Exploring A* Search For Single and Multi Layer Routing

Mohamed Shawky
Computer Engineering
Cairo University
Email: mohamed.sabae99@eng-st.cu.edu.eg

Mahmoud Adas
Computer Engineering
Cairo University
Email: mahmoud.ibrahim97@eng-st.cu.edu.eg

Remonda Talaat
Computer Engineering
Cairo University
Email: Remonda.Bastawres99@eng-st.cu.edu.eg

Evram Youssef
Computer Engineering
Cairo University
Email: evram.narouz00@eng-st.cu.edu.eg

***Abstract*—**
***Index Terms*—**

I. Introduction

II. Terminology

A. Basic Terms

Maze: it's a $[WxH]$ matrix that contains cells, used to simulate the grid.

Cells: each cell is a pin, a cell could represent a pin or an obstalce.

Pin: the part where transistors gets connected to.

Source: the starting point (pin) that needs to be connected to some targets.

Tatgets: one or many point/s (pin/s) that needs to be connected to the source in minimum cost.

Obstacle: a block that wires can't go through.

Vias: like a ladder to the upper or lower layer.

Wire: what connects the pins with each others.

Path: the route which the wire will take in order to connect the source with all of the targets.

Cost: the length of the path, the longer the wire the longer the delay.

Multi-layers: instead of having only one $2D$ grid, we have multiple grids, stacked vertically.

Steiner Point: intermediate points that targets can be connected to.

B. Assumptions

All wires are the same size.

No geometric rules (ie. spacing) violations.

All pins are the center of cells.

III. Related Work

A. Basic History

In 1959, Moore, Edward F. 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]. Starting from there 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 memic the routing problem precisely providing different techniques [3].

B. Router Problem Anatomy

The Routing problem has many sections and subsections, in this paper we are mainly concerned with Detailed routing. Detailed routing is divided into many subsections, as you can see in fig.1 and we are exploring Maze and Line Search subsections.

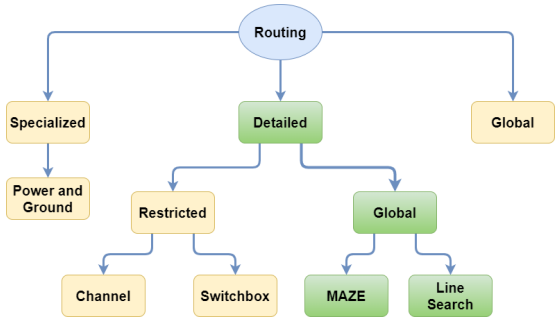


Fig. (1) Anatomy of various routing techniques.

In the next section we are showing how various algorithms work, and discussing three algorithms, first the

Lee algorithm it's a Maze based algorithm, second the Mikami-Tabuchi algorithm it's a Line-Probe algorithm, and finally the Steiner algorithm it's a baseline algorithm.

C. Simulation environment

Rapidly we'll describe our Simulation's 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
- Only one source exist as a start.
- Multiple targets exist (nested), fan-out.
- Consistent Cost: no bending cost, no extra cost due to vias.
- Vias cost is 1.
- Within the same 2D grid path could be horizontally or/and vertically chosen.

D. Explored Techniques

Before diving into these algorithms make sure to read the Assumptions section first.

1) Lee algorithm: This is one of the most common and origin routing algorithms [2].

If there's a path between source S and some target T, the algorithm will definitely find it, and in case of consistent cost (ie. no variable cost) the algorithm will not only find the a path but also the shortest one.

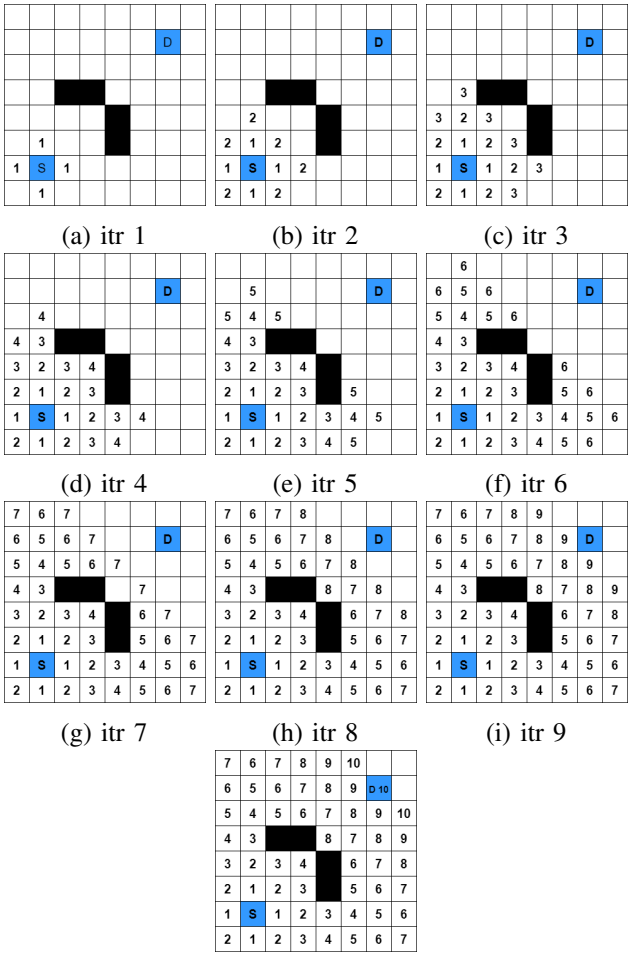
It uses BFS (breadth-first search) to connect targets with the source.

It works appropriately with multiple layers (ie, where vias exist).

The algorithm has three main stages:

- Expansion
- Back-tracking
- Clean up

The *Expansion* stage fig.**??** creates like a *halo* shape around the source, and it gets larger and the cost of each cell is incremented, unless there's an obstacle (block cell), until it hits a target and terminates, if the expansion reached it's limit with no target hit, this means that the target/s is/are not reachable.



(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

The *Back-tracking* stage fig.3 gets the path from the target T to the source S. Since we are dealing with consistent cost, then the backtracking stage is not that much of an issue all we have to do is to decrement the cost by 1 from the target, until we reach the source. In other versions where inconsistent cost exist, and the cost of the path is the total length of it. **Data structure** such as **priority queue** is used to *pop-up* the cell with minimum cost.

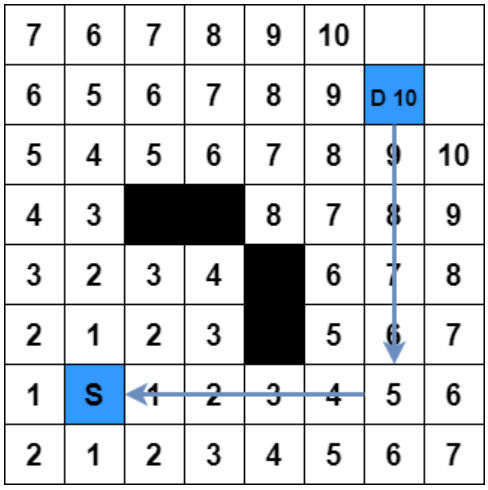


Fig. (3) Back-tracking Stage

The *Clean up* stage fig.4 converts the path from the source to the target into obstacles (blocks) so that no interference between pathes may exist. and then starts to connect another target to the source with that path of obstacles added.

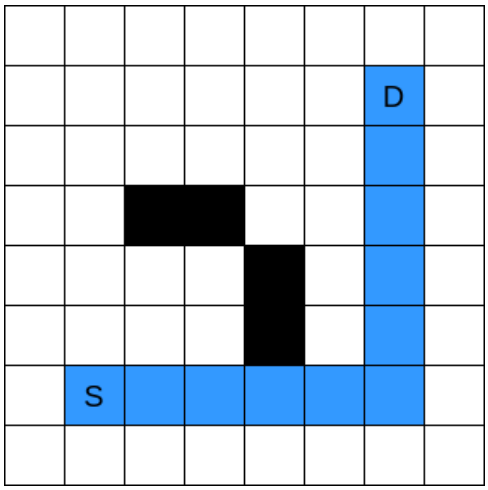


Fig. (4) Clean up Stage

2) Mikami-Tabuchi algorithm: Mikami-Tabuchi developed the algorithm named after them [4] to address the issues of Lee maze solver, which is the huge requirements of time and space.

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

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.

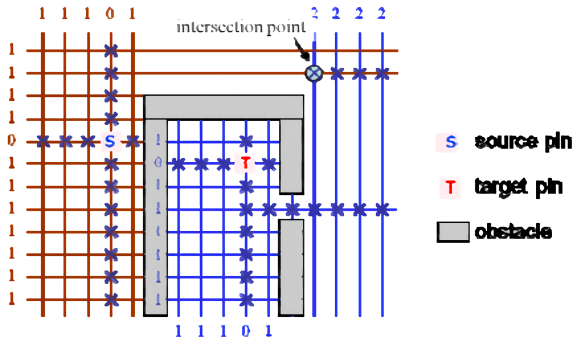


Fig. (5) Mikami Algorithm Illustration [5]

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 ot both source and target, and that's the solution.

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

3) Steiner algorithm: is the baseline algorithm for our work.

The definition Steiner Tree is very general, we are mainly concerned with menimum Stiener tree problem, and it's algorithm.

Minimum Steiner Tree (MST) algorithm provides the minimum spanning tree that may exist in the graph.

It's basic idea is that, after including the first path -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 earlier extracted path.

The algorithm works as follows [7]:

As mentioned before, Steiner 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 ot 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.

Algorithm 1: Mikami-Tabuchi Algorithm for Automatic Routing [6]

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

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;
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

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

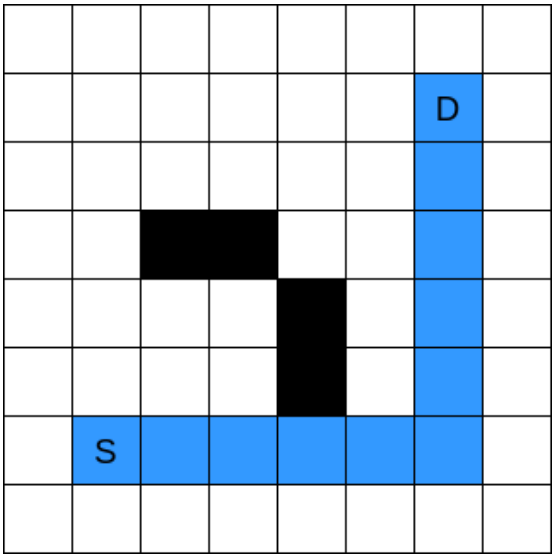


Fig. (6) Example of occupancy grid map (OGM), where S is the source, D is the destination, white cells aren't occupied, black cells are occupied and blue cells represent the path.

Routing is formulated as a grid search problem 6, 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.

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.

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*. 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. Unlike *Lee's algorithm*, this algorithm is 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 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

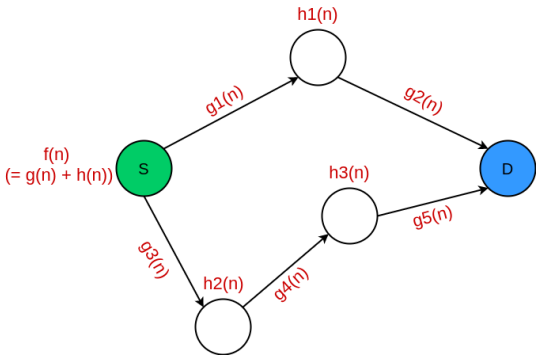


Fig. (7) An illustrative example for A* search process and cost function.

The main contribution of this work is the usage of *A* search* as a new routing technique. The main idea of *A* search*, as shown in fig.7, 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.

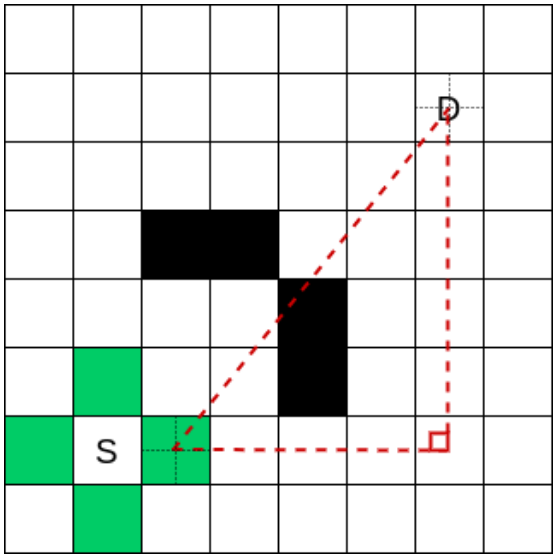


Fig. (8) An illustrative example for the new heuristic function based on Euclidean distance.

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.8. 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.

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 chose to compare our results aginst:

- 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.
- Lee Maze: First algorithm used in the industry. Helps us know how far we have progressed relative to the industry practical techniques.

B. Code

We implmented our proposd 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 the `stdin` and writes json output to `stdout`, so we can chain them with the other scripts. And 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.
- $h \rightarrow$ Hieght.
- $w \rightarrow$ 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 lengthes.

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

D. Helper Scripts

We wrote 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 gurantee 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, run given algorithm on each test, and verify the results.
- `random_comp`: Generates N random input, run each algorithm on each input, calls `calc_total.py` to calculate total cost and verify 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 assumed all grids are squares. So $w = h$ in all tests. We also assumed number of layers $d = 2$ in all tests.

We need to very the area while the number of targets is constant. And in the other case, vary the number of targets while the area is constant. And in both, we will 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 implmentation has timeouted 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 started 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 algo given the same random input (for each experiment) in which the number of targets (n) is const and $n = 5$.
2) `areaConst`, which for each number of targets in [6, 10, 15, 20, 50], conducts 20 random experiments on each algo 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 had so far 396 unique experiment, each experiment is a unique input given to the 4 algorithms and all the 1584 results are in `summary.json`.

F. Comparisons

We made the plots using `plot.py`. The following is the line of thoughts we had through the comparison.

1) Running Time Scatter: We started by scaterring all the running time per #Targets and per Grid Width, see Fig. (9).

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 may use the median function multiple times instead of the average.

2) Median Running Time: To make the plots clearer, we calculated the median of the runing time instead of scattering all points. See Fig. (10).

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.

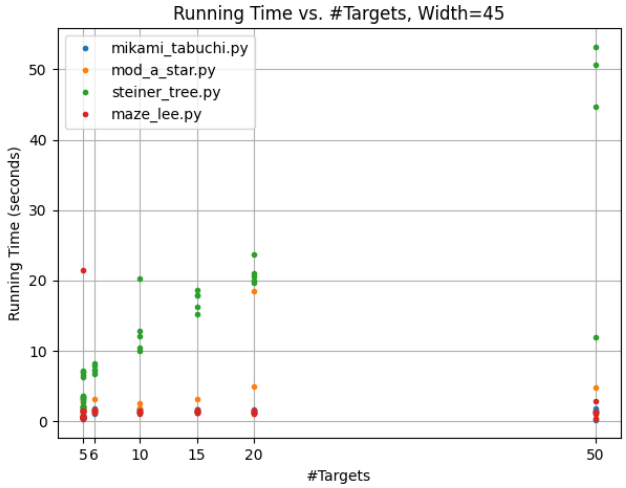
3) Cost: Next we move on to comparing cost. Fig. (11, 12, 13) shows maximum, average and total cost respectively achieved by each algorithm per area and per number of targets.

We notice that a* has achieved overall lower costs in the paths it found, regardless of number of targets or grid area. For some reason it didn't find any on $n = 50$.

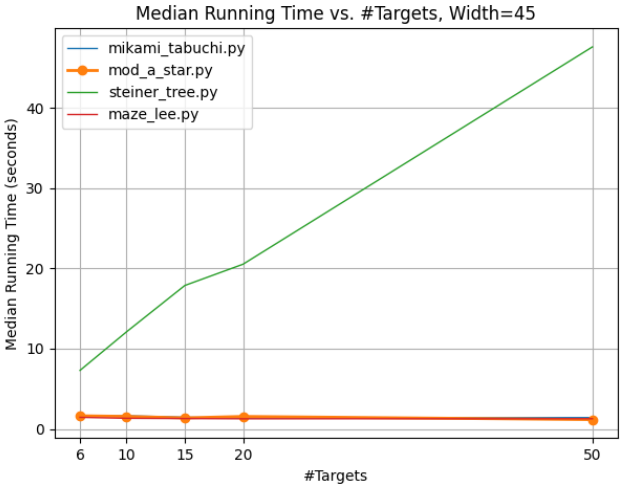
All algorithms found no results when grid area = 100. Because `gen-input.py` generated non-reachable points on that size.

Mikami-Tabuchi was the worst overall in the costs.

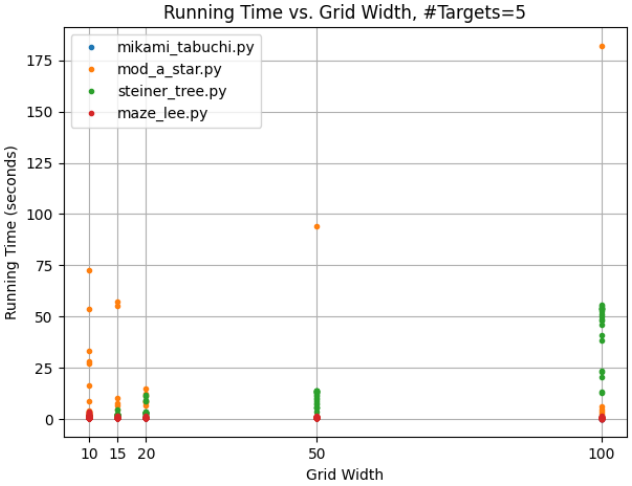
4) Errors: We noticed that some algorithms went to 0 on cost. So we calculated the precentage of found targets



(a) Time / #Targets

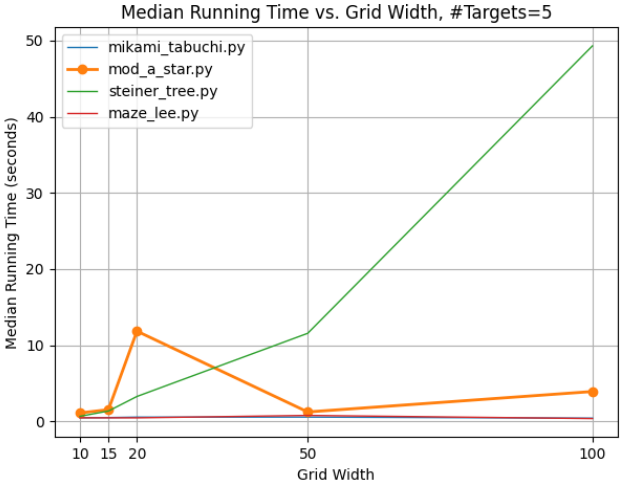


(a) Median Time / #Targets



(b) Time / Grid Width

Fig. (9) Running Time



(b) Median Time / Grid Width

Fig. (10) Median Running Time

by each algorithms, to compare how many targets each algorithms found per n. Fig. (14) shows that precentage.

The precentage 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. (15) is a plot of timeouts per area and per n. We notice that both steiner and a* have high timeouting 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 timeouting.

VI. Conclusion

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.

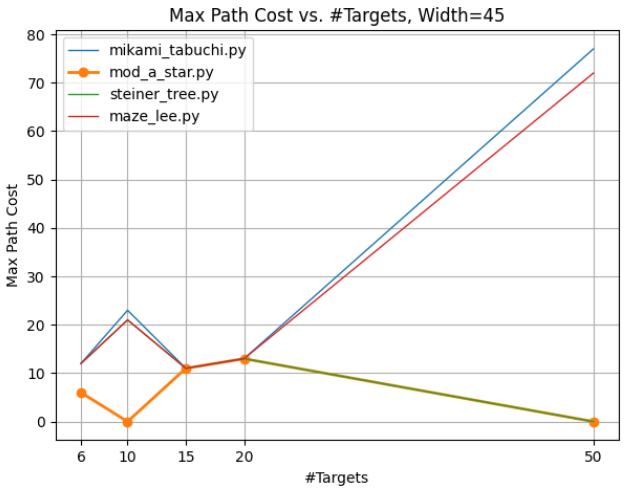
[3] D. Hightower. The lee router revisited. *IEEE international Conference on computer design: 136-139*, 1983.

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

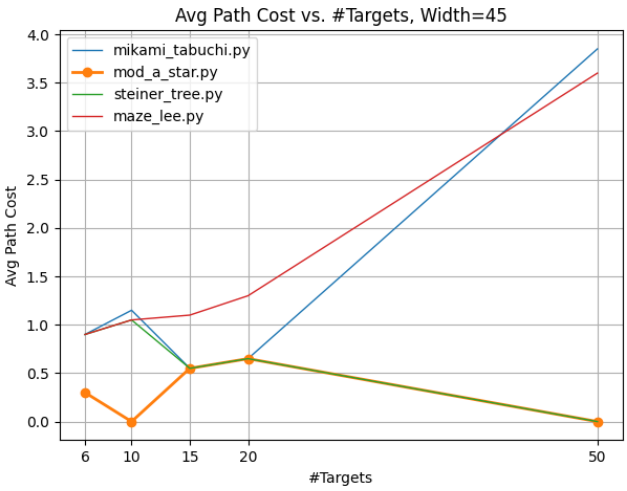
[5] Huang-Yu Chen and Yao-Wen Chang. Global and detailed routing. In *Electronic Design Automation*, pages 687–749. Elsevier, 2009.

[6] Indranil Sengupta. Grid routing. pages ?–?, 2018.

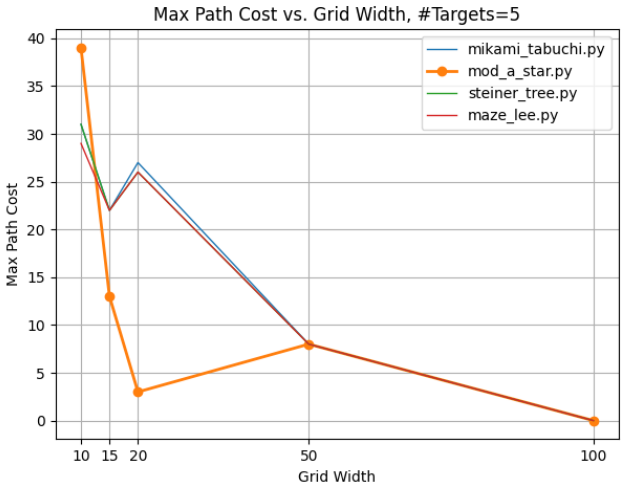
[7] M. Zachariasen. Algorithms for plane steiner tree problems. *Phd. Thesis, Department of Computer Science, University of Copenhagen*, 1998.



(a) Max Path Cost / #Targets

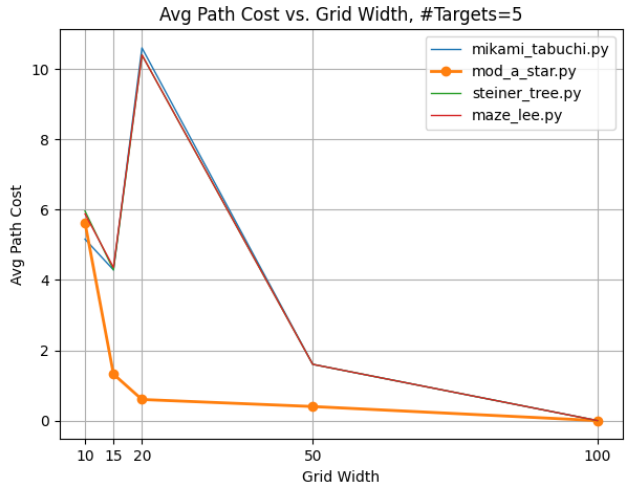


(a) Avg Path Cost / #Targets



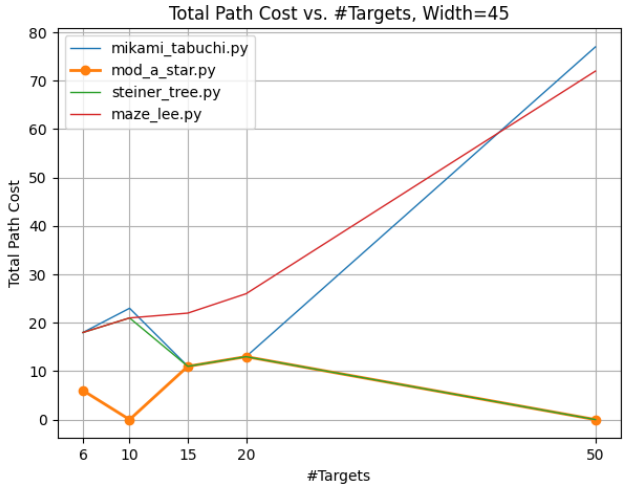
(b) Max Path Cost / Grid Width

Fig. (11) Maximum Path Cost

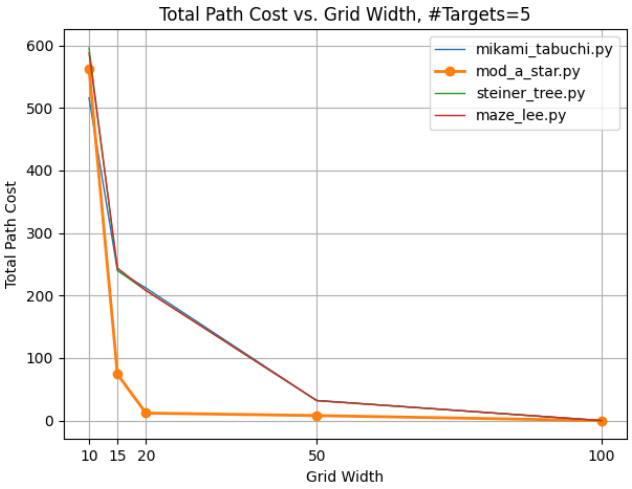


(b) Avg Path Cost / Grid Width

Fig. (12) Average Path Cost



(a) Total Path Cost / #Targets



(b) Total Path Cost / Grid Width

Fig. (13) Total Path Cost

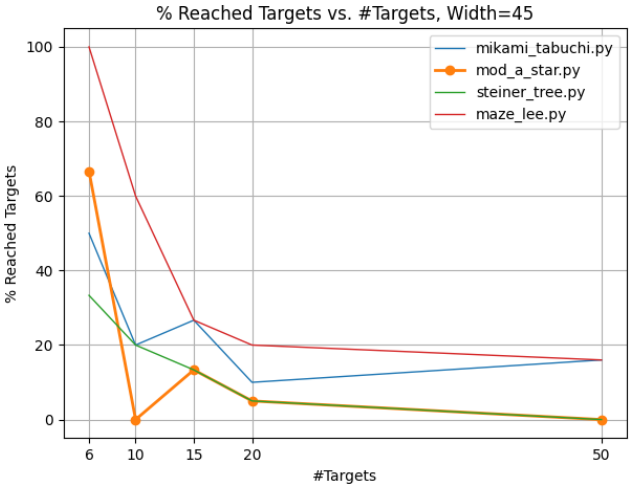
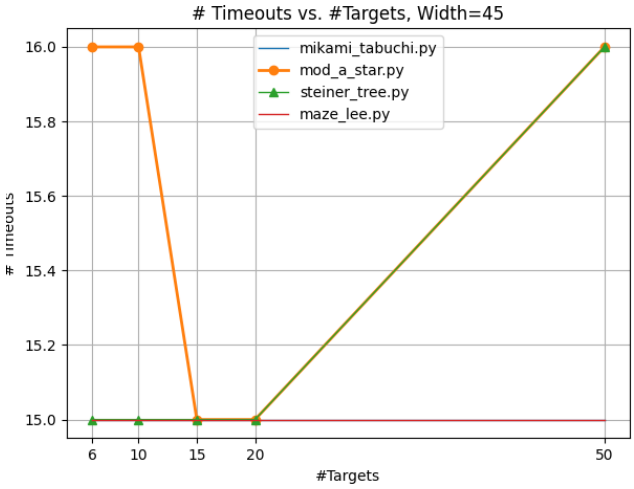
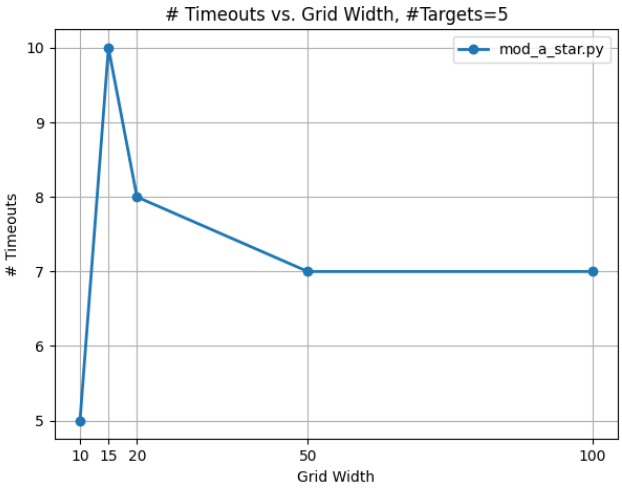


Fig. (14) Percentage of Found Targets / #Targets



(a) #Timeouts / #Targets



(b) #Timeouts / Grid Width

Fig. (15) #Timeouts