# Final Proposal: Chip Routing Solver Proposal

James Mastran (jam2454)

November 13 2025

## 1 Introduction & Motivation

As circuitry advances and the number of transistors that can be placed on a single chip increases, more components must be considered within the circuitry. In particular, finding paths to connect pins with wires in the circuit can pose challenges.

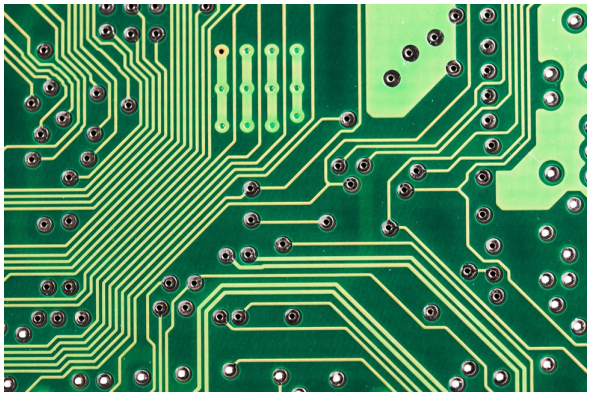

Figure 1: *Manual or interactive routing, advantages and inconveniences*

Identifying how to or the best way to connect a starting pin and a target pin with a tiny wire can be difficult when there are many other connections and components creating obstacles. While it ranges, there can be billions of connections on a single chip. Compounding the complexity, it is sometimes desired to connect a single pair of pins through multiple non-intersecting paths. This concept is known as "routing" and relates to integrated circuits (ICs) and very large scale integration (VSLI).

This is described in more detail in "Node-Disjoint Paths on the Mesh and a New Trade-Off in VLSI Layout":

> "A number of basic models for VLSI layout are based on the construction of **node-disjoint** paths between terminals on a multi-layer grid. In this setting, one is interested in minimizing both the number of layers required and the area of the underlying grid."

While routing, there are a few goals and strategies to keep in mind:

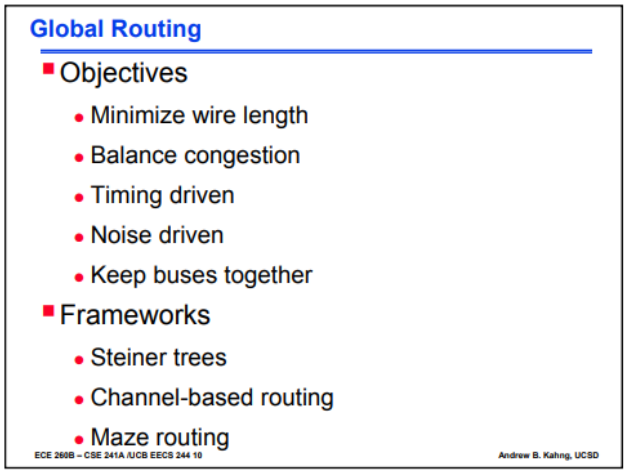


Figure 2: *Routing in Integrated Circuits*

In practice, chips often have multiple layers to help solve this difficult problem.

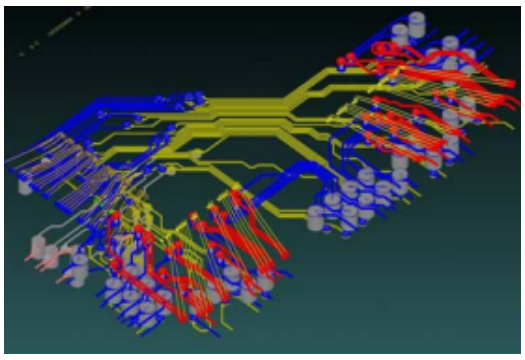

Figure 3: *Electromagnetic and Circuit Co-Simulation and the Future of IC and Package Design*

## 2 Objective

For the purposes of this project, we will focus on reducing the number of layers by finding a specified

number, $k$, of disjointed paths between a selected pair of pins (e.g. the start and the target node). We will also focus on minimizing cost, or equivalently, we will aim to minimize the length of wire for all, $k$, disjointed paths between two specified pins. There is some inherent monetary and performance cost in using more wire than is necessary. To make the problem more realistic and more complicated, we will also account for obstacles, such as other components or wires on the IC, while finding paths.

To tackle this real world problem, we will represent the problem as a graph and use the Maze Solving idea to find paths. In the graph, nodes represent either pins or places where wires can be attached and edges represent paths in which wires may be placed. Removing an edge can represent a barrier or obstacle in which a wire cannot be placed.

# 3 Proposed Solution

In order to find multiple disjointed paths in a graph, we plan to use the Sequential Shortest Path (SSP) algorithm on an undirected graph. The SSP algorithm is useful because it will find the minimum cost for a desired "flow" amount, $k$, on a provided graph. Each edge has a specific capacity of flow it can withstand.

Relating it back to the routing problem, setting a capacity of 1 for each edge should give us a solution to the problem of finding $k$ disjointed paths in a graph. We can think of flow as the number of disjoint paths and minimizing cost as minimizing the length of wire:

```
algorithm successive shortest path;
begin
    x := 0 and π := 0;
    e(i) := b(i) for all i ∈ N;
    initialize the sets E := {i : e(i) > 0} and D := {i : e(i) < 0};
    while E ≠ ∅ do
    begin
        select a node k ∈ E and a node l ∈ D;
        determine shortest path distances d(j) from node s to all
            other nodes in G(x) with respect to the reduced costs c^π_ij;
        let P denote a shortest path from node k to node l;
        update π := π − d;
        δ := min[e(k), − e(l), min{r_ij : (i, j) ∈ P}];
        augment δ units of flow along the path P;
        update x, G(x), E, D, and the reduced costs;
    end;
end;
```

Figure 4: *Network Flows*

Where $e(i)$ is the excess flow of a node. Only starting and target nodes should have some excess or deficit flow, respectively. All other nodes should be considered balanced (e.g. $b(i) = 0 \quad \forall \quad i \notin \{start\_node, \quad end\_node\}$). Lastly, $x$ and $\pi$ is the psuedoflow and potential, respectively.

The SSP algorithm specifies that we must determine the shortest path distances from node $s$ to all other nodes. This part of the algorithm can be solved using dijkstra or A*. However, for our particular use-case, we do not need the distance from the start to all other nodes, but just from the start to the target node. Using inspiration from a previous semester's project, *Parallel Functional Programming Proposal: MazeSolver*, we plan to use the A* algorithm for this part of the SSP.

One final caveat is that the SSP algorithm finds edge disjointed paths, but we are seeking node disjointed paths. This can be simply solved by node splitting, which is the process of duplicating each node such that there is a *in* and a *out* node except for the start/target node. Providing a single edge with a capacity of 1 between these nodes will in effect yield a solution that is node disjointed ("Maximum flow problem").

# 4 Simplifying assumptions

While the SSP algorithm can work on non-grid graphs, for the purposes of this project, we will assume operating over a 2D grid. Therefore, each node/pin can be connected to at most 4 others. As a result, this also limits $k$, the number of disjoint paths, to a maximum of 4. This also means our graph will be sparse.

On a grid, diagonal movements will not be allowed; only movements along the horizontal or vertical path (e.g. the only valid movements can be "right", "left", "up", or "down") are permitted. This also gives us the opportunity to leverage the A* algorithm with Manhattan distance as the heuristic.

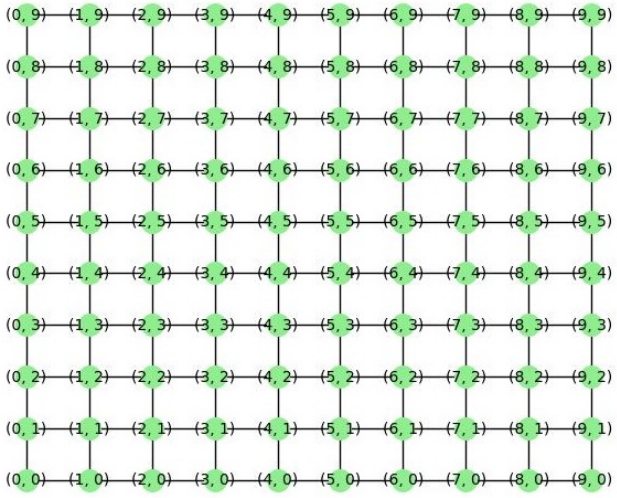


Figure 5: *Graph Theory - Infinite Graphs*

# 5 Acquiring a Graph

We will use a graph generator and parser to procure example graphs to solve. The generator will take in $N$, the number of rows, $M$, the number of columns, $p$, the probability that an edge exists between two nodes, and $k$, the minimum number of disjoint paths between the start and end node. If $p = 0$, then there won't be any additional edges added except for those to satisfy $k$ disjointed paths. Conversely, if $p = 1$, then the graph will be grid-like connected. The start and end node should be placed far apart such as at $(3, 3)$ and $(N - 3, M - 3)$. The generator will generate some file to be used by a graph parser.

A graph parser will then be implemented to convert the graph representation into a representation useable in Haskell, where node splitting will take place.

# 6 Sequential Speedup Ideas

1. Although it can increase complexity, especially during the parallelization phase, a way to speed up the A* algorithm would be to perform a bidirectional search. This would require maintaining two separate priority queues: one that maintains the path from the start to the target and another that maintains the path from the target towards the start. Additionally, two heuristics and a merging tactic would have to be adopted ("Bidirectional A Search with Additive Approximation Bounds"). This may be out of scope for this project, but is something to look into.

2. Another way to speed up the sequential algorithm is to accept some suboptimal answer within a factor of $w$. If we allow some wiggle room in minimizing cost, we can modify the cost function to become $f(n) = c(n) + w \cdot h(n)$ where traditionally $w = 1$ and $c$ is the cost and $h$ is the heuristic of a node $n$. If we increase $w$ (to something like 1.01), we emphasize the heuristic and can lead to a decrease of iterations (Jordan T. Thayer, "Bounded Suboptimal Search: A Direct Approach Using Inadmissible Estimates").

# 7 Parallelization Strategy

There are a few potential opportunities for speeding up the sequential implementation described before:

1. The A* algorithm's runtime can be reduced through parallelization:
   (a) On each iteration of A*, a node is selected and its neighbors are added to a priority queue based on its cost. We can explore the cheapest nodes in parallel on the next steps.
      i. Since there may be only a single "cheapest" node, we can relax the strictness for a optimal solution by using a bounded suboptimal search. We can explore all nodes in parallel that are within some margin of error of the "cheapest" node in the priority queue. These are nodes that are within a factor of $w$ of the cheapest node: $\{n\}\ \forall\ n \in\ f(n) \leq w \cdot f(cheapest_{node})$. This will find a solution with a suboptimal cost within a factor of $w$ (Jordan T. Thayer, "Bounded Suboptimal Search: A Direct Approach Using Inadmissible Estimates").
   (b) Calculations could *potentially* be parallelized, if there are any that are complex enough. Although, the manhattan distance calculation is simple.
   (c) If birdirectional search is implemented, it could be parallelized as well.

2. The SSP most likely cannot be parallelized much, but we could investigate parallelizing some of the updates at the end of each iteration

3. Graph parsing could be parallelized
   (a) Node splitting, which is required for finding node-disjoint paths, could be parallelized

For most of these parallelization methods, we will be utilizing mainly the Par monad. For smaller mathematical computations, we may use the par strategy. Also, we may need to use locks or mutexes (MVar or TVar) to lock shared resources (e.g. the priority queue), as raised in Samya Ahsan, *Parallel Functional Programming Proposal: MazeSolver*.

# 8 Evaluation

We will be evaluating our algorithm's implementation based on:

1. Correctness: The solution must provide the correct number of paths, $k$, that also minimize the amount of wire required. The paths must begin/end at the project start/end nodes.

   (a) In some cases, we may sacrifice optimality for speed by using some approximation methods. If this is pursued, we should require the solution to be bounded by some $w$ factor.

2. Performance: The parallel solution will be compared to the sequential solution's runtime.

In some cases, we may sacrifice correctness with approximation in favor of enhancing performance.

# 9 Concerns

The following are potential concerns with this project:

1. The SSP algorithm is more complicated than the A* algorithm alone. To mitigate risk, we can begin with the A* algorithm to find a single shortest path between any pins. Once that is completed, we can move to implementing the SSP algorithm. Solving with just A* would be disappointing, but still relates to the problem. Worst case, we can use A* on a loop, which is greedy and not optimal.

2. Choosing a sufficiently difficult graph to solve and consistently using the same graph to compare benchmarks.

3. A* can be difficult to parallelize due to the heuristic and priority queue. Dijkstra, while less applicable, could be used and potentially lend itself better for parallelization. **However, we would still prefer to pursue A*.**

Also, as raised in *Parallel Functional Programming Proposal: MazeSolver*, we may have to pay attention to the following while parallelizing the solution:

1. An efficient way to store nodes and the graph representation. In our problem, edges will additionally store capacity and flow.

2. Reduce conflicts of accessing the priority queue (PQ) that will be used in the A* algorithm. We can explore utilizing either a global PQ or giving each thread a local PQ that will need to somehow be merged after the thread's work is complete.

# 10 Summary

To summarize the salient points in this proposal:

1. We are aiming to solve for some of the complexities of routing on ICs and other circuitry

2. To do so, we are using the idea of Maze Routing to find a path that avoids obstacles between two nodes

3. Maze Routing can be solved using the A* algorithm with a Manhattan distance heuristic

4. To find $k$ node-disjointed paths with minimum cost, we will be using the Sequential Shortest Path algorithm in-conjunction with A*

5. We will assume that the IC board or the graph is a 2D grid plane. There aren't any diagonal movements between nodes. Where obstacles allow, there is movement only allowed along the horizontal or vertical direction. A generator and parser will help create example grids.

6. We have multiple parts of the algorithm that can potentially be parallelized: A*, SSP, graph parsing, or node splitting. The Par monad and MVars or TVars will be leveraged.

7. Suboptimal solutions, within a bound/factor of $w$, exist that can enhance performance in the sequential and parallel versions.

   (a) One suboptimal solution places greater emphasis on the heuristic than cost to reach a node, speeding up the sequential version by reducing the number of expansions.

   (b) The other solution allows us to explore paths in parallel whose costs are within a $w$ factor of the cheapest path currently known, rather than just exploring a few/single best path(s).

8. The A* algorithm could be used in a bidirectional search, which can be investigated and could be parallelized.

# 11 References


A. Kahng K. Keutzer, A. R. Newton. *Routing in Integrated Circuits*. URL: https://people.eecs.berkeley.edu/~keutzer/classes/244fa2004/pdf/4-routing.pdf.

Alok Aggarwal Jon Kleinbergt, David P. Williamson. "Node-Disjoint Paths on the Mesh and a New Trade-Off in VLSI Layout". In: (1996), pp. 585–594. URL: https://dl.acm.org/doi/pdf/10.1145/237814.238007.

Cendes, Zoltan. *Electromagnetic and Circuit Co-Simulation and the Future of IC and Package Design*. URL: https://ewh.ieee.org/soc/cpmt/tc12/fdip06/Cendes_Part_I.pdf.

Jordan T. Thayer, Wheeler Ruml. "Bounded Suboptimal Search: A Direct Approach Using Inadmissible Estimates". In: *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence* (). URL: https://www.ijcai.org/Proceedings/11/Papers/119.pdf.

*Manual or interactive routing, advantages and inconveniences*. URL: https://www.proto-electronics.com/blog/manual-or-interactive-routing.

Michael N. Riceand, Vassilis J. Tsotras. "Bidirectional A Search with Additive Approximation Bounds". In: *Proceedings of the Fifth Annual Symposium on Combinatorial Search* (). URL: https://ojs.aaai.org/index.php/SOCS/article/view/18235/18026.

Point, Tutorials. *Graph Theory - Infinite Graphs*. URL: https://www.tutorialspoint.com/graph_theory/graph_theory_infinite_graphs.htm.

Ravindra K. Ahuja Thomas L. Maganti, James B. Orlin. *Network Flows*. 1993, pp. 320–324.

Samya Ahsan Nicole Lin, Alice Wang. *Parallel Functional Programming Proposal: MazeSolver*. 2023. URL: https://www.cs.columbia.edu/~sedwards/classes/2023/4995-fall/proposals/MazeSolver.pdf.

Wikipedia. "Maximum flow problem". In: (). URL: https://en.wikipedia.org/wiki/Maximum_flow_problem#Maximum_flow_with_vertex_capacities.