Ty Davis
Dr. Justin Jackson
ECE 5110
April 20, 2025

Algorithms for Layout and Routing of VLSI Design

One of the key things that I've learned in this class is that a wealth of time working on a VLSI project can be spent on the physical design and layout of a chip. As with any task in engineering, efficiency and efficacy in VLSI layout improves with practice, but with modern transistor counts in computer chips approaching the hundreds of billions, manually placing each transistor becomes impossible. While many of the design structures are repeated in large chips, such as the GPUs and CPUs that boast such impressive transistor counts, hundreds of thousands of decisions about cell location and layout need to be made for the design of one chip. With each decision affecting multiple important factors that should be considered in VLSI design, using automated routing and layout is necessary in the modern day.

Much of the motivation for the design of a chip comes down to speed/performance, the power consumption, and how easy the chip is to manufacture. Attempts to reduce the total length of interconnects is a large priority in automated routing, and decreasing unnecessary capacitance improves the power-consumption, and therefore the heat produced by the chip.

In this essay I will talk about some of the interesting and important concepts that have emerged over the recent decades that pertain to automatic layout and routing techniques. Many of the techniques in VLSI design are described as "NP-complete" or "NP-hard problems" ("34 The Complexity of Theorem-Proving Procedures (1971)"), where NP means "non-deterministic polynomial-time" – referencing the amount of time/compute power required to verify that a solution is valid is polynomial-time. Notably, the problems of cell placement, floorplanning, routing, etc. would take so much time to solve computationally that finding a perfect solution is not viable. As such, many of the proposed techniques rely on heuristics and approximations to find a near-complete answer.

Maze Routing Algorithms

Lee's Algorithm

Lee's Algorithm was one of the first algorithms used for grid-based automatic routing and was made in the early 1960's. It was initially an endeavor to answer the question, "Is it possible to find procedures which would enable a computer to solve efficiently path-connection problems inherent in logical drawing, wiring diagramming, and optimal route finding?" (Lee). In response to that question, Lee says, "The results are highly encouraging."

In today's standards the algorithm is rather simple, but it was originally implemented on an IBM 704 computer, and I believe that the idea is interesting enough to be explored here. The premise of the algorithm is that you have a set of cells in some space, and a way to determine each cells' neighbors. The shortest path between two cells, c^i and c^j can be determined in a breadth-first approach by iteratively considering all of the neighbors of c^i, and their neighbors, until you reach c^j. To dive a little deeper into the implementation of the algorithm, imagine a grid of cells of any size. Let c^i be on the left side, and you're trying to reach c^j, which is placed on the

right side. However, between c^i and c^j is some obstacle. Some of the cells between them are marked as an obstacle, representing another wire connecting two other traces.

Starting at c^i and marking it with the value 0, consider all of its neighbors (the four adjacent squares in the grid) and mark them with the value 1, assuming that none of them are blocked. Repeat this process for all of the neighbors, marking the subsequent neighbors with increasingly higher values (1, 2, 3, ...) until you reach the destination cell c^j. Now, from c^j, simply follow the cells marked with the lowest number back to the original starting cell c^i. The number that each cell is marked with can be interpreted as its distance to the starting cell c^i.

Lee's algorithm is effective, but because of its breadth-first nature, it becomes inefficient in memory and speed. A number of other algorithms have been shown to improve the performance of smallest path algorithms. Consider, as well, that Lee's algorithm only considers paths on a 2-dimensional plane, neglecting the use of metal-layers and vias. Beyond this, Lee's algorithm may offer the shortest path to whichever route is considered first, when a more optimal layout for the overall design may be achieved with an algorithm that considers which connections to make first. Of course, that's why routing usually comes as a later step, following placement and floorplanning.

A* Algorithm

The A* algorithm (pronounced "A-star") is an algorithm that essentially solves the same problem as Lee's algorithm, though in a more time and memory-efficient way. It achieves superior efficiency by estimating which paths are more likely to find the solution earlier, resulting in fewer cells checked.

The A* algorithm is implemented as follows. Imagining a similar grid situation as in the example when considering Lee's algorithm, the starting cell's neighbors will all be considered, and their "costs" estimated with a heuristic function. The heuristic function ($f(n)$) follows the from $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of getting to a certain cell from the starting cell, and $h(n)$ is the estimated cost of getting from that certain cell to the destination cell. You can think of it essentially as an expansion of Lee's algorithm, where Lee's algorithm considered only $g(h)$, neglecting the cost of each neighbor's path towards the destination cell.

Now, for each of those neighbors whose heuristic function f is lowest, you iterate and calculate the same for its neighbors. Unlike Lee's algorithm, instead of searching breadth-first in every direction from the starting cell, the A* algorithm avoids evaluating or estimating the cost of the cells which seem least likely to result in the shortest path to the destination cell.

A deeper look into the A* algorithm shows that it may not only be used for cells on a grid, but that it can be used for any graph composed of nodes connected by edges, where the cost of each edge is known. For example, explored in the paper by Hart et al., we see their example of the implementation of the A* algorithm for cities connected by roads. The algorithm may use the calculated cost $g(n)$ (found as it traverses the graph and sums the edge-costs) and the estimated $h(n)$ "might be the airline distance between city n and the goal city."

Manhattan vs Non-Manhattan Routing

As of the early 2000's, "most routing algorithms use[d] a Manhattan geometry with horizontal and vertical traces" (Stan et al.). Such designs prevailed because of their simplicity, but the

increase in wire length (27% longer on average than the Euclidean distance) of the interconnections resulted in significantly slower circuits. The concept of "non-Manhattan" routing, therefore, is any routing that doesn't strictly adhere to a grid.

Non-Manhattan layout designs can be applied not only to VLSI design projects, but also PCB and other circuit layouts. The study cited above showed the use of two different Non-Manhattan layout algorithms that routed wires on horizontal, vertical, and 45° diagonal lines, but other non-Manhattan layout paradigms exist which operate on other bases. While the results from that study can only be applied to the specific layout algorithms tested, it is safe to say that Non-Manhattan layouts should always be a consideration when trying to decrease average interconnection length.

As shown with the A* algorithm, routing doesn't necessarily have to take place on a strict grid of cells. Non-manhattan graphs can be used in combination with the maze-routing algorithms to more effectively plan routes and reduce the total interconnect length.

Miscellaneous Techniques

There are other widely used techniques in routing and planning that can be utilized in VLSI design that I felt deserved to be mentioned here.

River-routing is the concept of laying out the connections between cells such that the paths don't have to cross, allowing the lines to look like a parallel streams, or the illusion of a river.

X-routing and Y-routing is a similar technique where you use two different metal layers in different directions. For instance, a metal1 layer may be used East-West (on the x-axis) while a metal2 layer is used from North-South (on the y-axis).

Placement and Planning

The planning algorithms we've considered so far are limited in their direct application to VLSI routing and planning. Maze routing algorithms consider only how to connect two points within a 2-dimensional space, or a space that can otherwise be demonstrated in the form of a graph. Such algorithms don't consider the use of vias, other metal layers, nor do they consider more than one path at a time.

There are more techniques within graph theory that can be used to improve and optimize the layout of a design, and they may have a more direct impact on the overall VLSI design that is ultimately achieved.

Partitioning with the Kernighan-Lin Algorithm

One of the first things that should be done in the physical design process, considerably earlier than the maze routing signals above, is partitioning the design. In the perspective of VLSI design, imagine a large netlist showing the connections between different logic cells in a graph configuration. Choosing how to place which cells closer to which can be a daunting task to do by hand, and a good first step might be separate all of the cells into smaller groups so that you can decide how to place them in each group. Such a process is called "partitioning", and the goal of an effective partitioning algorithm is to keep the total number of connections between each partition to a minimum.

The Kernighan-Lin algorithm in an effective algorithm that relies on heuristics to produce good solutions in a reasonable amount of time. The steps of the Kernighan-Lin algorithm are once again straightforward. An example follows that shows how to split a graph into two equal (or about equal) subsets while minimizing the number/cost of edge cuts between the two subsets.

First split the graph into subsets A and B. With both subsets separated, an estimate of the improvement that would occur if you swapped each of the nodes to the other subset is computed. Of course, calculating the cost for each of the nodes for each of the subsets possible in unrealistic beyond trivial examples, so an estimation is necessary. In order to estimate the gain of swapping a given node to the other subset, the following are calculated. The "internal cost" ($I(v)$) is the sum of edge weights to nodes in the same set. The "external cost" ($E(v)$) is the sum of edge weights to nodes in the other set. The D-value is then shown by $D(v) = E(v) - I(v)$, this is how much the cost would improve by moving node v to the other side. A D-value is essentially an estimate of how much better the cost would be if the node v were in the other subset.

After computing D-values for all of the nodes, find which nodes are best to swap between the two subsets. The equation used to do this is $g(a, b) = D(a) + D(b) - 2 \cdot w(a, b)$, where $w(a, b)$ is the edge-cost between the two nodes a and b if there is one. Subtracting that term from the equation essentially disincentivizes swapping two nodes if they are connected to each other, assuming that there is a better swap available elsewhere.

This algorithm is effective because it attempts "to identify [the nodes to swap] from A and B, without considering all possible choices" (Kernighan and Lin).

Rip-Up and Reroute

While partitioning certainly helps to solve the problem of routing and reducing the amount of crossing and overall interconnect length between nodes, the maze-routing algorithms still suffer from some problems. Most notably, they might suffer from significant congestion in high traffic areas, and a manual technique for resolving such conflicts is identifying which nets are problematic and manually rerouting those. Or, as could be said, "ripping" and "re-routing" those nets. There have been attempts to replicate the human behavaior of rerouting with automated systems.

When maze routing fails (for example, the path has become completely blocked by other nets), an example reroute effect index for evaluating whether a net should be rerouted can be shown with the following equation: $P(W) = \min\{N_A(W) - L_A(W), N_B(W) - L_B(W)\}$ (Kuh and Ohtsuki). This $P(W)$ is caluclated for each net W in the region.

In this equation, W represents a blocking net in the area, and N_A is the number of cells in that blocking net that could be reached by the source cell of the new path we are trying to reroute. Accordingly, N_B is the number of cells in that blocking net that could be reached by the destination cell of the new path. L_A is 1 if the both of the pins of the blocking net are visible to the source cell of the new path, otherwise 0. L_B is similarly defined. Utilizing this expression for $P(W)$ we can determine whether attempting to reroute the net is even worthwhile. If $P(W) >= 1$, then it is worth attempting to reroute, otherwise no improvements are possible for that net W.

Rip-up and rerouting is a technique that still has room to improve, especially as newer techniques utilize the analysis of multiple nets at a time, where this technique currently shows some weakness. The implementation used here is just an example of one rerouting technique that

has been implemented in some cases, and can be a basis for other possibly more complex rip-up and rerouting techniques.

Simulated Annealing

The technique of "simulated annealing", much like the A* algorithm, has many uses beyond VLSI design, and being such a versatile tool can actually be used in multiple places in VLSI design, so it should be mentioned here.

Simulated annealing can be used in floorplanning, cell placement, and even detail routing, as it's just a technique that wraps around an algorithm, rather than replacing it. As a process, it avoids local minima by allowing iterations on the starting conditions of an algorithm to vary widely at the start, and more narrowly as computation continues. It gets its name from a processes in metallurgy known as "annealing" where materials are heated and cooled to reduce defects.

On the high-level, here is how it works. Starting at some initial condition, the algorithm iteratively explores the solution space by making small random changes. After a given iteration, the solution is either accepted or rejected. If the conditions improve, then it is accepted, but if the conditions worsen the solution isn't necessarily rejected. The earlier iterations have a higher "temperature" value which gradually decreases as the iterations go on. The higher the temperature value, the worse that a potential solution can be while still being selected. By allowing potential worse solutions to be accepted at the start, simulated annealing increases the chances of escaping local extrema, in an effort to find the global minimum or maximum of the function.

Simulated annealing finds such a useful place in VLSI design because many of the problems are NP-hard problems, where the compute time to find an optimal solution is unrealistic. An example using simulated annealing in VLSI could be in floorplanning, where exhaustively selecting the locations of hundreds of thousands of cells would be computationally impossible. So, the algorithm would suppose starting from a base condition where all of the cells are placed randomly on a substrate. After analyzing the possible solutions, moving some of the cells randomly (or heuristically) the solution is then analyzed. If the condition improves it is selected. But, based on the process of simulated annealing, if the temperature value is still high (e.g. it is an early iteration), a worse condition may be selected. Possible worse floorplanning will be chosen an iterated on, as this will reduce the chance of falling into a local extrema.

Machine Learning

Modern-day improvements in layout and routing are being explored through the use of machine learning. Machine learning is a massively expanding branch of computer science that has proven to tackle large-scale problems like these with lots of success. Many forms of machine learning exist that can be used for problem solving, each with their pros and cons.

Reinforcement Learning

Reinforcement learning (RL, or DRL for deep reinforcement learning) is a form of machine learning where an agent will perform a task (such as floorplanning or routing) and then the solution will be evaluated and given either a punishment or a reward. The agent then "learns" from that reward. The solutions that are given rewards will be replicated and iterated on, while

those that are not will be rejected in future iterations. Hundreds of thousands of examples will be used in order for the neural network to effectively learn and perform in a way that fits design needs. A group from the Google Chip Implementation showed their results and methods in a 2020 paper. They mentioned that the reward system for their DRL was based on “wirelength, because it is not only much cheaper to evaluate, but also correlates with power and performance (timing)” (Mirhoseini et al.).

The paper outlined performance improvements that resulted from using RL instead of simulated annealing processes. On average, wirelength and congestion decreased, meaning that overall power consumption and performance would be improved.

Conclusion

As chips become increasingly complicated, and accordingly the layout becomes increasingly complex, the methods for layout and routing in VLSI design become progressively more difficult. Decades of research and exploration have resulted in optimized algorithms and methods for automating the design of VLSI chips. As Moore’s law becomes increasingly difficult to maintain, as well as manufacturing processes getting increasingly difficult, optimized routing and the decrease of interconnect wires will be essential. Machine learning methods have proven effective in case-studies and may be an extremely pivotal technology in the future of VLSI design.

Works Cited

"34 The Complexity of Theorem-Proving Procedures (1971)." *Ideas That Created the Future: Classic Papers of Computer Science*, 2021, pp. 333–38.

Hart, Peter E., et al. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, 1968, pp. 100–107. https://doi.org/10.1109/TSSC.1968.300136.

Kernighan, B. W., and S. Lin. "An efficient heuristic procedure for partitioning graphs." *The Bell System Technical Journal*, vol. 49, no. 2, 1970, pp. 291–307. https://doi.org/10.1002/j.1538-7305.1970.tb01770.x.

Kuh, E.S., and T. Ohtsuki. "Recent advances in VLSI layout." *Proceedings of the IEEE*, vol. 78, no. 2, 1990, pp. 237–63. https://doi.org/10.1109/5.52212.

Lee, C. Y. "An Algorithm for Path Connections and Its Applications." *IRE Transactions on Electronic Computers*, vol. EC-10, no. 3, 1961, pp. 346–65. https://doi.org/10.1109/TEC.1961.5219222.

Mirhoseini, Azalia, et al. "Chip Placement with Deep Reinforcement Learning." *CoRR*, vol. abs/2004.10746, 2020. *arXiv*, arxiv.org/abs/2004.10746.

Stan, M.R., et al. "Non-Manhattan maze routing." *Proceedings. SBCCI 2004. 17th Symposium on Integrated Circuits and Systems Design (IEEE Cat. No.04TH8784)*. 2004, pp. 260–65, https://doi.org/10.1145/1016568.1016637.