Chipmunks

Guido Visser, Laura Veerkamp
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 a 2D-plane: lines can also be drawn upwards in the z-direction. The height of the z-direction is limited to seven layers. A final note here is that the gates are located on the bottom z-level, lines can't be drawn below it.

1.1 The AStar Algorithm

We chose the A* pathfinding algorithm to find the shortest paths between two points in the matrix. The lines drawn in the matrix are considered walls in future runs of the algorithm. Therefore, no paths can cross each other. The A* algorithm doesn't simply take the shortest route between two points distance-wise. It takes the cheapest path, influenced by heuristic costs. The pathfinding is therefore dependent on these heuristics.

1.2 Stochastic Priority Queue

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. Often times there are multiple path-choices that are equally good choices, i.e. a step in the x-direction could be equal in cost as a step in the z-direction. The algorithm was expected to be consistent in choices it made between equal candidates. That this was not the case makes the results differ a bit while it runs with the same parameters.

1.3 Paper structure

This paper will report on the two methods used for gathering results. The first method uses the stochastic nature of the priority queue to test a set of parameters by running it a number of times. These parameters will vary the netlist ordering and values of heuristic costs.
In the second method the stochastic nature of the algorithm is suppressed and partly the same parameters are tested this way. However this second method also tests a swapping algorithm that changes the netlist ordering while the Astar algorithm runs.

2 First Method

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 had to be inferred from a number of reruns. Looked into are four proposed orderings. The first is the original netlist. The second is paths ordered longest to shortest, which is arguably profitable for the success of the A* run, as the paths that will later be obstructed mostly are laid down first. The opposite can be claimed as well: that the shortest paths, which are less likely to obstruct other paths, are best to handle first. The results of these first three proposed orderings are shown in figure 1. 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.

What prevented the netlist to be processed completely in all netlist sorts was the enclosing of gates by laid down lines. These gates could not be reached for future connections. Because of this, higher costs around gates were considered for direct, first grade neighbours (children of gates), and second grade neighbours (grandchildren of gates). 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 2A. If both children and grandchildren are assigned extra costs, values of 25 respectively 1 can be chosen, resulting in figure 2B.

For a less dramatic effect, also the extra cost combination of 13 and 1 is analysed in the A* run of netlist 30. More strict run conditions are set for 29 and 2 (with an extra 8 to overcome costs for grandchildren).
As can be seen in figure 3, 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 a new netlist sort was considered. This is where the fourth sort is introduced: laying down paths for gates occurring most frequently in the netlist first. If the gates in a tuple occur

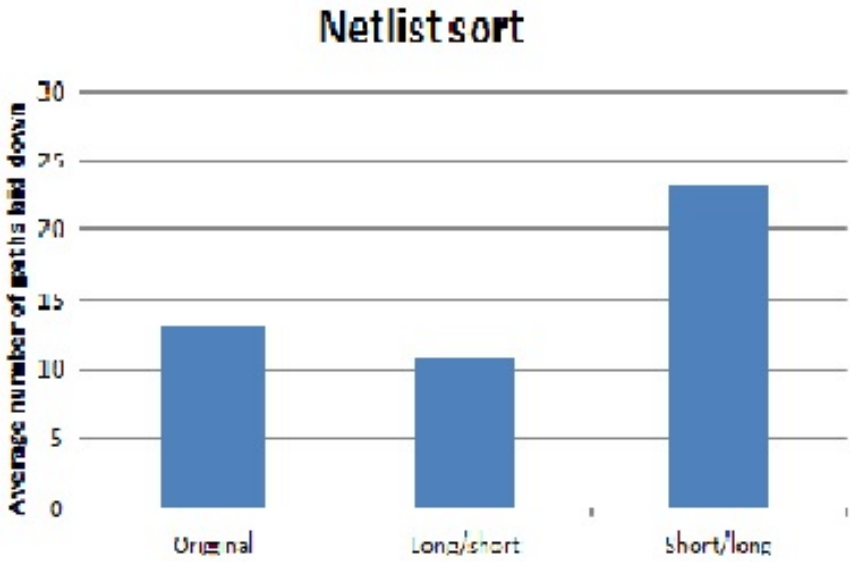

Figure 1: The average success of A* with differently sorted netlists, for the netlist of length 30. A* was run 50 times.

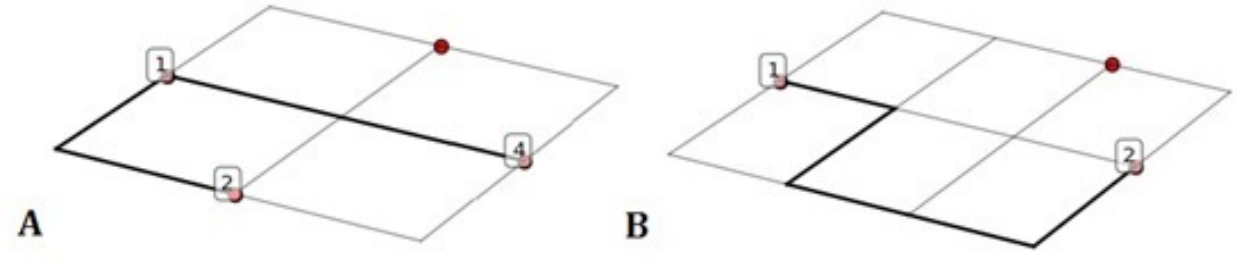

Figure 2: Example of how costs around gates influence pathfinding

equally often, then the shortest of those paths will still be laid down first. The result of this new ordering is also shown in figure 3.

3 Second Method: Suppressing the Stochastic PriorityQueue

Assuming the output of priorityqueue doesn't vary and the costs of positions around gates are constant. Finding a solution is only dependant on the sequence

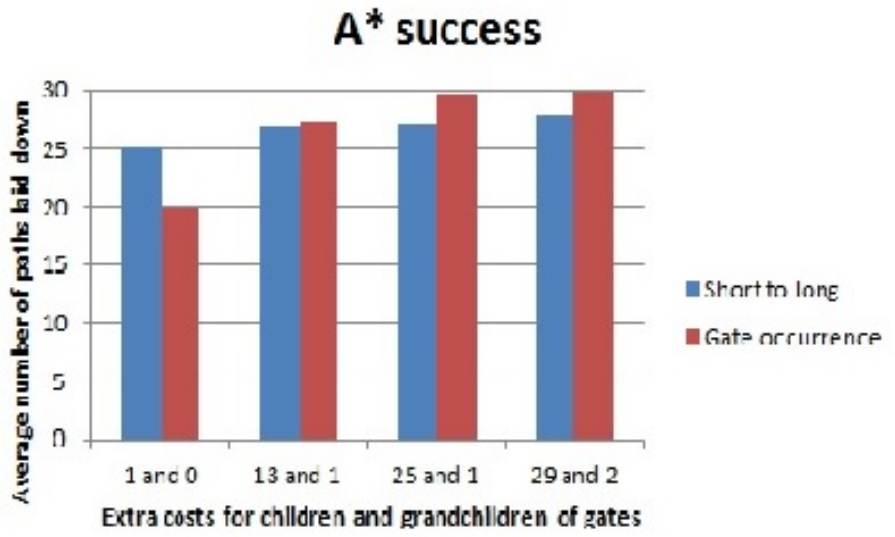

Figure 3: The average number of paths laid down by A* with increasing costs for children and grandchildren of gates for two different netlist sortings. With a limited interference of gate children costs (only 1 extra), the netlist sorted short to long runs best, while shortly the netlist sorted on gate occurrence takes over.

in which the program runs A*. For instance, when the netlist consists of three connections: (A, B, C). Running A* on A first, then B, then C, might not give a solution, whereas (B, C, A) might. To account for this, an algorithm was written that rearranges the sequence while the A* algorithm is running. The advantages and disadvantages of this algorithm will be discussed in the discussion. In this section the results of this algorithm will be presented.

4 Discussion Conclusions

Applying extra costs to children and grandchildren of gates showed most importantly for the netlist sort to gate occurrence. Chances of the netlist being solved increased dramatically. First laying down paths for gates that have to be kept clear of other paths seemed like a good approach to higher the success rate of A*.

Looking at the visualisation of A* star runs already gives insights in possible improvements to obtain even lower total path lengths. Figure 6 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, no extra costs around other gates and the other lines appended as walls, it would return a straight line. This procedure

can be repeated for each line, optimizing total path length.

This being said there is no guarantee that A* has the best possible solution to the problem. A best solution is not necessarily found with A*. A* could guarantee the shortest path between two points but since an earlier connection determines the shape of later made connections it could be better not to take the shortest distance for earlier connections. The use of heuristics will make A* look for the cheapest path rather than the shortest path, with heuristics determining the cost.

Because of the nature of the algorithm, the sequencing in which paths are laid out is one of the determining factors to find a solution. With 30 connections in the smallest netlist and 70 in the longest netlist the amount of different netlist configurations ranges between 2.7×10^{32} and 1.2×10^{100}. Finding an optimal solution would involve an algorithm that efficiently works through this statespace.