CPEN513 Assignment 1: Routing

Martin Chua
35713411
January 30, 2023

Summary

This report details the implementation of the Lee-Moore algorithm, A* algorithm, and A* improvements. These algorithms are run on benchmark circuits given on Canvas. The algorithms were implemented in Python. The graphics library was implemented using matplotlib. A copy of source code and documentation can be found in Appendix A. This report is divided into the following sections: Lee-Moore algorithm implementation, A* algorithm implementation, initiatives on top of A* to improve its performance, how the code was tested, and graphics.

Table I: Comparison of the implemented Lee-Moore vs A* algorithm.

Benchmark Circuit (total # of segments)	**Lee-Moore Routable Segments**	**A* Routable Segments**	**Total Lee-Moore Cells Visited**	**Total A* Cells Visited**	**% difference over Lee-Moore**
rusty (4)	4	4	231	71	30.74%
sydney (3)	3	3	177	116	65.54%
stanley (5)	5	5	235	155	65.96%
impossible (5)	1	1	338	64	18.93%
misty (5)	3	3	238	143	60.08%
impossible2 (4)	3	3	862	503	58.35%
oswald (2)	1	1	133	133	100.00%
kuma (6)	5	4	859	261	30.38%
wavy (1)	1	1	843	715	84.82%
temp (17)	4	4	10114	4690	46.37%
stdcell (17)	14	15	8714	2970	34.08%

Table I above compares the performance between the Lee-Moore and A* algorithm and their number of routable segments. Cells in green show circuits that were fully routable. Cells in red show circuits that were not fully routable. Routable segments refer to a single pin-to-pin connection. Multiple nets with each net having multiple segments may exist. We observe that the worst case A* performs the same as the Lee-Moore algorithm with one exception: A*'s *kuma* route is one segment less, possibly due to A* prioritizing older locations with the lowest distance cost over newly-found locations with the same cost. The greatest percentage difference between the two is the *impossible* circuit. A* visits fewer cells than Lee-Moore and generally finishes faster. Due to *oswald's* circuit layout, the total cells visited between Lee-Moore and A* are the same. Besides the circuits that are impossible, some improvements can be made to improve performance. This will be discussed in the initiatives section and compared with A*.

Implementation of the Lee-Moore Algorithm

The Lee-Moore Algorithm implemented is a basic, 4-direction expansion according to the pseudocode below:

```
Algorithm 1 leemoore(grid, source, sinks)
    1. From the source, visit in all directions until the first sink is reached.
        a. Backtrack and construct a working net from the sink to the source.
    2. For all other sinks (if any), visit in all directions (ie. left, up, down, right) until any part of the previous constructed net is reached.
        a. Backtrack and construct a net from that point back to the sink.
    3. If all values are visited within the boundaries surrounding the current pin, set the current pin as 'unable to route' and repeat (2)
```

Multiple sinks (step 2) can be handled in the same way as a single sink (step 1) as we expand in all directions. The main difference is that subsequent routes follow a greedy approach, taking the first valid intersection point with the working routed net as the completed route. We define 'working' in step 1a as a not-yet-complete value. Henceforth we use 'working' to delineate between complete and in-progress routed nets or pins. Screenshots of the stanley and stdcell final routing are shown in Fig. 1.

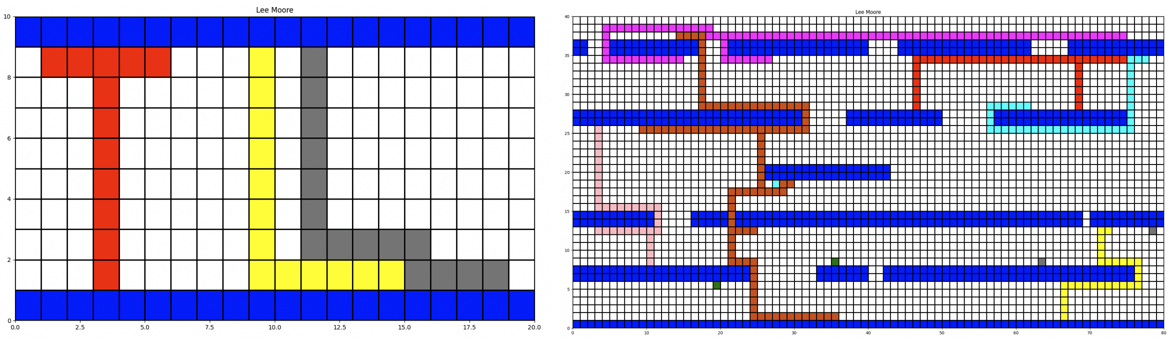


Fig. 1: Lee-Moore routes on the stanley.infile (left) and stdcell.infile (right) benchmark circuits.

Implementation of the A* Algorithm

The standard A* algorithm is implemented without any reordering of nets. The ordering of nets in the benchmark circuit affects whether a circuit can be fully routed, and is later discussed in the Initiatives section. One major difference between A* and Lee-Moore is the addition of a "cost" value, which consists of the working distance taken from the source (current travelled distance) added to the working distance to the destination (Manhattan distance). Cost is minimized during traversal, and the lowest cost value is traversed first. Due to this cost value, multiple sinks require special handling. As the Manhattan cost depends on a destination, we set the current sink pin as the working source and minimize the Manhattan distance from the working source to any location on the working routed net. Therefore, rather than setting the destination as the previous routed pin, we set the destination as the location on the working routed net with the lowest Manhattan distance to our working source (current sink pin). Screenshots of the *stanley* and *stdcell* final routing are shown below. The A* algorithm is able to route one more segment in *stdcell* compared to Lee-Moore without any additional initiatives or optimizations. The final *stanley* A* route is the same as Lee-Moore, but with a 66% reduction in cells visited.

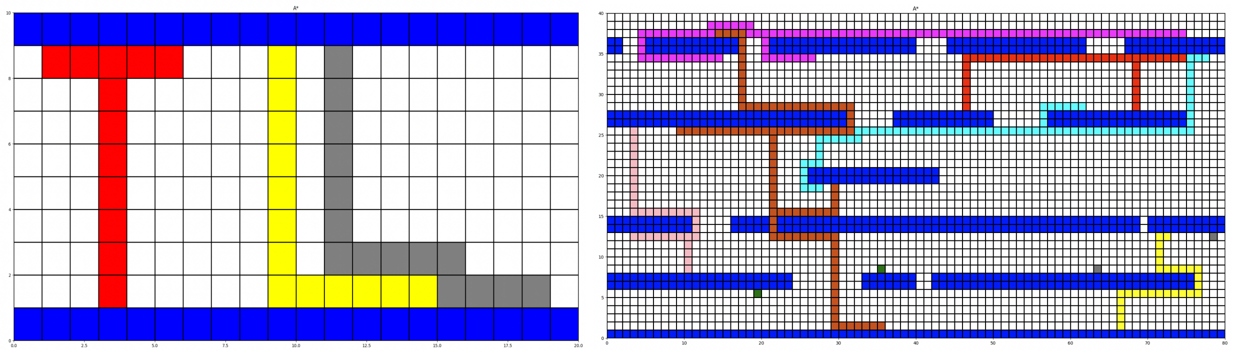


Fig. 2: A routes on the stanley.infile (left) and stdcell.infile (right) benchmark circuits.*

Initiatives

As A* has an overall higher performance, improvements will be made to the A* algorithm. Several ideas in order from least to highest complexity were brainstormed. Unfortunately, only the first two were implemented in the interest of time.

1. Approximately pre-sort wires by shortest path with a weight multiplier for the number of pins before any route. This is done by adding up all the manhattan costs for each pin to another, taking the average, and then adding the number of pins multiplied by the weight multiplier. Modifying the weight multiplier affects the rate of convergence of harder circuits. Due to the 3 page maximum, I've left it as 1x without sweeping or plotting it.
2. Rip-up and re-route. We retry and increase priority of failed routes. This is done by keeping a priority queue and increasing the priority of failed routes. The number of retries is defined in a variable so that it does not keep trying. Similar to the first initiative, I've left it as 50 routes.
3. Global (low granularity) and detailed (high granularity) route. This *could* be done by partitioning the overall benchmark circuit into smaller benchmark circuits in memory. These smaller benchmark circuits could be partitioned as the largest rectangle within its boundary. Smaller benchmark circuits that contain pins will have corresponding edges, with the edge location being the minimized Manhattan cost to its destination. The pins are routed to edges and each smaller benchmark circuit is appropriately connected to others.

The details of the first two implementations can be found in Appendix A - `implementations.py`. Table II gives a comparison between Lee-Moore, A* and Initiative 1+2.

Table II: Comparison of the implemented A* vs Initiatives 1+2 algorithm.

Benchmark Circuit (total # segments)	**Lee-Moore Routable Segments**	**A* Routable Segments**	**Initiative 1+2 Routable Segments**	**Total A* Cells Visited**	**Total Initiative 1+2 Cells Visited**
rusty (4)	4	4	4	71	70
sydney (3)	3	3	3	116	117
stanley (5)	5	5	5	155	159
impossible (5)	1	1	3	64	197
misty (5)	3	3	5	143	190
impossible2 (4)	3	3	3	503	530
oswald (2)	1	1	1	133	132
kuma (6)	5	4	5	261	878
wavy (1)	1	1	1	715	712
temp (17)	4	4	15	4690	2835
stdcell (17)	14	15	17	2970	2930

Cells in green and red show circuits that were fully routable and not fully routable respectively. Although Initiative is unable to fully route all benchmark circuits, we see a significant improvement in routable segments over the base A* algorithm in *temp*. The total cells visited is also significantly lower. Both *stdcell* (Fig 4, left) and *misty* are now able to be routed. We can observe that with the implemented initiatives, even at the worst case, it is equal to the Lee-Moore algorithm. These results make sense, as a re-route gives improved results with an increased number of fully routable circuits. Although *oswald* and *kuma* are unable to be fully routed, the third (non-implemented) initiative could potentially make these routable.

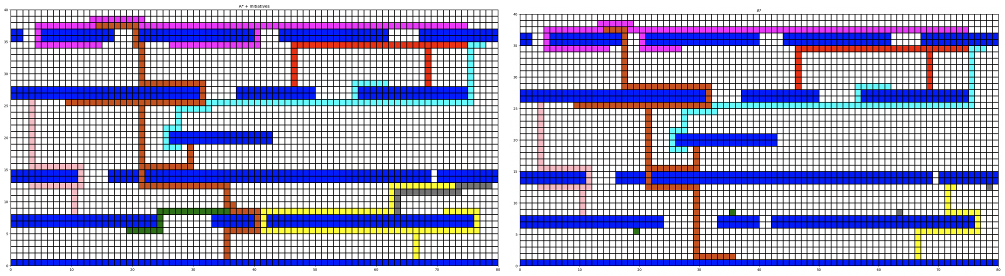


Fig. 4: Final routing of the stdcell.infile benchmark circuit for Initiative 1+2 (left) and A (right). The Initiative 1+2 algorithm is able to fully route.*

Verification

Whitebox testing (unit tests) was done with PyTest. The `test_asn.py` file can be found in Appendix A and is split into two sections: initialization tests and A* tests. For the sake of keeping the 3 page limit, screenshots are not included. Small checks are also within the algorithm code to ensure correct functionality. A* tests check for proper cost function calculations. Blackbox testing was done only via visual inspection of the animation as it would take significantly more time than writing the actual code. Examples of the animation are in the next section.

Graphics

The `asn1.py` script can be run via command line or an IDE like Visual Studio Code + Python extension. Video examples can be found here: Lee Moore and A*. GIFs can be found in the documentation located in Appendix A.

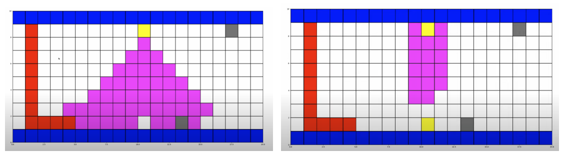


Fig. 5: Visualized intermediate routing of the Lee-Moore and A routes on the sydney.infile benchmark circuit.*

The script is designed to be fully configurable and can be run via terminal. Variables can be modified within `asn1.py` before running: `plot_final_only`, `plot_empty`, `plot_leemoore`, `plot_astar`, `plot_initiative`, and `pausetime`. The speed of the algorithm's progress can be sped up or down using `update_interval`. However, there seems to be a bottleneck limit to the animation speed depending on the circuit. For larger circuits such as stdcell, the fastest speed is not very fast, likely due to each update requiring the refreshing of all data points.

Appendix A

Source code and documentation can be found here: https://github.com/mchuahua/CPEN513/blob/master/asn1/

README.md

Pre-requisites

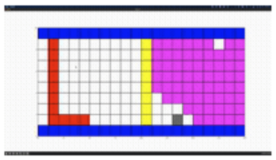
- Benchmark `*.infile` circuits within a subdirectory named `benchmarks`
- `pip install matplotlib`
- (optional) `pip install pytest`

Usage

- `py asn1.py` for visualizing A* and Lee-Moore
- `pytest` for running unit tests
- Open `asn1.py` to modify variables for configuration

Examples

Lee-Moore



A*