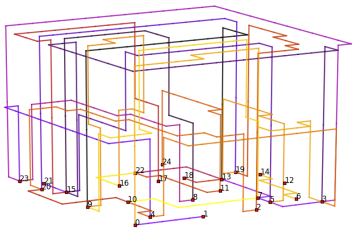
Chips & Circuits



Aynel Gül
10595945
Gijs van Horn
10070370
Harm Manders
10677186

Programmeertheorie

Supervisors
Dhr. drs. Daan van den Berg
Dhr. Wouter Vrielink

Faculty of Science
University of Amsterdam
Science Park 904
1098 XH Amsterdam
December 22, 2016

1 Introduction

An integrated circuit (IC), or a "chip", is essentially a small electronic device made out of semiconductor material. IC's are used for a variety of devices, from computers and microwaves to moon-landing rockets and satellite navigators. They are usually designed logically and consist of gates, which are connected by nets. Because of the logical design, the end gates for each net are known beforehand.

The objective here is to design the physical layout of net-paths in a Manhattan grid with a minimum number of layers, given a configuration of gates and list of nets. This is a constraint optimization problem, because the paths are not allowed to intersect and a path needs to be planned for each net. The total length of paths needs to be minimized. So, altogether, the question here is how can we find a solution for configuring paths between logical gates and optimize these to a minimum length?

In this case we examine two gate layouts, with three netlists per gate layout of increasing length. To estimate the complexity of the problem, it is often useful to look at the state space. However, the state space of this problem is so immense, it is hard to determine all possible configurations within the problem. The responsibility for this enormous state space lies mainly in the fact that, in theory, you could add an infinite amount of layers. Also, adding one net to the grid is already possible in a myriad of ways. Imagine adding another one, up to an amount of 70 nets, resulting in an explosion of the state space.

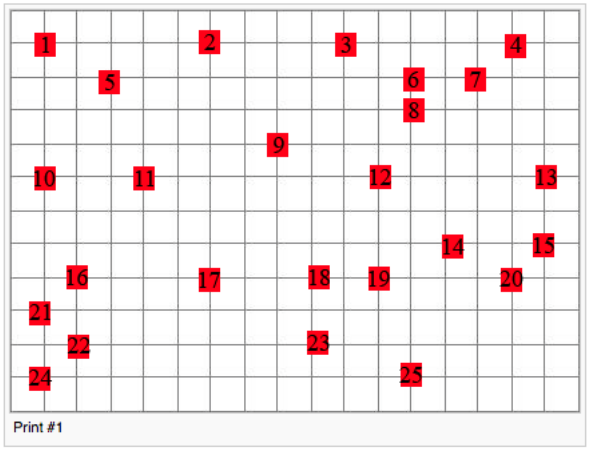


Figure 1: *Print #1, arrangements of gates on a base without nets*

2 Data structures

One of the big decisions that have to been made when implementing something is the data representation. The board is a three dimensional array initialized with zeros, effectively setting a maximum depth from the start. Each cell of the board contains either a 0, or a reference to a Gate- or a Net object. For effective retrieval of nets and gates, the board object also contains dictionaries of these gates and net objects, which respectively contain a coordinate or a list of coordinates.

3 Algorithms

During the design process of the router several algorithms have been used. First, an implementation of Dijkstra's algorithm was implemented to find a path for an individual net. Due to long running times, we replaced this later by the more efficient A* path planner. Finally, optimization attempts of the paths were made using a genetic algorithm.

3.1 Score function

Scoring of a net configuration is simply the sum of the length of all nets, and minimizing this score is the ultimate goal. However, this score does not take the number of nets and absolute distance between endpoints into account, making scores between different board and gate configurations much harder. This is the reason for the introduction of a relative score.

$$S_r = \sum_{n=1}^{|N|} \frac{L_n}{D_n}$$

Where L_n is the path length of net n, and D_n is the Manhattan distance between end-gates of n.

3.2 Net solving order

The order in which the paths are planned is important, because the nets placed earlier can form obstacles for nets that have to be placed later. We expected that shuffling the order in which the nets are planned to have a significant impact on the total cost.

After reordering sequences, based on the minimum length between two gates calculated with the Manhattan distance, and approximate cardinal direction of the path (horizontal, vertical, two diagonals) yielded no solutions, a recursive function did. This recursive function attempts to solve the netlist in order, until it encounters an unplannable net. It then places this net at the start of the list and starts over, until a solution is found. As result of this function, we were able to actually find solutions for all netlists on both prints.

3.3 Dijkstra & A*

Dijkstra's algorithm is an algorithm used for finding the shortest path between nodes in a graph, which in our case, represent the paths between the gates. For a given source node in the graph, the algorithm finds the shortest path between that node and every other. First, you put the value of the initial node to zero. After determining the neighbors of the initial node, you determine its values by adding 1 to the initial node. When all neighbors of the current node are explored, the current node is marked as visited and removed from the unvisited set to avoid exploring the same node twice or more. This process is repeated until you hit the target node. In our case, it is important to avoid nodes that are already occupied by another net or gate to avoid overlap. Therefore, when determining the neighbors, nodes that are already occupied are not included in the list of neighbors that have to be explored. After all available nodes are given an appropriate value, you can count back from the target node to the source node. To find the shortest path, all you have to do is to jump to the neighbor with the lowest value until you reach the initial source node with a value of 0.

The A* algorithm is an extension of Dijkstra's algorithm which attempts to reduce the number of nodes that need to be explored. It does so by introducing a heuristic, which in our case is an estimated distance between a node and the target node. This estimation needs to be optimistic for the algorithm to work, so we take the Manhattan distance, which is always smaller or equal to the shortest path that exists. The number of nodes to be discovered is reduced by only expanding the nodes with the lowest compound cost (the sum of the length of the path between the start node and that node, and the estimated cost of getting from that node to the end node). This still guarantees finding the shortest path [2], but prevents calculating costs for nodes in the opposite direction.

The problem with both Dijkstra and A* is that the paths they return occupy a lot of space on the lower levels of the board, which is due to their nature of finding the shortest path. Because of all the gates being at the lowest level, this quickly leads to encapsulation of gates that we need to connect later on when these paths tend to stay as low as possible to find the shortest paths.

In an attempt to prevent this from happening, the implementation of A* introduces a bias in distance, instead of neighbouring nodes being 1 apart, the distance from node a to b is increased depending on the number of gate and net

objects adjacent to b and the height of node b (according to algorithm 1).

```
Data: Adjacent nodes a and b
Result: distance between a and b
distance = 1 + (board_depth^2) / (b_height + 1);
for neighbours of b do
    if neighbour is gate then
        distance += 4;
    end
    if neighbour is net then
        distance += 3;
    end
end
return distance;
```

Algorithm 1: *Calculating distance between node a and node b*

3.4 Genetic algorithm

After finding a solution for every netlist on both boards with a modified A* algorithm, we shifted our attention to the optimization of these solutions. The order in which the nets are placed seemed to be a key element in finding solutions. Therefore, we attempt to make the optimization order-independent by implementing a simplified genetic algorithm, akin to parallel hill climbing.

There are three important steps the algorithm adheres to: initialization, selection and reproduction. The algorithm starts with initialization, by copying the solution found by the recursive A* procedure until the population is a certain given size. A genetic algorithm should generally initialize with a varied population, but since the first generation of mutations introduces a great deal of variation, it is implicitly included in the initialization. During each generation, each individual is mutated by selecting a random net, finding the first 100 shortest paths with (unbiased) A* for that net, and selecting one of these randomly. This prevents the individual from getting a higher score (or in terms of genetic algorithms: decrease in fitness), again, making this implementation more like a parallel hill climber. Selection is then performed by deleting the fifty percent of individuals with the lowest score and reproduction by copying the best fifty percent.

During each generation we want to make it likely that each net is altered, and it should also be the case that every net is optimized in all solutions found. Making sure that this happens with random net selection is a common statistics problem known as the coupon collectors problem; how many draws (with replacement) does it take to draw each ball from an urn? According to paul Erdős [1], the expected number of draws needed is of the order $n\,ln(n) + n$. In this case we use this formula to calculate the population size and maximum number of generations, where n is the length of the netlist.

4 Results

Our decision to switch to A* for individual net path planning is justified by the results in figure 2 where we have compared the running time of both Dijkstra's algorithm and A*. It shows that the running time for A* increases at a lower rate than Dijkstra's with increasing distance between end nodes, and that it has less variance.

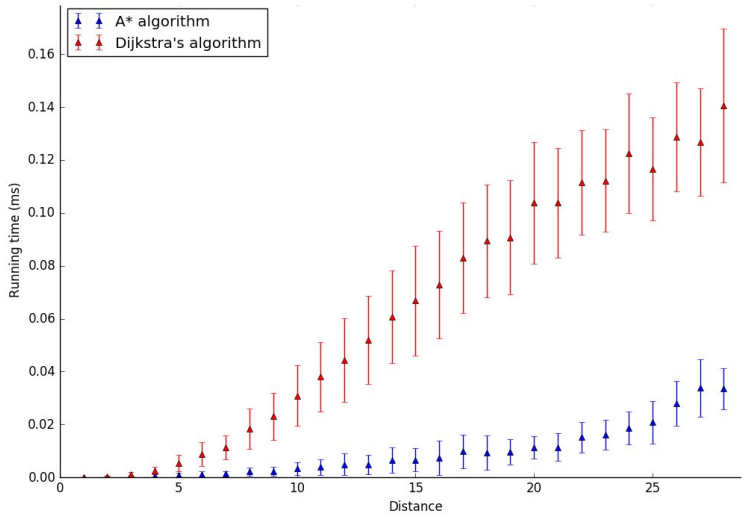


Figure 2: *Median and variance in running time as a function of distance between gates*

The scores that the complete procedure, both finding and optimizing a solution, achieved can be seen in table 1. The columns labeled *before* contain the results of the constraint satisfaction phase; A* with altered distance measures. The columns labeled *after* contain the results of optimization by the genetic algorithm. Notable here is that the relative scores increase, and that improvements decrease, as the netlist length increases.

Figure 3, along with the other figures in appendix A, shows an example of the physical layout the method produces. Note that the bias in A* causes very long and high paths in the left configuration, leaving a great deal of improvement for the genetic algorithm. The result for the first print and netlist is a configuration that is only 3 layers deep.

print	**netlist**	**before** absolute score	**before** relative score	**after** absolute score	**after** relative score	**improvement**
1	1	541	3.022	361	2.017	33.3%
1	2	657	3.335	469	2.381	28.6%
1	3	955	3.141	697	2.293	27.0%
2	4	1140	3.931	834	2.876	26.8%
2	5	1114	3.789	868	2.952	22.1%
2	6	1625	4.243	1291	3.371	20.6%

Table 1: *Results before and after the optimization routine on 10 layer prints.*

5 Conclusion

The question we focused on is how we could find a solution for configuring paths between logical gates and optimize these to a minimum length. As mentioned earlier, the order in which the nets are placed seemed to be a key element in finding valid solutions. However, only shuffling the nets did not do the trick. Tweaking the A* algorithm was an important step towards finding valid solutions, because the distance-biased A* algorithm managed to avoid closing in gates.

After doing so, our focus shifted to optimizing these solutions. With the tactics used for finding a solution in the back of our heads, we decided to keep focusing on the order of the nets being placed. We did so by implementing a custom genetic algorithm which made the optimization order-independent. The results were solutions with some significant improvements; the notable increase in relative score as the netlist length increases is due to the fact that all endpoints of nets are on the bottom layer. As the number of nets increases this means that bottom layers are occupied by the tail ends of paths that connect to the gates, which forces all nets to be slightly higher (and thus, longer) to avoid conflicts on the lower layers. The decrease of improvement that the genetic algorithm achieves can be explained similarly. Because all netlists are first solved on the same number of layers (10, in this case), the paths cannot be shifted down as much in denser configurations as opposed to sparse configurations.

Nevertheless, there is always room for improvement, both in performance and solution quality. The performance of the recursive method for finding a netlist order for example. The implementation presented here starts over when a conflict is found. This could be altered by not removing all nets, but random or neighboring nets until the current net can be solved, in order to reduce running time. Also, we expect that the quality of solutions could be improved by taking more properties of the paths into account, such as the number of corners, the longest distance travelled over the x or y axis, etcetera.

Because of the complexity of the case, we dealt with it in a rather specific way. This makes the implementation quite unique and difficult to generalize to other cases and problems. However, integrated circuits are used in a wide variety of devices that all use the same principle which our implementation obeys to.

References

[1] Paul Erdős. On a classical problem of probability theory. 1961.

[2] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

Appendix A Result Configurations

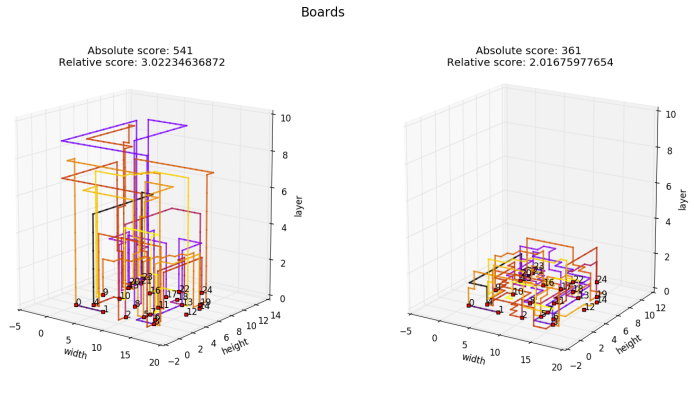


Figure 3: *Print 1, netlist 1, configuration before (left) and after (right) optimization*

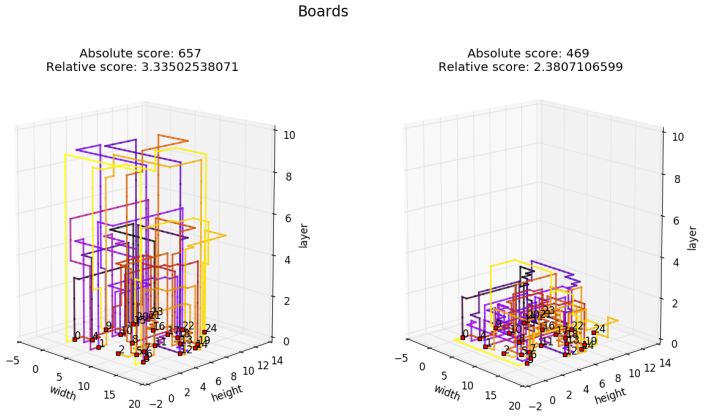


Figure 4: *Print 1, netlist 2, configuration before (left) and after (right) optimization*

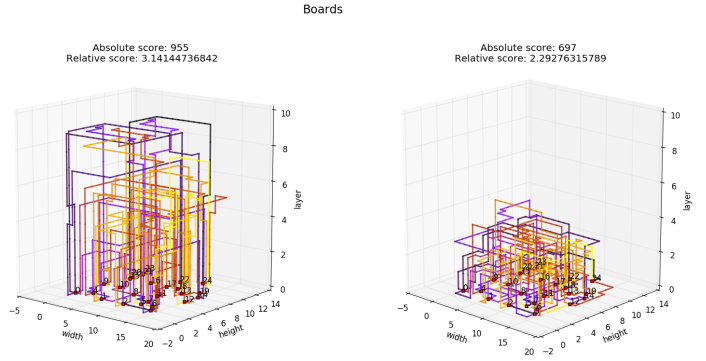


Figure 5: *Print 1, netlist 3, configuration before (left) and after (right) optimization*

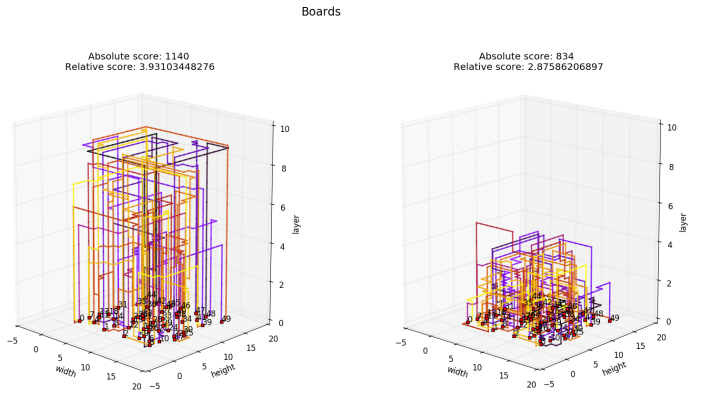


Figure 6: *Print 2, netlist 4, configuration before (left) and after (right) optimization*

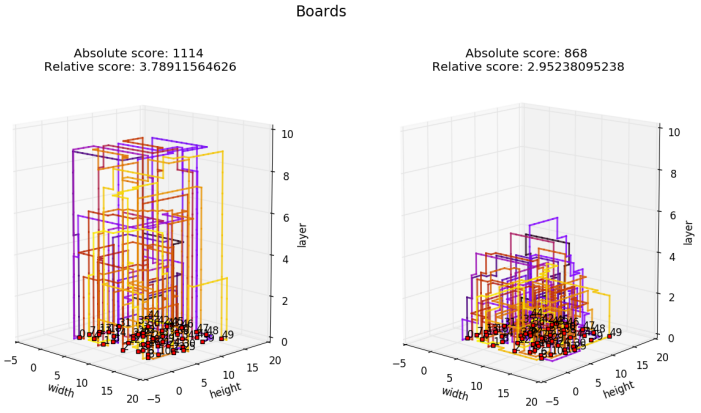


Figure 7: *Print 2, netlist 5, configuration before (left) and after (right) optimization*

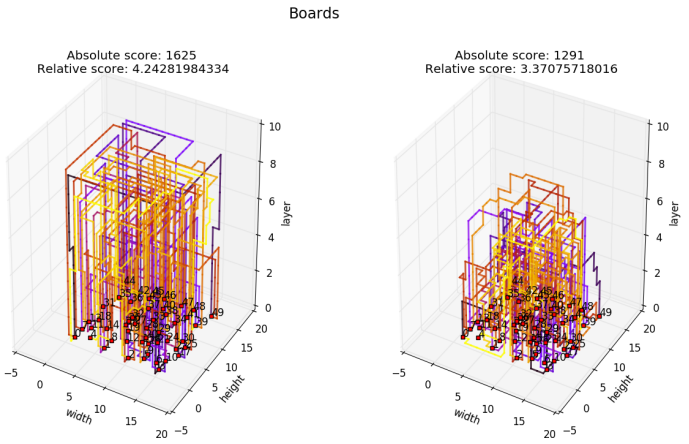


Figure 8: *Print 2, netlist 6, configuration before (left) and after (right) optimization*