Exploring A* Search For Single and Multi Layer Automatic Routing

Team #1
Mohamed Shawky SEC:2, BN:16*, Remonda Talaat SEC:1, BN:20†,
Mahmoud Othman Adas SEC:2, BN:21‡ and Evram Youssef SEC:1, BN:9§

Department of Computer Engineering, Cairo University
Email: *mohamed.sabae99@eng-st.cu.edu.eg, †Remonda.Bastawres99@eng-st.cu.edu.eg,
‡mahmoud.ibrahim97@eng-st.cu.edu.eg, §evram.narouz00@eng-st.cu.edu.eg

***Abstract*—Recent advances in supercomputers and massive data centres increase the demand for fast processing on various types of data. Consequently, high performance chips become one of the critical factors in any computational system. Modern chips can contain million or even billions of transistors to achieve high performance and demanded functionalities. This makes the process of their design, implementation and integration complicated and tedious. One of the key challenges in modern chip manufacture is routing. Routing is the process of wiring different source transistors to their fan-outs. This process can be complicated in huge chips and requires the usage of specialized software, which is known as automatic routing. Several research projects have worked on this problem and various techniques are proposed. These techniques are mainly a compromise between execution time and solution optimality. In this work, we propose the usage of A* search algorithm, with a simple modified cost function, to solve the routing problem in a grid search formulation. The proposed solution can achieve the same or even better results than the considered baselines, while maintaining a reasonable execution time.**

***Index Terms*—vlsi, routing, grid search, a* search, maze, steiner tree, mikami-tabuchi**

I. Introduction

Fast computational systems require a large number of transistors, which makes their manufacturing process very hard. Many Chips have millions or even billions of transistors, which affects the circuit timing, power consumption, chip reliability and manufacturability that complicate all the design rules. One of the most important challenge is routing to connect the transistors, without causing any problem on the chip. Routing problem, in *VLSI*, is considered as an *NP-hard* problem, so it is divided into two design phases, *global routing*, where the grid is constructed with its nodes and edges, and *detailed phase* (our target), to find shortest paths to connect the required pins in the grid together. Many algorithms can be used to solve routing problem, as it can be formulated as grid search problem, where speedup and optimality are a trade-off. We experiment using *Lee's*, *Mikami-Tabuchi*, *Steiner tree* and *A* search* algorithms, where our main approach was A* search. We compare the different algorithms based on their execution time and length of metal, produced on the chip.

II. Terminology

- Maze : A $[D, W, H]$ matrix that contains cells, used to simulate the grid.
  - D : Number of layers.
  - W : Width of the layer.
  - H : Height of the layer.
- Cells : Each cell is a pin, a cell could represent a pin or an obstacle.
- Pin : The part where transistors gets connected to.
- Source : The starting point *(pin)* that needs to be connected to some targets.
- Targets : One or more point/s *(pin/s)* that need to be connected to the source in minimum cost.
- Obstacle : A block that wires can't go through.
- Vias : A passage to the upper or lower layer.
- Wire : What connects the pins with each others *(made of a specific metal)*.
- Path : The route, which the wire takes, in order to connect the source with all of the targets.
- Cost : The length of the path. The longer the wire, the larger the delay.
- Multi-layers : Instead of having only one $2D$ grid, we have multiple grids *(stacked vertically)*.
- Steiner points : Intermediate points that targets can be connected to.

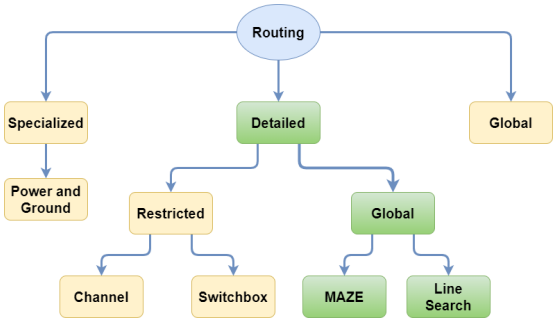


Fig. (1) Anatomy of various routing techniques

III. Related Work

A. Basic History

In 1959, *Edward F. Moore* presented one of the first shortest path through a maze algorithm [1], after a couple of years in 1961 *Lee, C. Y.* presented the idea of simulating the board wiring on electronics board as a maze [2]. Afterwards, the idea of *Lee Maze* has been revisited many times. In 1983, *Hightower, D.* made more contribution to the idea, such that using modern computers and virtual memories, we can mimic the routing problem precisely providing different techniques [3].

B. Router Problem Anatomy

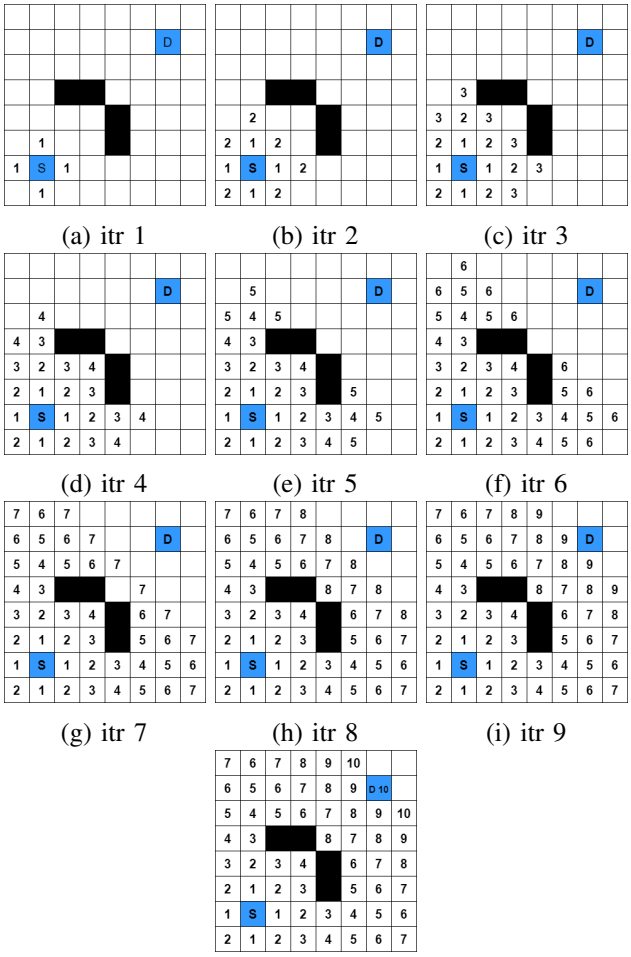
In this paper, we are mainly concerned with *detailed routing*. However, routing is divided into many subsections [4], as shown in fig.1. We only consider maze and line search *(global routing)*.

In the next subsection, we show how various algorithms work, and discuss three algorithms. First, the Lee algorithm which is a maze-based algorithm. Second, the Mikami-Tabuchi algorithm, which is a line-probe algorithm. Finally, the Steiner algorithm, which is a baseline algorithm.

C. Simulation Environment

In brief, we describe the simulation properties:

- Detailed routing *(not global)*.
- System is presented, as $3D$ Grid,$[D, W, H]$.
  - D: number of layers
  - W: width of each grid
  - H: height of each grid
- One source exists as a starting point.
- Multiple targets exist *(fan-out)*.
- Consistent Cost, which means no *bending* cost and no *extra* cost due to Vias.
- Vias cost is 1.
- Within the same $2D$ grid, path can move *horizontally* or/and *vertically*.



(a) itr 1 (b) itr 2 (c) itr 3

(d) itr 4 (e) itr 5 (f) itr 6

(g) itr 7 (h) itr 8 (i) itr 9

(j) itr 10

Fig. (2) Expansion Stage

D. Explored Techniques

1) Lee algorithm: This is one of the most common routing algorithms [2]. If there's a path between source S and some target T, the algorithm *definitely* finds it, and in case of consistent cost (i.e. no variable cost), the algorithm guarantees the shortest path. It uses BFS (breadth-first search) to connect targets with the source. It works appropriately with multiple layers (i.e. where Vias exist). The algorithm has three main stages:

- Expansion.
- Backtracking.
- Clean up.

The *expansion* stage creates a *halo-like* shape around the source, which expands and the cost of each cell is incremented, unless there's an obstacle (block cell). This continues until it hits a target and terminates. If the expansion reaches its limit with no target hit, this means that the target is not reachable.

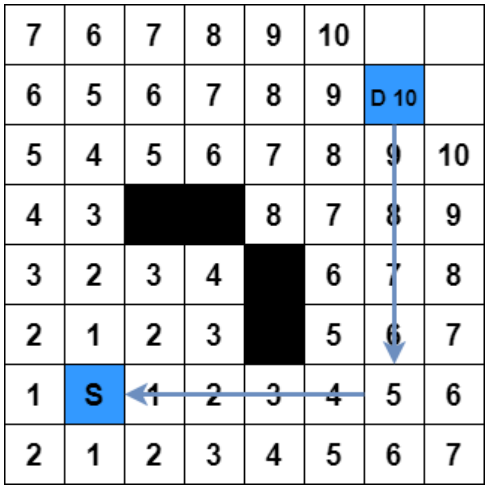


Fig. (3) Backtracking Stage

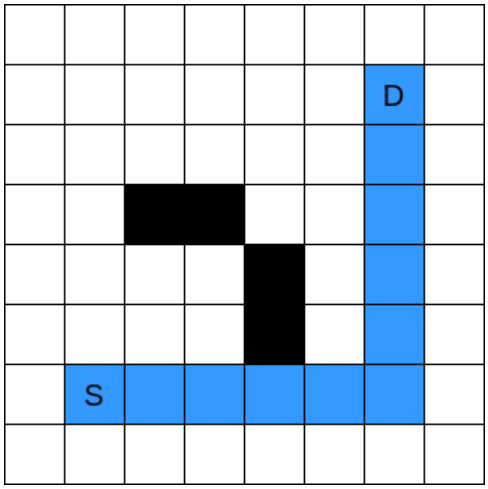


Fig. (4) Clean up Stage

The *backtracking* stage, shown in fig.3, gets the path from the target T to the source S. Since consistent cost is considered, then the backtracking stage is not that much of an issue. The cost is to be decremented by 1 from the target, until we reach the source and the cost of the path is its total length. *Data structure*, such as **priority queue**, is used to *pop-up* the cell with minimum cost.

The *clean up* stage, shown in fig.4, converts the path from the source to the target into obstacles *(blocks)*, so that no interference between paths exists.

2) Mikami-Tabuchi algorithm: Mikami-Tabuchi developed the algorithm named after them [5] to address the issues of *Lee maze* solver, which are the huge requirements of time and space.

Mikami-Tabuchi is simple, fast, low on memory resources, but it doesn't guarantee finding the optimal path. It only guarantee finding a path *(if exists)*.

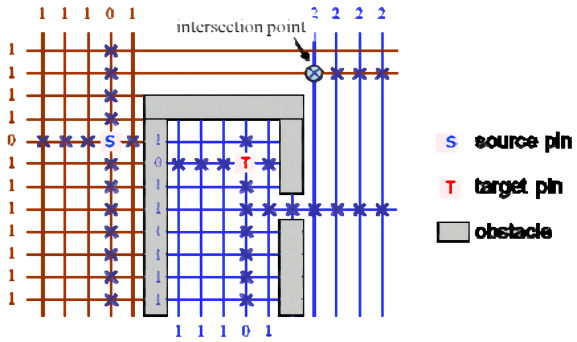


Fig. (5) Mikami Algorithm Illustration [6]

Mikami-Tabuchi algorithm is sometimes referred to as *line-probing algorithm* or *line-search algorithm*.

Mikami-Tabuchi basic idea is to extend horizontal and vertical lines/probes from source and target. The algorithm extends those lines, until they either hit an edge of the area or an obstacle.

If one of the lines of the source intersected with one of the lines of the target, the algorithm backtracks from the point of intersection to both source and target, and that's the solution.

Mikami-Tabuchi algorithm wasn't designed with multi-layer in mind. But it can be extended to work with multi-layers by using 3d probes/lines and representing the VIAs as lines that are perpendicular to the layer. If a line, in the layer, intersects with the VIA line, the algorithm, recursively, extends a probe on the same point, but on the other layer.

3) Steiner algorithm: The definition of *Steiner tree* is general, we are mainly concerned with minimum Steiner tree problem, and it's algorithm. Minimum Steiner Tree (MST) algorithm provides the minimum spanning tree that may exist in the graph, so that it connects multiple points with each others, using intermediate points called *Steiner points*.

It's basic idea is that, after including the first path, shown in fig.6, the shortest/closest target to the source any other target that will be connected to the source may be connected to any point of the extracted path, so that it could be connected to the main source or any Steiner point that's being produced *(Grey cells)*, and to connect that target, we need to know which of the left targets has the minimum distance to the source and all of the Steiner points, then pick up the one with the minimum cost, as shown in fig.7, and that's what makes time grows exponentially.

Algorithm 1: Mikami-Tabuchi Algorithm for Automatic Routing [7]

Result: Some path between source and destination.
Let S and T be a pair source and destination respectively;
Generate 4 lines (2 horizontal and 2 vertical) passing through S and T;
Extend these lines till they hit obstacles or the boundary of the layout;
if *a line generated from S intersects a line generated from T* **then**
    backtrack to find path;
    return path;
end
$i \leftarrow 1$;
while *no new lines were created* **do**
    foreach *line* $L \in$ *level* $i-1$ **do**
        foreach *Point* $p \in$ *line* L **do**
            Generate a perpendicular line L2;
            Extend line L2 till it hit obstacles or the boundary of the layout or intersects with other line L3, where $L3 \in levelj$ and $j > 0$;
            if *L2 intersects with L3 and L2 and L3 are from different terminals* **then**
                backtrack to find path;
                return path;
            end
        end
    end
    $i \leftarrow i + 1$
end

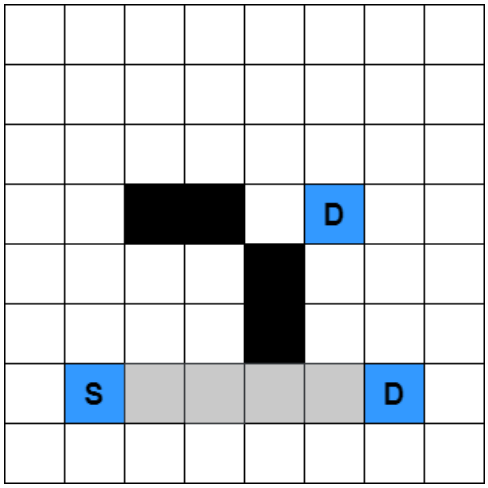


Fig. (6) Steiner tree: step 1

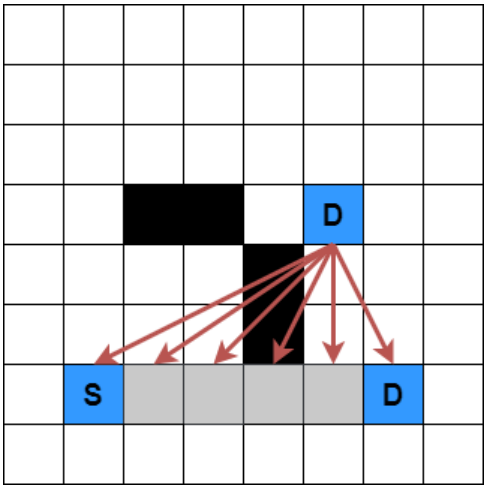


Fig. (7) Steiner tree: step 2

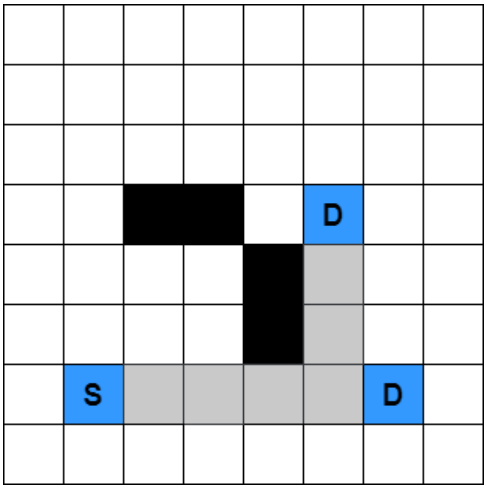


Fig. (8) Steiner tree: step 3

The algorithm works as follows [8]:

Algorithm 2: Steiner Tree algorithm For Maze Routing

Result: Optimal/Minimum path between source and all target cells.
Find the closest target to the source and Steiner points;
while T *Doesn't span all terminals* **do**
    Select terminal x not in T that is closest to a vertex in T;
    Add to T the shortest path that connects x with T;
end

As mentioned before, Steiner tree always provide the optimal path, when it exists, but the Steiner tree problem is

NP Hard problem, both exact and approximate solutions exist, and as for our condition we are interested in the exact solution. The solution provided to the problem is very simple, the minimum path to be found every time is provided using *Dijkstra* algorithm, due to all this computations, as the dimensions of the graph get larger and the number of targets increase the time complexity increases exponentially. The intermediate points that are now sources and targets can be connected to are called Steiner points.

IV. Methodology

A. Motivation

Metal routing is a critical step in systems integration process, where each source is connected to its fan-outs using non-crossing metal. Routing in a modern chip, with millions or even billions of transistors, can be very complicated and cumbersome, so the manufacture process has moved to automatic routing.

Automatic routing is a very vast field, with several techniques being developed through time. Automatic routing involves lots of problems like finding shortest non-blocked path and VIAs in multi-layer routing.

Automatic routing techniques are always a trade-off between optimality and speed. Some techniques target execution time by reaching a sub-optimal solution. Others target optimal solution with very high execution time.

In this work, we seek a good balance between performance and optimality of solution by introducing the usage of A* search for automatic routing with a modified cost function.

B. Formulation

Routing is formulated as a grid search problem 9, where an occupancy grid map *(OGM)* is provided with some source and destination cells to be joined. The *OGM* cells can have the value of 1 for occupied cell or 0 for empty cell. The *OGM* can be of multiple levels, in case of multi-layer routing, where the cell can have a third value to indicate that the cell can be used as a VIA. Only 4-neighbor cells are considered, so the path can move up, down, left or right. The path can also move from one level to another through VIA cells.

Given this problem formulation, several grid search and shortest path algorithms can be used to find the optimal path between each source and its given destinations. These algorithms can vary based on optimality and performance.

The two main metrics considered in this work are path length and execution time. Total length of metal, covering the grid, is used as a measure of solution optimality. It's defined as the number of grid cells, on which the metal is placed for routing.

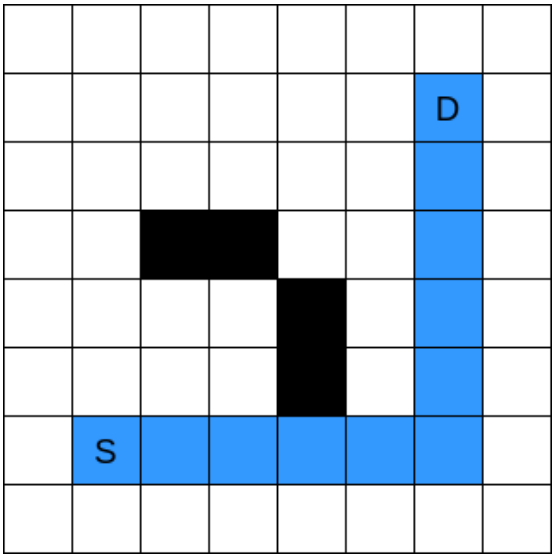


Fig. (9) Example of occupancy grid map (OGM), where S is the source, D is the destination, white cells aren't occupied, black cells are occupied and blue cells represent the path.

Execution time is used as a measure of algorithm performance. It's simply the total elapsed time by the algorithm to find all required paths.

To develop a comparison framework between the baselines and our proposed technique, we propose the following assumptions:

- All wires are of the same size.
- No geometric rules violation (i.e. spacing).
- All pins are placed at the center of cells.

C. Baseline

To measure the validity and performance of our proposed method, three main baselines are used. These baselines are well-established algorithms that are currently used in industry.

1) Maze Routing (Lee's Algorithm): The first baseline algorithm, to be considered, is *maze routing*, discussed in III-D1. Maze routing is one of the most well-established and widely-used algorithms in metal routing. It consists of a wide range of algorithms and techniques. *Lee's algorithm* is the basic maze routing solution and it's considered as our first baseline for comparison. This algorithm is basically a grid search algorithm for routing.

2) Mikami-Tabuchi's Algorithm: The second considered baseline algorithm is *Mikami-Tabuchi's* algorithm, discussed in III-D2. Unlike *Lee's algorithm*, this algorithm is

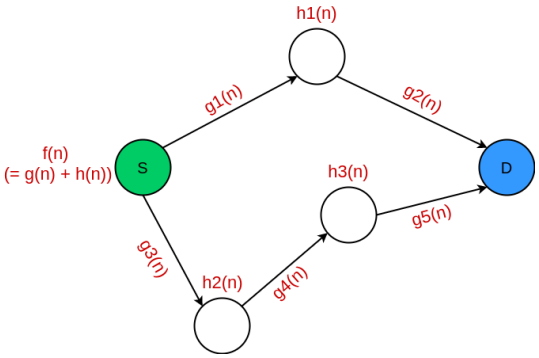


Fig. (10) An illustrative example for A* search process and cost function.

a line search technique that adopts line-search operations. It can find a path between source and destination cells, but it doesn't guarantee the shortest possible path.

3) Steiner Tree Algorithm: Steiner tree spans though the given subset of vertices in a given graph. Steiner tree, discussed in III-D3, is suitable for any situation, where the task is minimize cost of connection among some important locations, like VLSI Design, Computer Networks, etc. So, Steiner tree is considered as our final baseline algorithm.

D. Modified A* Search

The main contribution of this work is the usage of *A* search*, with modified cost function, as a new routing technique. The main idea of *A* search*, as shown in fig.10, is the usage of some heuristic function to assign a cost for each node. To define a path from a source node to a destination node, the nodes that minimizes the estimated path cost are chosen at each step. The main objective is to minimize the following function:

$$f(n) = g(n) + h(n) \tag{1}$$

Where *f(n)* is the total cost of the path from a specific node, *g(n)* is the cost of the edge between current node and next chosen node and *h(n)* is the estimated cost of the next chosen node based on the used heuristic function.

Thus, the required information to solve an *A* search* problem is the cost of edge between each two nodes and the heuristic function to get an estimated cost for each node.

The *OGM* representation of the routing problem can be formulated to become an *A* search* problem. The edge cost *g(n)* is always 1, as the path can only move from one cell to an adjacent cell. The heuristic function, proposed in this work, is *Euclidean distance* between next chosen cell and destination cell, as shown in fig.11. So, the cost of taking any of the adjacent cells is *Euclidean distance* + 1, which can be simplified to *Euclidean distance* between an adjacent cell and destination cell.

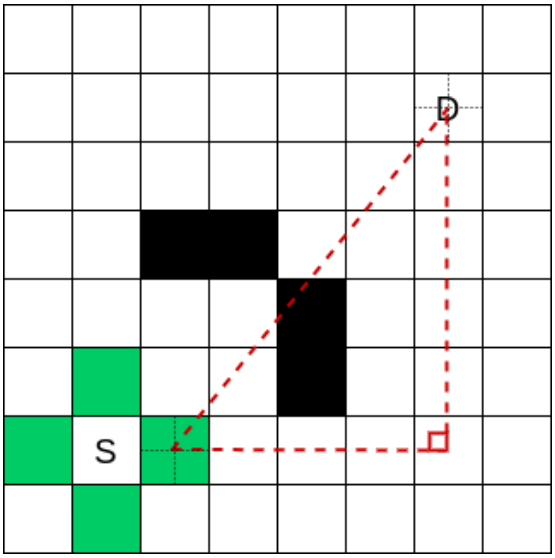


Fig. (11) An illustrative example for the new heuristic function based on Euclidean distance. The distance is measured between the centres (pins) of both cells.

The proposed algorithm can be summarized as follows:

Algorithm 3: Modified A* Search For Automatic Routing

```
Result: Optimal path between source and
        destination cells.
append source cell to the path;
while destination not reached do
    get adjacent cell that minimizes euclidean
     distance;
    append cell to the path;
    if current cell is blocked then
        remove current cell from path;
        backtrack to the previous cell;
        continue;
    end
end
```

V. Experimental Results

A. Goals

We want to answer those questions about our proposed algorithm:

1) How fast is it?

2) How short are the found paths?

Because *fast* and *short* are relative terms. We need to calculate them in a comparison with other approaches. We choose to compare our results against:

- Lee Maze: First algorithm used in the industry. Helps us know how far we have progressed relative to the industry practical techniques.
- Mikami: known by its speed, but very far from the optimal answer. Would give us insights on how much speed we have achieved.
- Steiner Tree: Finds the optimal answer. We will use it to know whether our results are optimal.

B. Implementation

We implement our proposed algorithm in `mod_a_star.py`, lee maze algorithm in `maze_lee.py`, mikami-tabuchi algorithm in `mikami_tabuchi.py` and Steiner tree in `steiner_tree.py`.

All scripts read JSON input from `stdin` and write json output to `stdout`, so we can chain them with the other scripts. They follow the `io_schema.md` specs about io format.

C. IO Specs

Input contains a $d \times h \times w$ grid matrix, where:

- $d \rightarrow$ Number of layers, either 1 or 2.
- $h \rightarrow$ Height.
- $w \rightarrow$ Width.

Each cell is either:

- $0 \rightarrow$ Empty.
- $1 \rightarrow$ Obstacle.
- $2 \rightarrow$ Via, only when $d = 2$ and must exist on the other layer, too.

Each input contains the source coordinates and a list of targets coordinates.

Output should contain the found paths and their corresponding lengths.

Both input and output should be in JSON format. See `io_schema.md` for more details.

D. Helper Scripts

We write a couple of scripts to assist with the comparison and testing:

- `gen-input.py`: Generates random input that follows `io_schema.md`. It doesn't guarantee that all targets are reachable. For more info `$ python3 gen-input.py --help`
- `verify.py`: Takes the input to the algorithm and its output and verifies the correctness of the result. See `$ python3 verify.py --help`
- `random_test`: Generates infinite random inputs, runs given algorithm on each test, and verifies the results.
- `random_comp`: Generates N random input, runs each algorithm on each input, calls `calc_total.py` to calculate total cost and verifies the results.
- `nConst`: Calls `random_comp` M times, each time with same number of targets, varying the grid area (w,h).
- `areaConst`: Calls `random_comp` M times, each time with same width and height of the input grid, varying the number of targets.
- `merge_comp`: Merges the outputs of `random_comp` into one `tmp/summary.json` with the summary of the experiment.
- `plot.py`: Plots given `summary.json` through the stdin to `tmp/plot`.

E. The Experiment

For simplicity, we assume all grids are squares. So $w = h$ in all tests. We also assume number of layers $d = 2$ in all tests.

We need to very the area, while the number of targets is constant. In the other case, we vary the number of targets, while the area is constant. In both cases, we record and plot the running times and costs.

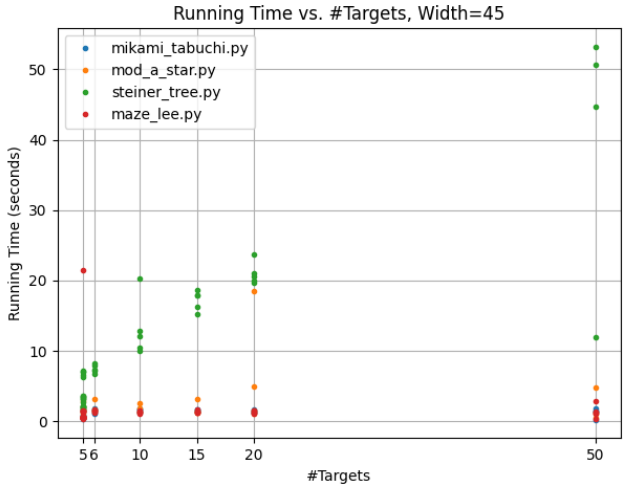
We expect some algorithm to take a very big time to calculate the output. Unfortunately, we can't just let it run forever. So, we just sat the timeout as 5 minutes. This is why we collected the number of found targets, so we compare how many times an implementation has time-outed and resulted in 0 final targets found.

Note: Not all targets in some input have to be reachable. Bigger grids have bigger probability of having non-reachable targets.

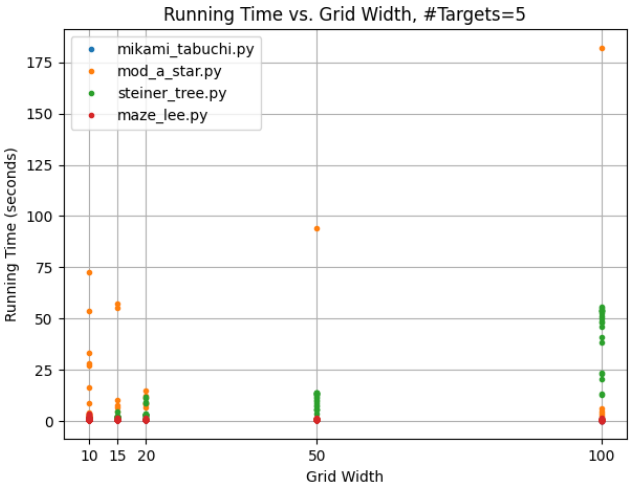
We start the experiments by running:

1) `nConst`, which, for each area of areas of grid in [10, 15, 20, 50, 100], conducts 5 random experiments on each algorithm given the same random input (for each experiment), in which the number of targets (n) is constant and $n = 5$.
2) `areaConst`, which, for each number of targets in [6, 10, 15, 20, 50], conducts 20 random experiments on each algorithm given the same random input (for each experiment), in which the area of the grid (w, h) is constant and $w = h = 45$.

After running the 2 scripts multiple times and then merging their outputs using `merge_comp`, we have so far 396 unique experiment, each experiment is a unique

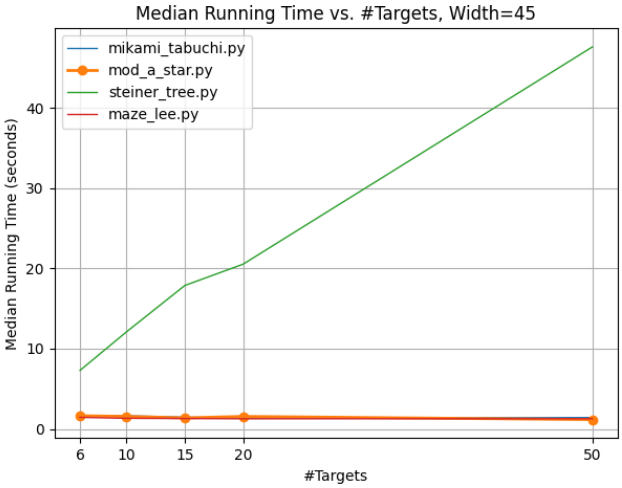


(a) Time / #Targets

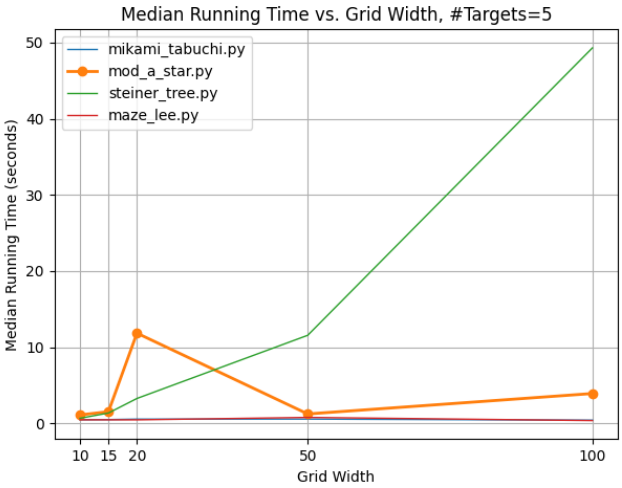


(b) Time / Grid Width

Fig. (12) Running Time



(a) Median Time / #Targets



(b) Median Time / Grid Width

Fig. (13) Median Running Time

input given to the 4 algorithms and all the 1584 results are in `summary.json`.

F. Comparisons

We make the plots using `plot.py`. The following is the line of thoughts, we have, through the comparison.

1) Running Time Scatter: We start by scattering all the running time per #Targets and per Grid Width, see Fig. (12).

The results are not very clear as the points of Mikami and lee are compressed down, because of Steiner and A*.

We can also notice a* has some outliers in time, specially in bigger grid widths (50, 100). This is why we can use the median function multiple times, instead of the average.

2) Median Running Time: To make the plots clearer, we calculate the median of the running time, instead of scattering all points. See Fig. (13).

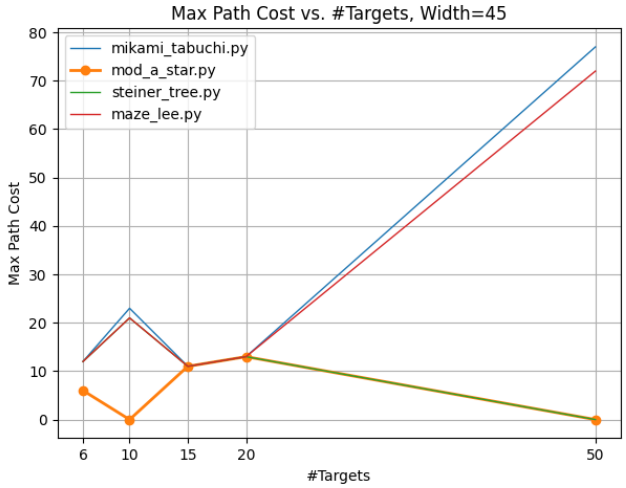
We notice that both Mikami and Lee are very fast, they take less than 2 seconds, no matter what the input size is.

We also notice that Steiner grows exponentially with both the size of the grid and the number of targets. But it's affected more by the growth of number of targets.

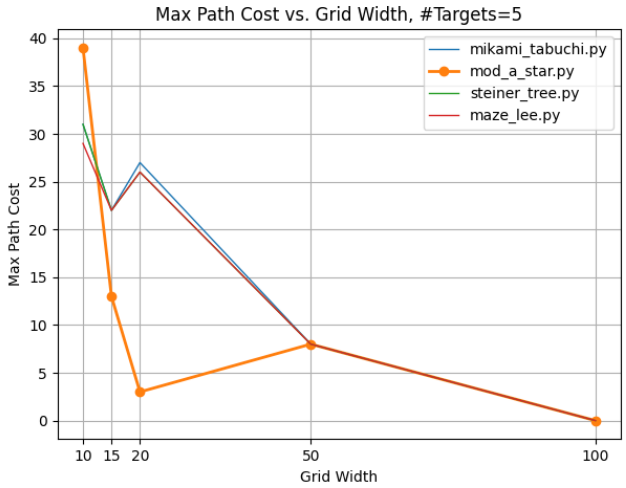
But our new approach grows linearly with only the size of the grid. Its performance is independent from the number of targets.

3) Cost: Next, we move on to comparing cost. Fig. (14, 15, 16) shows maximum, average and total cost, respectively, achieved by each algorithm per area and per number of targets.

We notice that A* has achieved overall lower costs in

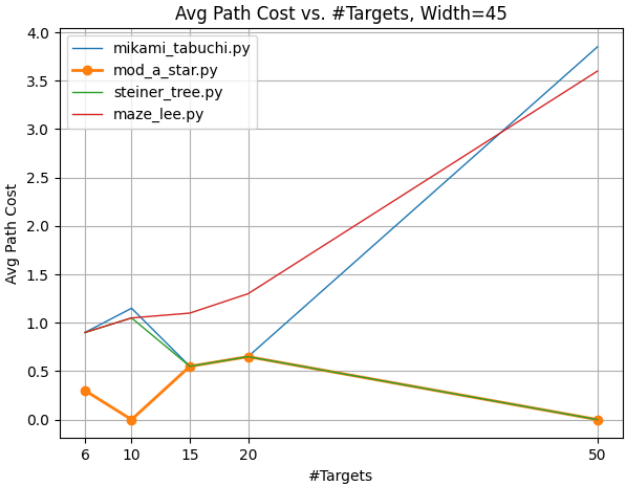


(a) Max Path Cost / #Targets

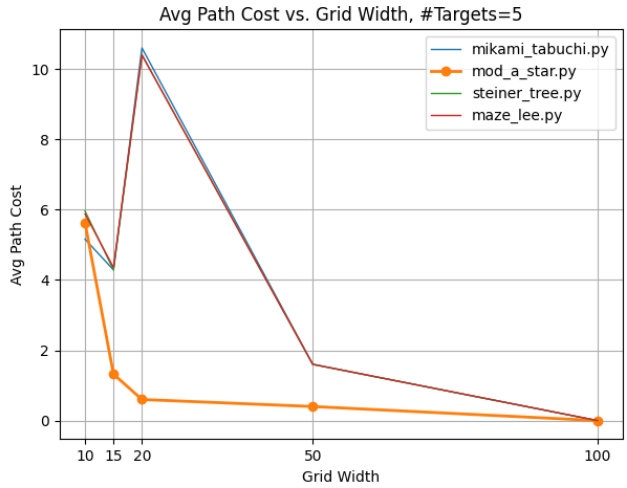


(b) Max Path Cost / Grid Width

Fig. (14) Maximum Path Cost



(a) Avg Path Cost / #Targets



(b) Avg Path Cost / Grid Width

Fig. (15) Average Path Cost

the paths it found, regardless of number of targets or grid area. For some reason, it doesn't find any on $n = 50$.

All algorithms find no results when grid area = 100, because `gen-input.py` generated non-reachable points on that size.

Mikami-Tabuchi is the worst overall in the costs.

4) Errors: We notice that some algorithms go to 0 on cost. So, we calculate the percentage of found targets by each algorithms, to compare how many targets each algorithms found per n. Fig. (17) shows that percentage.

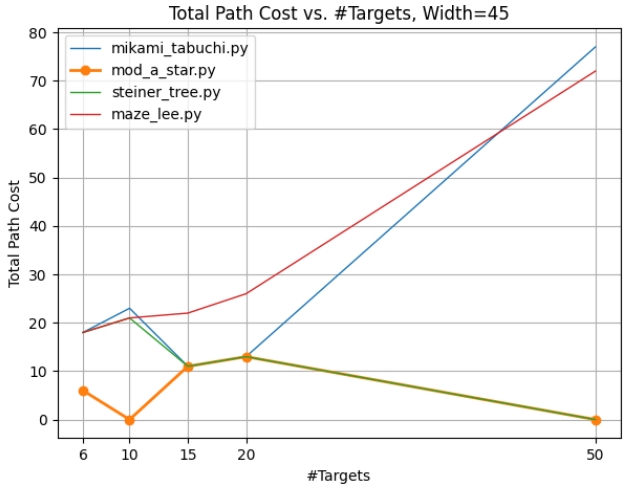
The percentage of targets alone don't explain the reason why both A* and Steiner had some zero costs. Is it because of timeouts? Or because some issues with our code?

Fig. (18) is a plot of timeouts per area and per n. We notice that both Steiner and A* have high time-outing chances, compared to Mikami and lee. Unfortunately, we can't increase the timeout threshold more than 5 minutes, otherwise the nearly 2000 tests won't finish executing.

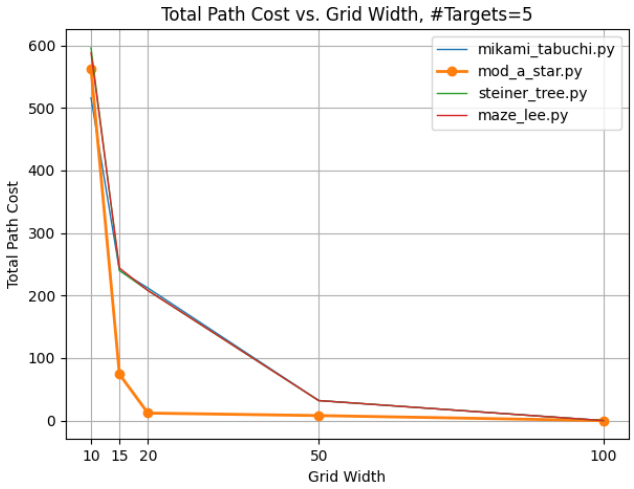
So far, given this data, we can conclude that the high than average error rate of A* and Steiner are due to their high chances of time-outing.

VI. Conclusion

In this work, we discuss the *detailed routing* problem, and how it's an important challenge in manufacturing process, showing that it can be formulated as grid search problem. *A* Search* algorithm, with euclidean distance as cost function, is our main approach in the problem solution. We have tried different grid search techniques and compared their results with our main approach with respect to performance and time, and found that *A* Search*



(a) Total Path Cost / #Targets



(b) Total Path Cost / Grid Width

Fig. (16) Total Path Cost

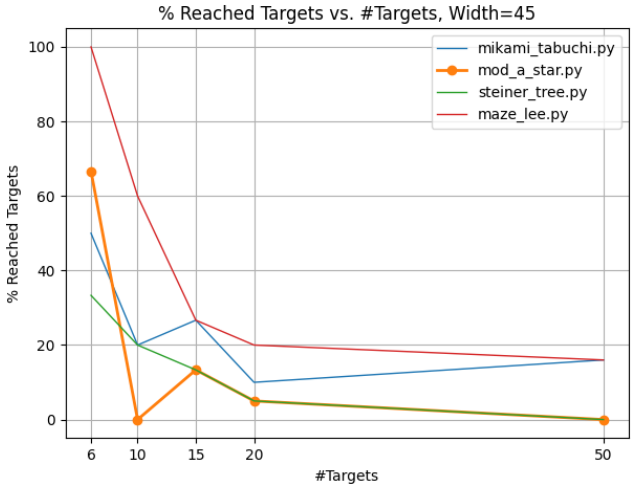
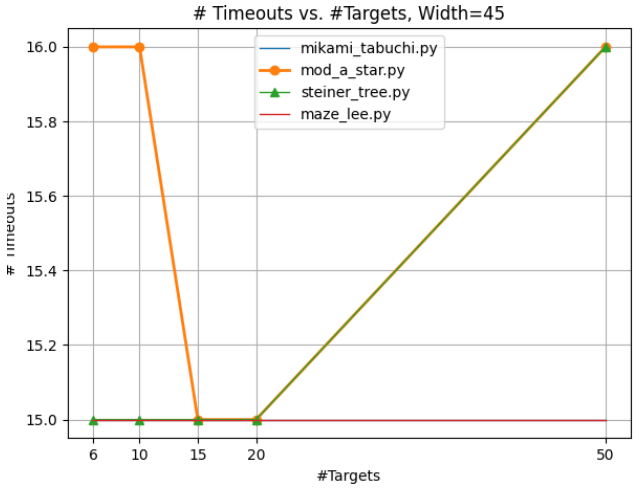
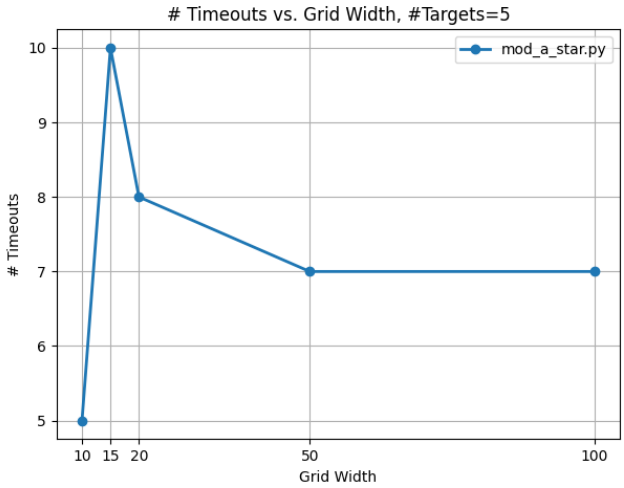


Fig. (17) Percentage of Found Targets / #Targets



(a) #Timeouts / #Targets



(b) #Timeouts / Grid Width

Fig. (18) #Timeouts

achieves better performance in some cases and same performance in others, maintaining an acceptable time, compared to optimal algorithms, but relatively longer time than approximate algorithms.

REFERENCES

[1] Edward F. Moore. The shortest path through a maze, 1959.

[2] C. Y. Lee. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers*, (EC-10 (2): 346–365, doi:10.1109/TEC.1961.), 1961.

[3] D. Hightower. The lee router revisited. *IEEE international Conference on computer design: 136-139*, 1983.

[4] R. Venkateswaran and P. Mazumder. Routing algorithms for vlsi design. 1990-1991.

[5] Koichi Mikami. A computer program for optimal routing of printed circuit connectors. *IFIPS Proc., 1968*, 1968.

[6] Huang-Yu Chen and Yao-Wen Chang. Global and detailed routing. In *Electronic Design Automation*, pages 687–749. Elsevier, 2009.

[7] Indranil Sengupta. Grid routing. 2018.
[8] M. Zachariasen. Algorithms for plane steiner tree problems. *Phd. Thesis, Department of Computer Science, University of Copenhagen*, 1998.