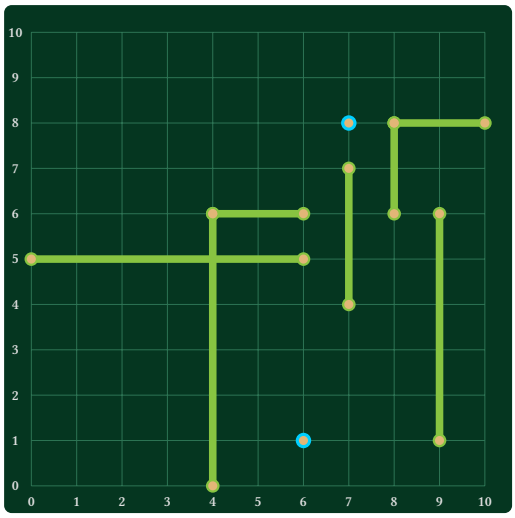

Figure 1: An example scenario with seven wires

For any given input, the objective is to calculate the minimum number of overlaps with existing wires for any path connecting the two given points. Overlaps with wire endpoints count as one overlap, just like overlaps with the middle of a wire. A path may not run on top of another wire but may overlap in single points.

For example, the purple path in Figure 2 connects the points with two overlaps, while the light gray path connects them with only one overlap. There is no cheaper path connecting them with less than one overlap, making the integer '1' the correct solution for this scenario. Only the minimum number of overlaps needs to be found, it is not necessary to output the path.

The red path in Figure 2 is invalid because it runs on top of the wire `7 4 7 7`.

There are limits to the number of wires and the grid size. The grid size, which is both the width and the height of the grid, of any given problem instance is a positive integer less than 10^9. The number of wires is a positive integer less than 100.

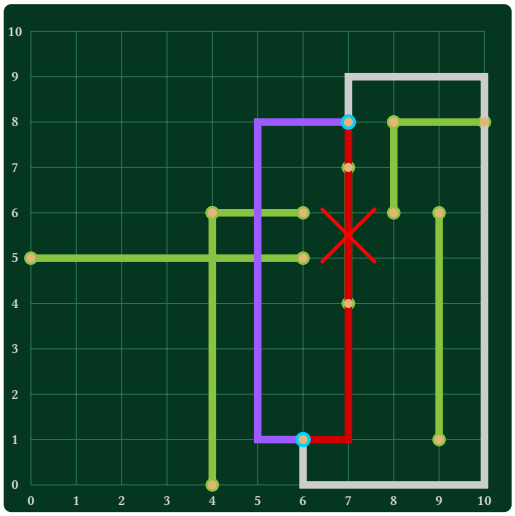

Figure 2: Valid (light gray, purple) and invalid (red) paths for the scenario shown in Figure 1

1.3 Input Format

The input is given in the form of three lines of plain text, containing decimal integers separated by spaces. The first line consists of the number of existing wires, and the size of the circuit board[1]. The second line contains — in groups of four numbers per wire — the coordinates of all wire endpoints. For each wire, the x- and y-coordinates of the lower left endpoint are given first, followed by the x- and y- coordinates of the upper right endpoint. The third line contains the coordinates of the two points that need to be connected. To give an example, the problem instance shown in Figure 1 could be described by the following input.

```
7 11
4 0 4 6 0 5 6 5 9 1 9 6 7 4 7 7 8 6 8 8 4 6 6 6 8 8 10 8
7 8 6 1
```

2 OUTLINE OF OUR SOLUTION

2.1 Grid Graph Model

It is quite natural to think of this problem as a graph problem. Specifically, it can be interpreted as a problem on a square grid graph[2], where each point of the circuit board corresponds to a node of the graph. Each node has edges to its four immediate neighbors[3], except the nodes on the border and the corners, which have only three or two edges. Each node is assigned a cost equal to the number of existing wires ending on or running through the corresponding point. For example, for the case shown in Figure 1, the node $(4, 0)$ is assigned a cost of 1, and $(4, 5)$ has a cost of 2.

We remove the edges of existing wires from the graph because an edge can be used for a new path if and only if there is no wire already present.

[1] i.e., the number of horizontal tracks, which is equal to the number of vertical tracks
[2] A square grid graph is a lattice graph with square tiling. It does not imply that the graph itself is a square.
[3] i.e., $(x + 1, y)$, $(x - 1, y)$, $(x, y + 1)$, and $(x, y - 1)$

Figure 3 illustrates this, showing the equivalent graph for the scenario depicted in Figure 1.

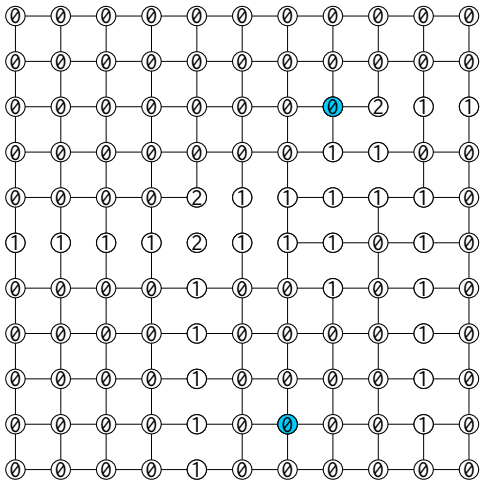

Figure 3: The equivalent graph for the scenario shown in Figure 1

We define the cost of a path as the sum of the node costs for all nodes on the path. Since the node cost is the number of intersecting wires for a node, it is easy to see that the cost of a path is equal to the number of overlaps with existing wires for the corresponding path on the circuit board. Consequently, the cost of the cheapest path between two given nodes is equal to the minimum number of overlaps for any connection of the given points on the printed circuit board, which is the number we want to calculate.

2.2 Naïve Approach

An intuitive subsequent approach would be to use a pathfinding algorithm like Dijkstra's algorithm or A* to calculate the cost of the cheapest path. However, the width and height of the input graph are only limited to be less than 10^9, meaning the graph could have almost 10^{18} nodes. Figure 4 shows an example where this would be a serious problem. With a width and height of 999, 999, 999 nodes

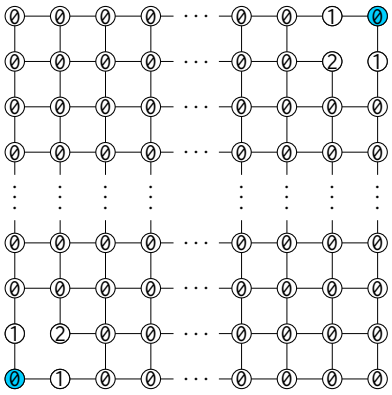

Figure 4: A graph for which the naïve approach is infeasible

and the wires `0 1 1 1` and `1 0 1 1`, as well as equivalent wires in the opposite corner, the "distance" — i.e., the path cost — from any of the nodes in the middle to a corner is strictly less than the minimum path cost for a connection of the corner nodes marked in blue. Therefore, if one were to calculate the cost of connecting the marked corners using Dijkstra's algorithm or A*, almost all of the nearly 10^{18} nodes would have to be visited. Even with the very optimistic estimation that visiting a node takes only a single CPU clock cycle, this would still result in a runtime of 7.9 core years on a 4 GHz CPU! So clearly, this naïve approach is not feasible.

2.3 Our Reduction Approach

Before any processing can be done, the input needs to be parsed. We parse line by line, number by number and store them into a simple struct with fields for the number of wires, the size, and the points to be connected; and an array field in which the coordinates of the wire endpoints are held.

Unlike the naïve approach, our reduction approach then leverages the low number of wires to drastically decrease the problem size.

The number of wires is limited to a maximum of 99, so the vast majority of the circuit board tracks on large inputs are empty. This gives us an opportunity to greatly reduce the graph to a manageable size, without changing the minimum cost between nodes. The idea is relatively simple: identical neighboring columns or rows, that do not contain a wire endpoint, are redundant and can be combined into only one column or row. Figures 5 and 6 illustrate this; the minimum cost to connect any two points in Figure 5 is the same as the one for the corresponding points in Figure 6.

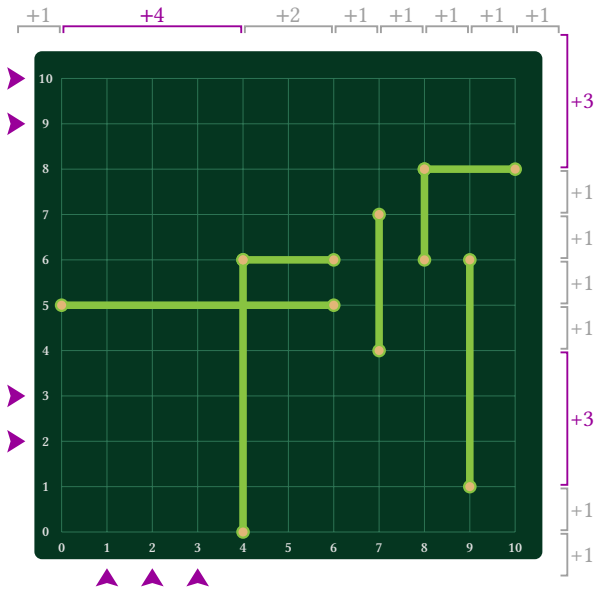

Figure 5: A problem instance with redundant rows and columns (indicated by the purple pointers)

The implementation of this idea is straightforward and should be run before transforming the input into a graph representation as

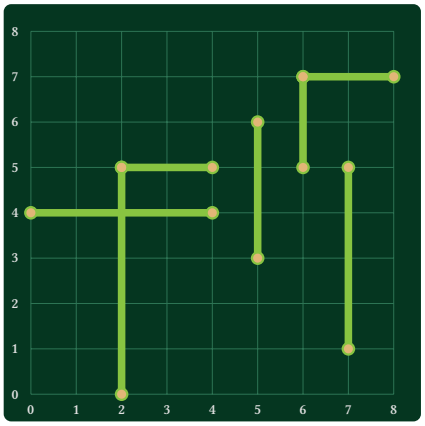

Figure 6: The same problem instance as in Figure 5, with the redundant tracks merged.

outlined in Section 2.1. It can be accomplished with the help of two arrays, one containing the x-coordinates and the other one containing the y-coordinates of all wire endpoints. The arrays are sorted and subsequently iterated over, identifying differences d between consecutive values equal to or greater than three. As illustrated in Figure 5, such a difference of $d \geq 3$ is found if and only if there are two or more consecutive columns or rows between these coordinates that do not contain any wire endpoints. By subtracting $d - 2$ from all of the remaining coordinates after the gap, the redundant columns or rows are effectively removed (cf. Figure 6). Attention should be paid to ensure that potential redundancies at the edges of the circuit board are also removed and that the given points to connect are placed back on the correct coordinates. Our implementation ensures both of these points rather trivially by including the width and height of the circuit board, as well as the coordinates of the points to connect, in the respective array. This simple and computationally cheap solution does, however, come at the cost of a slightly larger worst-case size of the reduced circuit board (see Section 2.4 for details).

After reducing the input problem instance as described, it is transformed into a graph representation (as explained in Section 2.1) that allows for the efficient execution of the pathfinding algorithm. We chose to represent the graph with two matrices: one saving the cost for each node, while the other contains four-bit bit-vectors that indicate whether there is an edge in the positive and negative x- and y-directions for each node. For example, the least significant bit represents the edge in the positive x-direction, so `neighbors[x][y]` has this bit set if and only if there is an edge connecting the node at (x, y) to the node at $(x + 1, y)$.

As the pathfinding algorithm, we opted for the A*-algorithm, since knowing the coordinates of both the start and end node allows us to gauge in which direction the path search appears most promising. As the metric of an arbitrary node to the start node, we use a tuple $(cost, distance)$, where the cost is the sum of all node costs traversed on the path as defined previously. The distance is simply the number of edges on the path. Metrics are compared

lexicographically, i.e., $(c_1, d_1) < (c_2, d_2)$ if and only if $c_1 < c_2$ or $c_1 = c_2 \wedge d_1 < d_2$[4].

For the heuristic function that estimates the metric for a node n to the goal g, we use $h(n) = (0, d(n, g))$, where $d(n, g)$ is the Manhattan distance from n to the goal node. This means we estimate that reaching the goal is possible without crossing any more wires and in the shortest distance possible. Clearly, this heuristic is admissible, i.e., it cannot overestimate the actual metric to reach the goal. It is also consistent, meaning that traveling one node further can never decrease the heuristic by more than the metric increase that resulted from the step just taken.

A notable alteration from the conventional A*-algorithm is that we do not record the predecessor nodes, since we are not interested in reconstructing the shortest path, but only in its cost[5].

2.4 Effectiveness of the Reduction

Note: The calculations in this section are based on the simple solution (as described in Section 2.3) that includes the width and height of the circuit board, as well as the coordinates of the points to connect, in the arrays used to identify redundant tracks. This is also the version implemented in the code belonging to this paper. A marginal improvement of the worst-case size is possible with a more sophisticated algorithm, but we chose to stick with this simpler, albeit slightly worse, version for ease of understanding and consistency of paper and implementation.

Since the previously described reduction algorithm removes redundant tracks, i.e., neighboring columns or rows that both do not contain any given points[6], there cannot be two neighboring rows or columns that both do not contain any given points after running the reduction. Therefore, the worst-case size of the circuit board after the reduction is a width of double the number of unique x-coordinates plus one, and analogously for the height with the number of unique y-coordinates. For example, in Figure 7, there are five unique x-coordinates and five unique y-coordinates, each of them having had one or more empty rows and columns to all sides before the reduction. This results in a width and height of $2 \cdot 5 + 1 = 11$, and thus $11^2 = 121$ nodes, which is the worst-case size for the case of two wires.

With this knowledge of the worst-case width and height, we can infer that the worst-case number of nodes is $width \cdot height = (2 \cdot |unique\ x\text{-}coordinates| + 1) \cdot (2 \cdot |unique\ y\text{-}coordinates| + 1)$.

Since each wire is either horizontal or vertical (and not diagonal), either its x-coordinates or its y-coordinates are identical, meaning each wire can have up to three unique coordinate components. The start and end point can also contribute up to two unique coordinate components each. We know that there are no more than 99 wires, therefore, we have at most $3 \cdot 99 + 2 \cdot 2 = 301$ unique components.

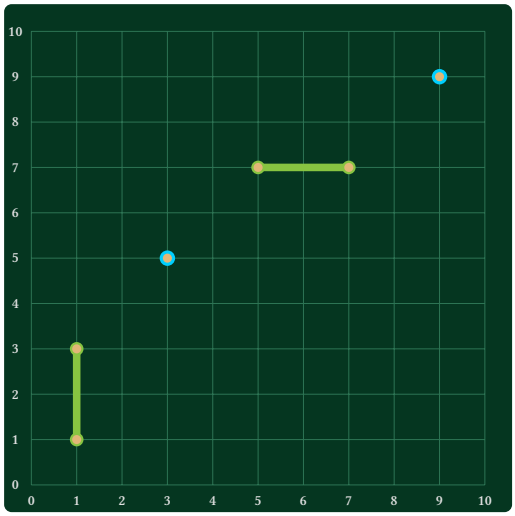

Figure 7: The worst case size after the reduction for the case of two wires.

We can find an upper limit for the product $width \cdot height$ by looking at the sum $width + height$ first:

$$
\begin{aligned}
&width + height \\
&= (2 \cdot |unique\ x\text{-}coordinates| + 1) + (2 \cdot |unique\ y\text{-}coordinates| + 1) \\
&= 2 \cdot (|unique\ x\text{-}coordinates| + |unique\ y\text{-}coordinates|) + 2 \\
&= 2 \cdot (|unique\ components|) + 2 \\
&\leq 2 \cdot 301 + 2 \\
&= 604
\end{aligned}
$$

Since the product of any two natural numbers is less than or equal to the square of their arithmetic mean[7], it follows logically that $width \cdot height \leq \left(\frac{width+height}{2}\right)^2 = \left(\frac{604}{2}\right)^2 = 302^2 = 91204$. As in the worst-case both $width$ and $height$ are odd numbers, the true upper bound on the number of nodes is actually $301 \cdot 303 = 91203$ nodes, which is many orders of magnitude smaller than the possibly close to 10^{18} nodes we had before the reduction.

More generally speaking, a problem instance with w wires has at most $3 \cdot w + 4$ unique coordinate components, so after the reduction, both the width and the height are in $O(w)$ and the resulting graph will not have more than $\left(\frac{2 \cdot (3 \cdot w + 4) + 2}{2}\right)^2 = (3w + 5)^2$ nodes.

3 PERFORMANCE ANALYSIS

Given that the original problem as set in the competition has upper limits on both the number of wires and the size of the circuit board, the number of unique problem instances is finite. Consequently, the complexity (both in space and time) of any algorithm solving it is in $O(1)$ as there is a constant time in which any of the finite number of possible inputs can be solved.

Because this does not provide meaningful insights, in this section, we analyze the complexity of a generalized version of the problem without limits on the size of the circuit board and the number of

[4] This comparison can be efficiently implemented by interpreting the tuple as an unsigned integer, wherein the cost occupies the most significant bits and the distance occupies the least significant bits.

[5] Our implementation offers an optional `-p` flag, with which the predecessor nodes are tracked and the path is reconstructed and displayed. However, this is only for demonstration and not necessary for the ACM ICPC problem.

[6] By *given points*, we mean the wire endpoints, the start, and the goal.

[7] a direct consequence of the AM–GM inequality

wires. From now on, unless marked otherwise, the discussed complexities are worst-case complexities in big O notation dependent on the grid size s and the number of wires w.

As the grid size s grows arbitrarily large, so does the number of digits of s, albeit only logarithmically. To store s, you need at least $\log_2(s+1)$ bits of memory, which affects the space and time required if s grows arbitrarily large. Since the maximum value for coordinates is obviously limited by the grid size, their number of bits is already accounted for in $\log(s)$. As shown in Section 2.4, the size after the reduction step is in $O(w)$, meaning its number of bits needed to store the size is in $O(\log(w))$. Note that $O(\log(w)) \subset O(\log(s))$, since $w < 2s^2$ (as there cannot be more wires than edges). This automatically introduces a factor of $\log(s)$ for any piece of code that processes coordinates or the grid size before or during the reduction step, and a factor of $\log(w)$ for any code handling coordinates or the size after the reduction.

3.1 Complexity Step by Step

Both the time and space complexity of parsing the input are proportional to the number of wires w multiplied with the aforementioned $\log(s)$ factor for the coordinates' growing number of digits. Therefore, they are in $O(w \cdot \log(s))$, which is another way of saying they are linearly bounded by the number of characters in the input string.

The reduction step requires sorting the wire endpoint coordinates, which can be achieved in $O(w \cdot \log(w))$ time with linear space complexity[8]. All other subtasks of the reduction step — which are splitting the coordinates into the two arrays and later iterating over the sorted arrays to identify gaps — can be accomplished in linear space and time[9]. This leaves us with a time complexity of $O(w \cdot \log(w) \cdot \log(s))$ and a space complexity of $O(w \cdot \log(s))$ for the reduction step[10].

Building the graph requires setting up two matrices, one storing the cost of each node and the other one storing whether there is an edge for each node adjacent in the grid. This can be achieved by iterating over all wires, for each wire iterating over all nodes it passes through, while incrementing the cost for each node and deleting each edge it passes through. As wires cannot be stacked, we can visit each node at most four times and each edge at most once. Therefore, both the time and space complexity of this step are in $O(|E|) = O(|V|)$[11], where V is the set of all nodes and E is the set of all edges. In Section 2.4, we saw that the number of nodes after the reduction is quadratically bounded by w and it is obviously not greater than the original number of nodes[12], giving us a space and time complexity of $O\big(\min(w^2, s^2) \cdot \log(w)\big)$ for constructing these matrices.

The A*-algorithm performs a loop iteration for each metric–node pair queued. Per loop iteration one pair has to be dequeued and up to three neighbors get enqueued into the priority queue, taking $O(\log(n) \cdot \log(w))$ time[13], where n is the number of elements in the queue. Each node cannot possibly get enqueued more times than the number of edges it has, i.e., not more than four times. This means there are $O(|V|)$ many loop iterations, each taking a time of $O(\log(|V|) \cdot \log(w))$. Just like in the previous step, we can substitute $|V|$ with $\min(w^2, s^2)$, resulting in a time of $O\big(\min(w^2, s^2) \cdot \log(w^2) \cdot \log(w)\big) = O\Big(\min(w^2, s^2) \cdot \log^2(w)\Big)$ for A*.

A*'s space is in $O(|V| \cdot \log(w)) = O\big(\min(w^2, s^2) \cdot \log(w)\big)$, because both the priority queue and the matrix that is used to save the metrics need to save a constant number of metrics (each of size $\log(w)$) per node.

Parsing	$O(w \cdot \log(s))$
Reduction	$O(w \cdot \log(w) \cdot \log(s))$
Building the Graph	$O\big(\min(w^2, s^2) \cdot \log(w)\big)$
A*	$O\Big(\min(w^2, s^2) \cdot \log^2(w)\Big)$
Total	$O((w \cdot \log(w) \cdot \log(s) + \min(w^2, s^2) \cdot \log^2(w))$

Table 1: Time complexity

Parsing	$O(w \cdot \log(s))$
Reduction	$O(w \cdot \log(s))$
Building the Graph	$O\big(\min(w^2, s^2) \cdot \log(w)\big)$
A*	$O\big(\min(w^2, s^2) \cdot \log(w)\big)$
Total	$O(w \cdot \log(s) + \min(w^2, s^2) \cdot \log(w))$

Table 2: Space complexity

Table 1 and Table 2 give an overview of the complexities discussed. Of course, the total complexity of our solution is the sum of all the steps. It is particularly noteworthy that a huge grid size s has very little effect on the required space and time, affecting only the parsing and reduction steps negligibly.

4 DISCUSSION OF OUR IMPLEMENTATION

4.1 Programming Language

We chose to implement our solution in the C programming language. Mainly, because C is one of the fastest, if not *the* fastest widely used general-purpose language today [1][3][6][14].

C's language design, most notably pointers, enabled some elegant solutions. For example, in C it is possible to iterate over an array of pointers, reading and changing the values at the locations pointed to. This is helpful if the values to be read or changed are different fields

[8] for example with heapsort or mergesort
[9] for an arbitrary but fixed s, otherwise there is the additional $\log(s)$ factor
[10] In our implementation, we use the C standard library function *qsort*, which — despite its name — has no complexity guarantees. The actual space and time complexities are implementation specific [2]. The latest GNU C Library version at the time of writing uses mergesort with heapsort as a fallback [7].
[11] ignoring the log-factor for the growing number of digits needed to identify the nodes
[12] Note that for large values of w, $O(s^2)$ is a tighter constraint than $O(w^2)$ since w itself is in $O(s^2)$.
[13] The $\log(w)$-factor is once again there because of the growing number of digits.
[14] This decision was also influenced by our personal preference and familiarity with C.

of different kinds of structs because it would not be possible to fill an array with different data types. To give an example, the `reduce` function in our implementation makes use of this possibility.

However, C also has some drawbacks. The lack of built-in high-level data structures, such as the priority queue needed for the A* algorithm, made implementing our solution much more time-consuming than we suspect it would have been in other languages.

In hindsight, while C was an appropriate choice, the extremely low execution times achieved were not really necessary for the purpose of demonstrating our algorithm. Thus, using a higher-level programming language with greater ease of use would also have been an appropriate option, despite the performance drawbacks.

4.2 Scalability and its Limits

For all problem instances of the original problem (i.e., grid size $s < 10^9$ and number of wires $w \leq 99$), our implementation solves each problem in at most a few milliseconds. However, when considering the generalized problem without these constraints, we run into the limits of data types and available time and memory.

From the moment of parsing, through the reduction until the problem instance is transformed into the graph representation, the width and height of the grid, as well as all the coordinates, are stored in the `int_fast32_t` data type, defined in the C standard library *stdint*. Their exact size is architecture- and compiler-dependent, but they are guaranteed to have at least 32 bits [5]. Therefore, their maximum value could be as low as $2^{31} - 1 = 2,147,483,647$, making this the maximum input grid size our implementation is guaranteed to work on.

The number of wires is stored as an `int`, making $2^{15} - 1 = 32,767$ the maximum number of wires for which the behavior of our program is defined [5]. However, in order to fit into an `uint16_t`, the size after the reduction has to be smaller than 2^{16}, which depends on the placement of the wires. For an arbitrary wire placement, this is only guaranteed to be the case for at most $2^{14} - 2$ wires[15]. Note that while such a case can be solved by our implementation, the quadratic complexity (cf. Section 3) might make this infeasible, depending on the wire placement and how hard it is to find the cheapest path.

Another limit to consider is memory space. While running the A*-algorithm, up to twelve bytes[16] of memory might be needed per node for the worst case[17]. With $2^{14} - 2$ wires, this could translate to up to 27.0 GiB[18] of memory. However, with dense wire placement and an easy-to-find path, a case with $2^{14} - 2$ wires could require as little as approximately 48.5 KiB[19] to solve.

Stack space is not a concern for our implementation, as our sole recursive function[20] is tail recursive and exhibits only a logarithmic recursion depth.

[15] because the width and height after the reduction can be up to $4 \cdot w + 5$ (cf. Section 2.4)
[16] Note that we assume a byte to be eight bits for the following explanation.
[17] Two bytes per node for the graph representation, six bytes for the key–value pair in the priority queue, and four bytes in the `g_scores` table.
[18] $(3 \cdot (2^{14} - 2) + 5)^2 \cdot 12$ bytes $= 28,989,849,612$ bytes
[19] with 76×109 nodes: $76 \cdot 109 \cdot (2 \text{ bytes} + 4 \text{ bytes}) = 49704$ bytes
[20] `_pq_heapify_node`, used for the priority queue

5 CONCLUSION

In this paper, we tackled the challenge of a potentially enormous input grid graph, developing a technique to reduce the graph through the removal of identical neighboring columns or rows, since they do not change the cost of the cheapest path. For the original ICPC problem, we found that our reduction technique guarantees a graph size of at most 301×303. For the generalized version of the problem, our analysis revealed that the space and time complexity of our approach is quasi-quadratic in the number of wires w, while the input grid size s contributes only a logarithmic factor, extending the realm of what is feasible far beyond what is imaginable with a naïve approach.

Future research could explore finding more techniques to reduce the grid even further, as well as adapting pathfinding algorithms specialized for grid graphs, such as jump point search, to the specific characteristics of this problem.

REFERENCES

[1] Marco Couto, Rui Pereira, Francisco Ribeiro, Rui Rua, and João Saraiva. 2017. Towards a Green Ranking for Programming Languages. In *Proceedings of the 21st Brazilian Symposium on Programming Languages* (Fortaleza, CE, Brazil) *(SBLP '17)*. Association for Computing Machinery, New York, NY, USA, Article 7, 8 pages. https://doi.org/10.1145/3125374.3125382

[2] Cppreference 2024. https://en.cppreference.com/w/c/algorithm/qsort Accessed: 2024-08-14.

[3] Mathieu Fourment and Michael R. Gillings. 2008. A comparison of common programming languages used in bioinformatics. *BMC Bioinformatics* 9, 1 (05 Feb 2008), 82. https://doi.org/10.1186/1471-2105-9-82

[4] ICPC 2024. https://icpc.global/ Accessed: 2024-09-10.

[5] ISO/IEC. 2007. *N1256.* Standard ISO/IEC 9899:1999, TC3. International Organization for Standardization and International Electrotechnical Commission.
`int_fast32_t`: §7.18.1.3, p. 256
`INT_MAX`: §5.2.4.2.1, p. 22
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf
Accessed: 2024-08-14.

[6] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2021. Ranking programming languages by energy efficiency. *Science of Computer Programming* 205 (2021), 102609. https://doi.org/10.1016/j.scico.2021.102609

[7] GNU Project. 2024. GNU C Library (glibc-2.40). Source code available at https://ftp.gnu.org/gnu/glibc.

[8] https://db.cs.uni-tuebingen.de/staticfiles/ACM-problems/Wiring-Assistant.pdf [n. d.]. Accessed: 2024-08-17.