Using an A* algorithm and heuristic assumptions to optimize the collective length of multiple paths between gates on a chip.

Guido Visser, Laura Veerkamp and Floris Kuipers

December 2016

1 Introduction

The chips and circuits problem is, as called in computer science, a constraint optimization problem. The problem consists of two components: a print and a netlist. The print is a rectangular matrix of given size with numbered gates placed on points in the grid in a random fashion. The netlist is a list of tuples of gate-numbers, and indicates which gates have to be connected. The objective is to connect all the gates in the netlist without crossing lines. Drawing the lines isn't limited to the 2D xy-plane: lines can also be drawn upwards in the z-direction, up to a total of seven layers.
For this case, two prints were given. The first print (print 1) has dimensions 12x17x7, and the second print (print 2) has dimensions 12x16x7. On print 1 three netlists are given, one of length 30 (netlist 30) one of length 40 (netlist 40) and one of 50 (netlist 1-50). On print 2 there are three netlists as well. One of length 50 (netlist 250), one of length 60 (netlist 60), and one of length 70 (netlist 70).

1.1 Complexity of the Problem

The complexity of this problem can roughly be understood as the amount of possible different path configurations given a grid. The amount of lines in a 3-dimensional grid is calculated by equation 1.

$$z[3xy + 2x + 2y + 1] + 2xy + x + y \tag{1}$$

Here x is the width, y the length, and z the height of the 3-dimensional grid. The problem's grids are called print 1 (17x12x7) and print 2 (17x16x7), having respectively 5067 and 6758 lines. Calculating the different configurations in which these lines can be occupied or not is 2^{lines}, respectively 2^{5067} and 2^{6758} for the grids.

These are very generous upper bounds because they do not take into account two factors: that lines need to be continuous paths and that lines cannot cross. These factors would greatly limit the possibilities of configurations.

The lower bound is calculated by summing the Manhattan distances (MDs) between two gates for each tuple of a netlist, as shown in equation 2.

$$\sum_{i=1}^{n} \left(\begin{array}{c} \Delta x_i + \Delta y_i \\ \Delta x_i \end{array} \right) \tag{2}$$

Here Δx_i is the difference in x-coördinates, as is $\Delta\ y_i$ the difference in y-coördinates. n is the length of the netlist.
This problems statespace is sufficiently large for brute force algorithms to be ineffective. Other algorithms are required to solve this problem.

2 The A* Algorithm

The A* pathfinding algorithm is used to find the cheapest path between two points in the matrix, where the cost is mainly determined by the distance between two points. Study shows that A* outperforms other pathfinding algorithms such as Dijkstra's algorithm, Bellman Ford's algorithm and Floyd-Warshall's algorithm[1]. A line drawn by A* in the matrix is considered a wall for the next path and so forth, thus not allowing lines to cross. The A* algorithm does not simply take the shortest MD between two points. It also takes into account extra heuristic costs considered to improve the success rate of the algorithm. The pathfinding is therefore dependent on these heuristics.

2.1 Stochasticity of the PriorityQueue

A PriorityQueue is used in the A* algorithm to choose the most promising position to find the path, i.e. the position with the lowest rating. However, the name queue is misleading. It seems that when multiple entries have the same rating, the standard PriorityQueue in python's Queue library functions stochastically, instead of choosing the first entry. This is likely caused by multi-threading: when calculations are distributed over multiple cores in a processor their results are returned by these processors not all at once but, uncontrollably, one at a time. This means that if multiple paths are possible as shortest path, A* will not prefer the same path each time it runs because some options are prompted earlier than others. Different results are therefore obtained each run, since a small change in paths at the beginning of the netlist can, according to the butterfly principle, result in a large change in the final result.

The stochasticity of PriorityQueue was suppressed in one part the results analysis. In another, it appeared possible to leverage the effect. By doing multiple runs of the algorithm with a stochastic PriorityQueue, the outcomes of the algorithm form a distribution. This distribution can be analysed for the influence of the specified heuristics: netlist order and children/grandchildren costs.
For the rest of the paper it is assumed that the reader knows how an A* algorithm works.

2.2 Heuristics

The netlist ordering is an important deterministic factor in A*. Looked into are four proposed orderings, each possibly helpful in solving the netlist. The first, for reference, is the original netlist ordering as provided. The second and third orderings are done on the basis of the distance between gates in a tuple. In the second ordering tuples with gates that are closest to eachother are put in the front. The third ordering puts tuples with gates that have the most distance between them at the front. This third ordering might be successful because the paths that will later be obstructed mostly are laid down first. The opposite can also be claimed: that the shortest-first ordering works well because these are less likely to obstruct other paths.
Lastly a fourth ordering is, in the spirit of keeping gates clear which have to be connected by the netlist still, paths for gates occurring most frequently in the netlist first. If the gates in a tuple occur equally often, then the shortest of those paths is laid down first.

In order to enable the A* algorithm to find a solution (to process the full netlist), extra costs of direct neighbours and indirect neighbours of gates were considered. Direct neighbours (children), had a MD of 1 to gates, and indirect neighbours (grandchildren) a MD of 2. The extra costs in proximity to gates would help to keep the space around gates clear of paths, which is important if the gate is still to be connected in upcoming tuples. Various costs for children and grandchildren are considered.

3 Method and Results

Two major approaches were used to get results. Firstly, the stochastic nature of the PriorityQueue is leveraged, by running A* with specific netlist orderings. This is desirable to get a large quantity of samples in a relatively fast pace, which in turn can lead to a good understanding of what extra costs should be assigned to positions around gates.

Secondly, the stochastic nature of the PriorityQueue is suppressed.In this analysis an extra algorithm is used to swap the sequence of the netlist, while A* is running. A swap happens when A* can't lay down a path. Then the last path is deleted and is put back at the end of the sequence. This ensures a higher chance of finding a solution. This experiment is done to get a better understanding of the influence of the xyz sequencing of steps (e.g. first y, then x, then z) and how costs around gates affect the success rate of finding a solution.

3.1 Analysis with a stochastic Priority Queue

The stochastic nature of part of the algorithm implies that one result comes out one time, another in a second run. The influence of the netlist order thus can be inferred from a number of reruns. The original, long to short and short to long netlist orderings are analysed in figure 1. The average success of A* (how far in the netlist did it get) was taken over 50 runs of netlist 30. Ordering the netlist short to long gave the best average result of 23 paths that were possible to lay down. There was improvement to be done however, since no run had completed all the paths in the netlist.

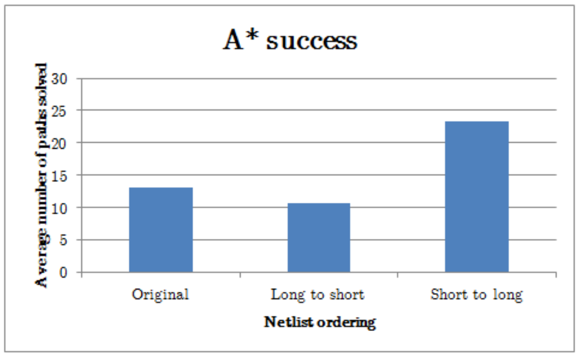

Figure 1: The average success of A* with differently sorted netlists, for the netlist of length 30. A* was run 50 times.

What prevented the netlist to be processed completely in all netlist sorts was the enclosing of gates by laid down lines, as can be seen in figure 2. These gates could not be reached for future connections. Because of this, higher costs around gates were considered for children and grandchildren of gates, for both the short to long netlist ordering and the ordering according to gate occurrence. If only children of gates are increased in costs, a value of 1 is chosen, which will result in a path being laid down according to figure 3A. If both children and grandchildren are assigned extra costs, values of 25 respectively 1 can be chosen, resulting in figure 3B. In analyzing the influence of different surrounding costs, also cost combinations of 13 and 1 and 29 and 2 are chosen. These costs interfere both less and more radical in the pathfinding.

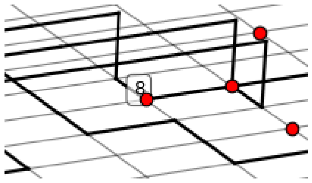

Figure 2: Example of gate 8 enclosed by other lines.

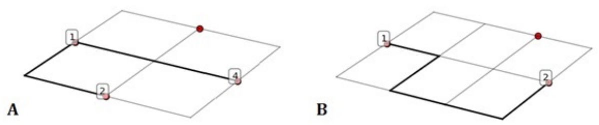

Figure 3: Example of how costs around gates influence pathfinding. In 3A, path 1-2 preferably goes down first and then to the side, to leave room for line 1-4. The extra cost of 1 of the child of the nearby gate ensures this. In 3B, an extra cost of 21 is added to children of gates in order to make up for preferably traversing back and forth, and another four for the four times grandchildren costs of 1 need to be overcome as well in order to keep more space available for the upper gate.

As can be seen in figure 4, applying a cost of 1 to children of gates already resulted in resolving more netlist tuples than without extra costs. The amount of resolved tuples increased further with an increase in child and grandchild costs. However, the algorithm became a lot slower the higher the costs for children and grandchildren got, and foremost, only twice did the netlist get solved completely: for the costs setting 25 and 1. Therefore the fourth sort is introduced: laying down paths for gates occurring most frequently in the netlist first. The result of this new ordering is also shown in figure 4. In higher children/grandchildren costs this netlist ordering almost always return a full solution.

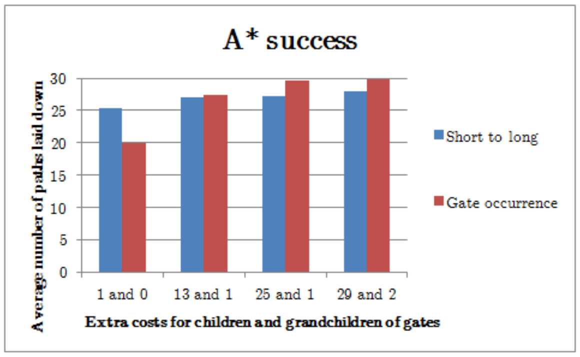

Figure 4: The average number of paths laid down by A* with increasing costs for children and grandchildren of gates for two different netlist sortings. With a limited interference of gate children costs (only 1 extra), the netlist sorted short to long runs best, while shortly the netlist sorted on gate occurrence takes over.

In examining the quality of full solutions by A*, the total path length found was divided by the lower boundary of path lenghts to obtain the percentage it took the solutions more than the (unattainable) lower boundary. Netlist 30 and 40 were examined. The results are shown in figure 5. Even though the total path length was divided by 291 for netlist 30, and 341 for netlist 40, being their lower boundaries, there was only one instance where netlist 40 took a lower percentage than netlist 30 on top of the lower boundary to complete the netlist (46,3% vs. 46,7%). All the other runs confirm that longer netlists need respectively more path length to successfully complete the run.

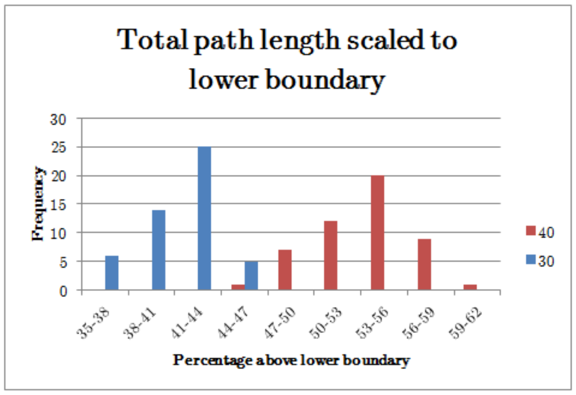

Figure 5: The total path lengths obtained in 50 successful runs for netlist 30 and netlist 40 are compared by taking these lengths as a percentage above the lower boundaries: 291 for netlist 30, 341 for netlist 40. Child and grandchild costs are 29 and 2. Netlists were sorted according to gate occurrence.

3.2 Analysis suppressing the stochastic nature of PriorityQueue

In the second part of the A* run analysis, the stochastic nature of the PriorityQueue is suppressed. In this analysis an extra algorithm is used to swap the sequence of the sequence of the netlist, while A* is running. A swap happens when A* can not find a path. The previous path is deleted and put back at the end of the netlist. This presumably ensures a higher chance of finding a solution.

This algorithm is not exhaustive, because it is possible to do a swap that creates a sequence that was already encountered before. If this happens the algorithm enters an infinite loop. Therefore, if a loop is detected, the algorithm is cut off. If coincidentally the algorithm chooses a path that never enters a loop, the deepest it will go is N!, where N is the length of the netlist. Furthermore in worst case the runtime of the algorithm is linear. In other words the algorithm has O(N!).

This experiment is done to get a better understanding of the influence of the xyz sequencing of steps (e.g. first y, then x, then z) and how costs around gates affect the success rate of finding a solution.

The rearrange algorithm was used on a netlist that was sorted according to frequency of gates. At first no extra costs have been added to children and grandchildren of gates in order to get a reference. No solutions have been found. But, since the algorithm has a worst case runtime of O(n!), with n being 30, it had not run through all possibilities before it was cut off. As can be seen in figure 6, it is found that the sequencing of x, y, and z has little effect on the rate in which a netlist is laid successfully. Henceforth all results will be based on A* runs with steps in the x-direction first, then the y-direction, and then the z-direction.

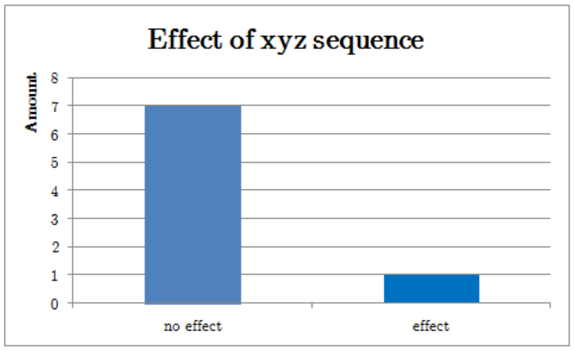

Figure 6: The number of runs where xyz sequence had influence on the success rate of the algorithm

For the netlists with length 30, 40, and 50 multiple runs have been done with different costs for both the children and grandchildren of gates. All perturbations of the values (1, 6, 11, 16, 21) have been checked. These values have been chosen according to the following formula,

$$c = \sum_{i=0}^{4} xi + 1 \tag{3}$$

where c is the extra cost assigned to a position and x is half the cost for a single step on the grid. A normal step on the grid costs 10. Since it costs 2 steps extra to move around a position, a path will try to go around a position with length cost 21, or a position with twice an extra cost of 11.

It is found that the rate of success in which A* finds a solution is affected by these parameters. This can be seen in figure 7.

		Children				
	30	21	16	11	6	1
Grandchildren	21	24	24	24	24	24
	16	29	29	29	30	30
	11	29	29	29	30	30
	6	29	29	30	29	25
	1	29	30	30	29	29

A

		Children				
	40	21	16	11	6	1
Grandchildren	21	40	40	40	40	40
	16	39	39	36	31	36
	11	39	39	39	36	36
	6	39	38	39	31	39
	1	39	37	37	31	31

B

Figure 7: This table shows each combination of costs between the children and grandchildren of gates, and how many paths could be laid out by A* with the netlist of length 30 (A) and 40 (B).

In the tables, the upper left corner is a large extra cost for both children and grandchildren, and the lower right corner is a low cost for both children and grandchildren. With the netlist of length 30 (figure 7A) it is shown that extra costs are most effective when they have values that aren't very large or low, whereas in the netlist of length 40 (figure 7B) it is shown that the cost around children of gates is less relevant than the cost around the grandchildren and a large cost for the grandchildren around gates works very well. This is likely the result of how crowded the netlists are. It is easier to solve a smaller netlist, and therefore a small adjustment is needed to get a solution, whereas a larger netlist is more difficult to solve, and therefore a larger adjustment is needed.A larger adjustment can be made in two ways: by a higher extra cost or a larger area around the gates with extra cost. This means a high cost for grandchildren is a larger adjustment than a high cost for children.

The same experiment was done with netlist 50 of print 1, but no solutions were found. In order to save time the rearrange algorithm was cut off at 200 swaps.

4 Discussion

Applying extra costs to children and grandchildren of gates proved to be most effective for the netlist sorted to gate occurrence. Chances of the netlist being solved increased dramatically. Laying down paths for gates that have to be kept clear first seemed like a good approach to higher the success rate of A*.

In examining the nature of the problem with netlists of different lengths, it appeared that a longer total path length is needed to lay down the full netlist, even if a correction for the lower boundary is applied. This indicates that, as was already theorized, the netlist complexity increases extensively with netlist length.

The sequence in which steps are made in the x, y, and z direction by A* proved to have little influence on the result, and in order to get more results on the influence of extra costs for positions around gates, all experiments where done with the same sequence of steps. This is a good way to get data on the effectiveness of extra costs, but it it important to note that the sequence of step directions affects the outcome of A*.

Looking at the visualisation of A* gives insights in possible improvements to obtain even lower total path lengths. Figure 8 shows part of the solved netlist 30 grid, with the purple line taking a detour around a gate which is already connected and doesn't need any more of the space around it. If A* was to rerun for only this line, without extra costs around other gates and the other lines appended as walls, it would return a straight line. This procedure could be repeated for each line, optimizing total path length.

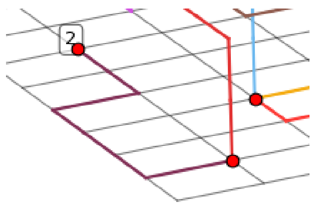

Figure 8: The purple line takes a detour while the gate on the right doesn't need any more spacing.

The sequencing of the netlist is a determining factor in finding the most optimal solution. In this experiment there was no effective way to check large quantities of sequences, because the rearrange algorithm had a O(n!). There is no guarantee that A* provides the best approach/solutions to this problem. In this area a lot of further understanding of the problem can be obtained by testing other algorithms. Since the complexity of the netlist sequence is n!, with n being the length of the netlist, brute force algorithms cannot be used to find the optimal solution. If an effective heuristic can be thought of, a beam search can be used to solve this issue. Next to that a breadth first or depth first algorithm with the right pruning could also work.

In order to get a better understanding of the effect of extra costs around gates it would be lucrative to run the algorithm with even more values for the extra costs. For larger netlists, even higher costs can be tried.

Not all netlists were tried out, and netlist 2-50 was not solved. In a more complete research into the case, these instances can be solved as well. It is likely that larger netlists behave differently than the smaller netlists, regarding the costs around gates. Therefore solving these netlists could give a more complete insight into the case and the principles discussed in this paper.

5 Conclusion

Both approaches show that assigning extra costs to children and grandchildren of gates increases the chances of solving a netlist. It is apparent that different costs affect netlists of varying sizes in different ways. If a netlist is relatively large, higher costs are more effective, while smaller costs are more effective on smaller netlists.

References

[1] Jain L.C. Finn A. et al Sathyaraj, B.M. Fuzzy optim decis making. 2008.