AOC 2025 Challenge 8

OVERVIEW

This document goes into a quick summary of the background of the problem and then into more of a deep dive on the SW solution as well as the HW solution and the corresponding testbench created to verify the HW solution. This challenge focuses more on the HW solution rather than the SW so there is a lot more detail provided in the HW related sections.

BACKGROUND

For AOC 2025 challenge 8 part 1 of the puzzle was to optimally connect points in a point cloud together to reduce overall connection lengths between various points. For example, given 1000 points in a point cloud what are the 1000 smallest connections out of all possible connections and given those connections, what are the sizes (i.e. number of points) of each of the unique networks created from these 1000 connections.

SW SOLUTION

I first solved this problem in SW to analyze what was the algorithm required to solve the problem to see if it was something that would be interesting and realistic to solve in HW. For the SW solution I read in each point from the list of points and generated all possible connections for that point with any existing points. For each connection generated I calculated the approximate distance and then ran an insertion sort algorithm on the distance to see if it was smaller than any of the existing current 1000 connections. If it was smaller the list was updated and this process was repeated for all new connections and then repeated again for all new incoming points. Once all connections were created and sorted I took that final sorted list of connections and created networks of connections that shared points with each other.

Key Takeaways

There were several optimizations that I identified while coding up this problem:

- The rules of the puzzle stated that only the top 1000 connections mattered and duplicate connections within those 1000 were ignored. So for sorting connections all that mattered was distance between points, not what network they were part of.
- The size of the network was based on points, not number of connections, so any connections between points already in a network were not connected. This meant that the network itself had no cyclical references to each other when navigating through the network.
- With each new point the number of calculations grows linearly. Point 2 had only 1 connection but by point 1000 there were 999 distance calculations to be made which ultimately causes the algorithm to slow down with every new point added to the point cloud.
- The real euclidean distance was not necessary so I avoided doing any square roots when calculating the distance, this ends up requiring a bit more memory to store the larger calculations but simplifies the amount and complexity of the compute required.

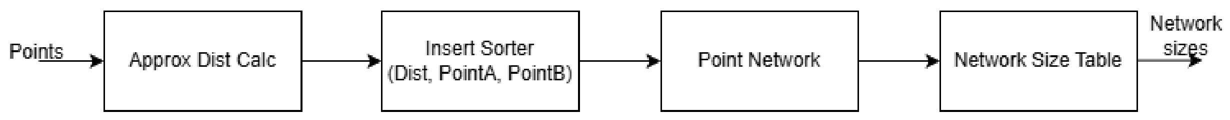
HW SOLUTION

For the HW solution for this challenge I followed the same flow that I did in SW to solve the problem but tried to evaluate different potential implementations of the HW to try to take advantage of the parallelization possible in FPGAs. In my past experience the kind of data being generated and how to process it was not something I typically would do in HW so I had to really evaluate where it made sense to parallelize and where it made sense to serialize.

My goal for performance for this design was to compute the network sizes as quickly as possible while still keeping the HW size reasonable and something that I could synthesize on my personal machine. I also wanted to achieve an overall clock rate of around 200-300MHz for a -3 Kintex US+ part.

Top Level Design

Below are the 4 main components of my HW module:



The block that has to process the most data is the approximate distance calculator (dist_calc) block and the insert sorter (ins_sorter) block. Given 1k total points every point has 999 possible connections but direction does not matter so only half of those are unique connections which gives you a total of **499500 distance calculations** that need to be created, sorted, and potentially added to a network.

To ensure the Point Network (point_ntwrk) block would not backpressure the ins_sorter and dist_calc blocks I had it stall until the final 1k connections had been sorted before creating any networks. See performance analysis section on more detailed reasoning for that decision.

Approximate Distance Calculator

The dist_calc block takes in 3 vectors that correspond to a point's x, y, and z location. It stores each location vector into a separate parallel table and backpressures any new points from being written until all calculations have been done with the current points. As each new point comes in that means the distance calculator has to do one additional distance calculation so the backpressure grows by one cycle with every new point.

So for example:

- First point no calculations are needed
- Second point one calculation [(P1, P0)] needs to be done
- Third point two calculations [(P2, P0) (P2, P1)] needs to be done
- etc.

To optimize storage for this the block stores the incoming point into a register and memory and then sweeps through that memory for all existing points to calculate all of the new distances. The formula I used for this distance is an approximate euclidean distance formula which is just the square of the normal distance formula.

$$d^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2$$

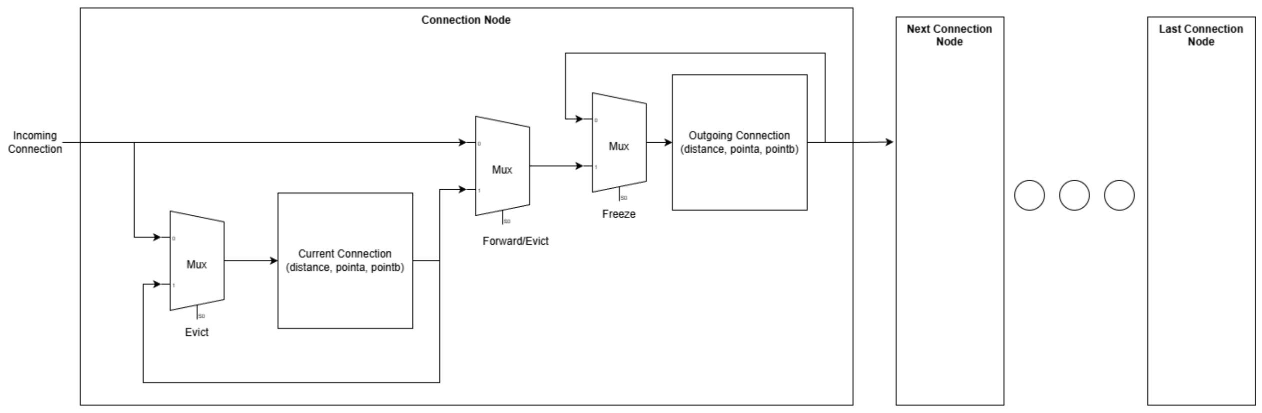
I chose this formula instead of the real euclidean distance as it saved me the trouble of having to implement a sqrt function in HW and is functionally equivalent to comparing distances vs the actual real distances. The tradeoff here though is that given a dimension size of 17bits I have to store 38bits of data for the distance info instead of 19bits of information. In the synthesis results section below you'll see this has a fairly big penalty in terms of FFs used so it would be worth the extra effort to implement a sqrt function here to save on a ton of FFs in other blocks.

After the last point is read in, a done signal is fired and sent to the ins_sorter block to indicate that it can enable sending the final sorted connections to the point_ntwrk block.

Insert Sorter

For the ins_sorter block I wanted to ensure that it would not backpressure the dist_calc block as it already takes ~500k cycles just to compute all calculations so this limited options for how to create a sorted list. It rules out any design that would use linked lists stored inside of a memory as you would not be able to instantly know where in memory to update the table as you would have to traverse the linked list to find the right place to update.

I decided on a shift register implementation that has additional storage per node to keep track of connection information and current distance. Shown below is a simple diagram of the main registers and decision logic for the sorter.



The ins_sorter has two phases, a sorting phase where it takes in a new connection at each clock cycle and a forward phase where it forwards the sorted connections out to the point_ntwrk block.

During the sorting phase the ins_sorter can not back pressure so it takes in a new connection every cycle. At each node it will check if its distance is smaller than the distance stored at this node and evict that node if it is. If the current node is evicted then it sends that evicted node to the next node to have it sorted. If the incoming connection is greater than the distance of the current node it will send that connection to the next node instead. Either the incoming connection or evicted connection will keep going down this shift register until it finds the appropriate node where it will be inserted. At the very last connection node the outgoing connection will be dropped as its distance was greater than any of the other existing nodes in the list.

The second phase is its forward mode which is enabled once dist_calc has sent its final distance and that has made it through the entire list of sorted nodes. During this phase the ins_sorter must be able to be stalled as the point_ntwrk block can not always insert a new point in a single cycle. To handle this case there is additional freeze logic in the sort nodes that is enabled to handle this backpressure. During this phase the node either forwards the incoming connection to its output port or it stores the incoming connection into the current node to save until the point_ntwrk is ready for the next point.

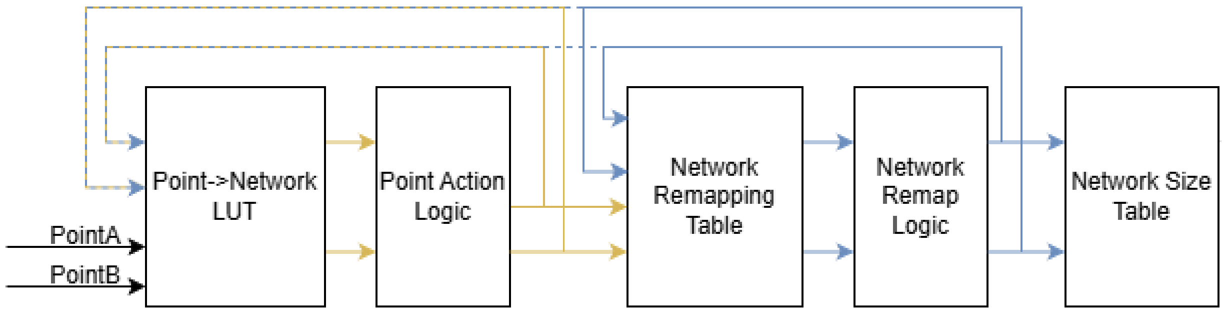
By designing the ins_sorter block this way I was able to avoid the ins_sorter block causing any backpressure to the calc_dist block but also able to stall the block when the point_ntwrk block had to cause back pressure to insert points into a network.

Point Network/Network Size Table

When all connections are sorted we start feeding the point pairs from the connection to the network block. At this point we no longer care about the actual distance between two points, only the connectivity, and so we ignore the distance data for the connections. For the final size calculation all we care about is how many points are inside of the same network. Note that the relationship between a network to a point can be one to many but a **point will always point to a single network**. When point pairs are fed into this block there are several actions that can occur based on the points coming in:

1. **NEW**: Neither point belongs to a network so create a new network consisting of those two points
2. **LOOKUP**: One or more points belong to a network, trigger an additional lookup to see if this network id is remapped
3. **IGNORE**: Both points already exist in the same network

The diagram below shows the structure of this block and how it handles creating networks as well as updating networks.

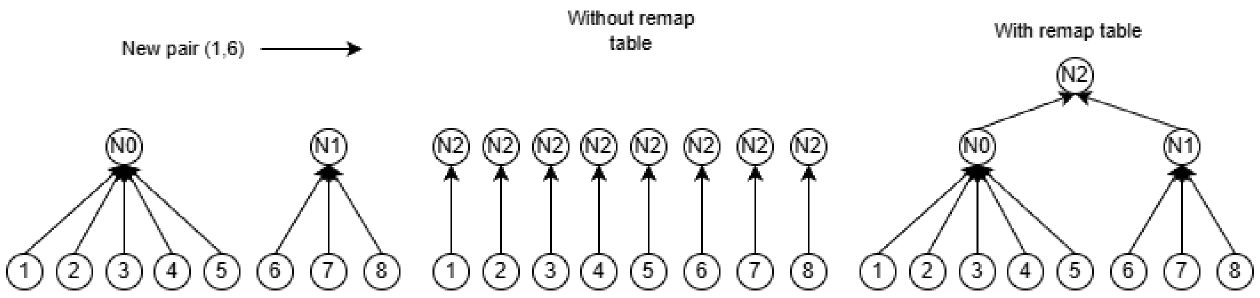


We take advantage of the 1:1 mapping between points and networks and use the point itself as the address to look up the network as there will always be just one network mapped to a given point.

The Network LUT takes in two points simultaneously and outputs two network IDs for the point action logic to decide on what is the next step. We use a network ID of 0 to indicate when a point has no mapped network to it. In the case that one or more points have a mapped network ID we do a secondary lookup to see if this network has been remapped. After the network remapping table is searched to find the final actual network id for each point there are a few possible actions:

1. **WR_A**: Add point A to the network of point B because A is not in any network yet.
2. **WR_B**: Add point B to the network of point A because B is not in any network yet.
3. **MERGE**: Both points belong to separate networks which means we need to merge those two networks together and remap the networks to a new ID.
4. **IGNORE**: Both points already exist to the same remapped network

One question you may have here is why we use a network remapping table for another level of indirection rather than just sticking to only using the network LUT. The main reason here is to reduce the amount of memory accesses needed for network merging. Below is a simple picture showing the difference of how network merging would happen with and without a network remapping table:



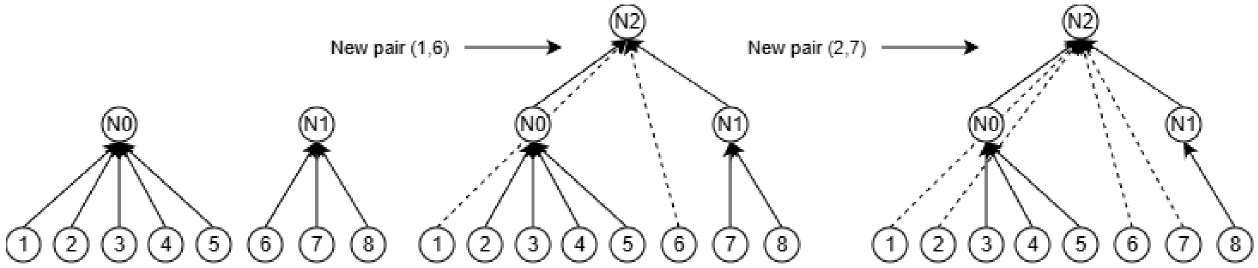
In the case where you do not have a remap table you must then read through the entire network LUT and check for every point if it needs to be updated to the new network ID. Assuming merging is happening frequently while reading in points this can slow down the network creation considerably.

In the case where you have a network remap table you simply take the final network ID for each point and update the network remap table to point to this new network for each old ID. The network remap table essentially becomes sets of linked lists that are traversed to find the final network ID for a given point.

Because the network remap table becomes a linked list it can sometimes require multiple reads to the remap table to find the final network ID. There are two additional optimizations here to reduce the lookup time for the remap table:

1. During a merge event, write back to the original network LUT the new network ID created for the two points while simultaneously updating the network remap table with this new network ID.
2. During a lookup if both points ended up at the same final remapped network ID then update the original network LUT to point to this final network ID to reduce the number of reads required the next time this point ID comes in.

Here's a diagram showing the two optimizations visually:



As points are read in the network size table is updated with the corresponding sizes. During a NEW event the network size table is initialized with 2 for that ID, during a WR_A or WR_B event the network size table is incremented for the respective ID, and during a MERGE event the network size table reads the two sizes from the current networks and writes the sum to the new network ID. The table also will keep track of which network IDs are valid/invalid as it must invalidate the old network IDs during a MERGE event.

Finally when all 1k points are read in and all network actions have finished then the network size table is read out and the three largest network sizes that are valid are recorded and sent out to the top level to generate the final product of all three sizes.

TESTBENCH

For the testbench for this project I decided to try out cocotb as I had heard good things from a former coworker and the idea of being able to use python to write my testbench rather than trying to setup an entire UVM test infrastructure was very appealing.

The testbench itself is setup to test the entire module together with various checkers on inputs and outputs as well as some internal signals for each block to verify its functioning properly.

There is one single test case in tb.py and it will first generate the golden files needed by the checkers and then start coroutines for each of the checkers (if enabled). After all of the checkers have finished their coroutines then it will wait for answer valid to go high and do the final check to make sure the answer matches what was generated from SW. The generated golden files can be found under misc/test_stimulus_00[1,2]/ depending on which test stimulus was used.

There are several classes in tb_classes.py I've created for checking HW:

PointOrchestrator

This class handles generating points to the HW. Whenever dist_calc block sets ready high, the block will then send it the next point from the golden file.

ConnChecker

This class handles both checking the incoming connections to the ins_sorter block match what is generated from SW and will also check the internal stored values of each sorter node matches the final sorted list. The structure for a connection has the following format:

(6074555086, 0, 1)

(distance, pointa_id, pointb_id)

NetworkChecker

This class checks both the incoming connections are in the correct sorted order as well as that each network action creates or updates the existing networks as expected. For checking network actions the checker function will update its own existing model of the networks with that action and check that this newly updated set of networks matches what SW has at that same point with the number of connections.

The structure of a network has the following format:

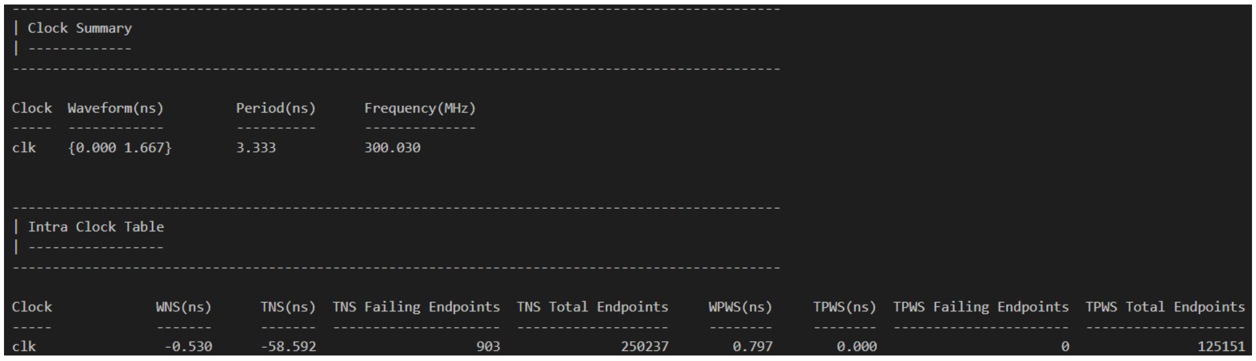
(network_id, valid=True|False, [point0_id, point1_id, ...])

PERFORMANCE AND AREA OPTIMIZATIONS

Besides coding up a solution to this problem in HW I also wanted to try and design something that I could actually synthesize and potentially run myself on an FPGA. The goal was to be able to synthesize and fit within a modest Kintex FPGA and have a target clock range in the neighborhood of 200MHz-300MHz. For the full size problem that contained 1k points and had to generate a network with 1k shortest connections see the following section for details.

Timing Analysis

Here's the clocking summary for the full design synthesis:



```
------------------------------------------------------------------------------------------------
| Clock Summary
| -------------
------------------------------------------------------------------------------------------------

Clock  Waveform(ns)       Period(ns)      Frequency(MHz)
-----  ------------       ----------      --------------
clk    {0.000 1.667}      3.333           300.030


------------------------------------------------------------------------------------------------
| Intra Clock Table
| -----------------
------------------------------------------------------------------------------------------------

Clock             WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WPWS(ns)     TPWS(ns)  TPWS Failing Endpoints  TPWS Total Endpoints
-----             -------      -------  ---------------------  -------------------      --------     --------  ----------------------  --------------------
clk                -0.530      -58.592                    903               250237         0.797        0.000                       0                125151
```

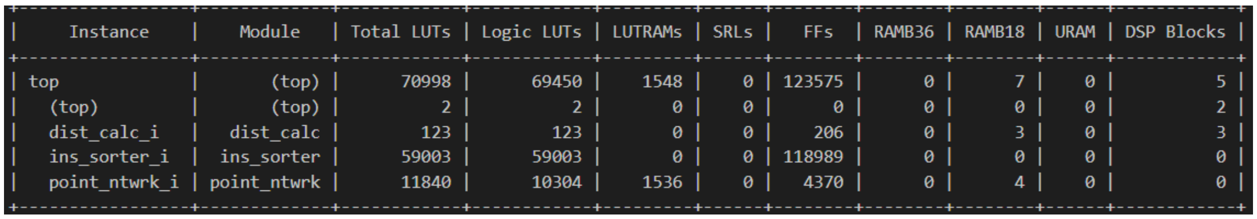
WNS was -0.530ns for a 300MHz clock so overall FCLK **~258MHz**

Overall my max clock rate is roughly in line with what I was trying to achieve with very limited work on fixing up timing paths. The only additional work I did to get to this WNS was some pipelining around the distance calculation blocks which is the most compute heavy logic in the design.

The worst case paths are mostly related to the multi dual port RAMs that I included in the point network block. I may actually be able to optimize those to be more traditional dual port rams as I do not necessarily need to do simultaneous 2 reads and 2 writes on a given clock cycle for the network LUT and the network remapping table. Additionally I should add some more pipelining in the point network block as I have logic stretching from both the network LUT to the network size table when a brand new network is created.

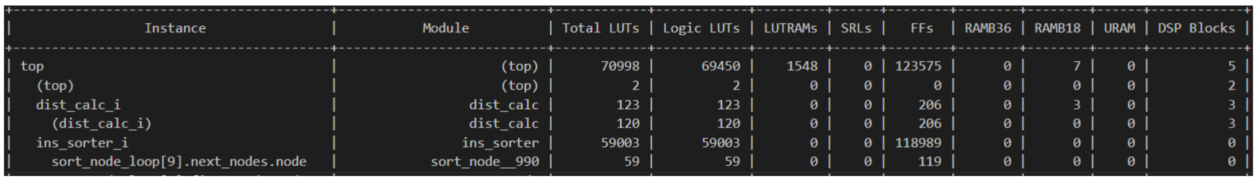
Utilization Analysis

Here's the top level resource utilization:



```
+-----------------+------------+------------+------------+---------+------+--------+--------+--------+------+------------+
|     Instance    |   Module   | Total LUTs | Logic LUTs | LUTRAMs | SRLs |   FFs  | RAMB36 | RAMB18 | URAM | DSP Blocks |
+-----------------+------------+------------+------------+---------+------+--------+--------+--------+------+------------+
| top             |      (top) |      70998 |      69450 |    1548 |    0 | 123575 |      0 |      7 |    0 |          5 |
|   (top)         |      (top) |          2 |          2 |       0 |    0 |      0 |      0 |      0 |    0 |          2 |
|   dist_calc_i   |  dist_calc |        123 |        123 |       0 |    0 |    206 |      0 |      3 |    0 |          3 |
|   ins_sorter_i  | ins_sorter |      59003 |      59003 |       0 |    0 | 118989 |      0 |      0 |    0 |          0 |
|   point_ntwrk_i | point_ntwrk|      11840 |      10304 |    1536 |    0 |   4370 |      0 |      4 |    0 |          0 |
+-----------------+------------+------------+------------+---------+------+--------+--------+--------+------+------------+
```

On the resource side the bulk of the design is attributed to the ins_sorter block. The main reason due to this is there are 1k instances of the sort node block. The sort node itself is not very large but because there are 1k instances of it, it quickly balloons in overall size for the module.

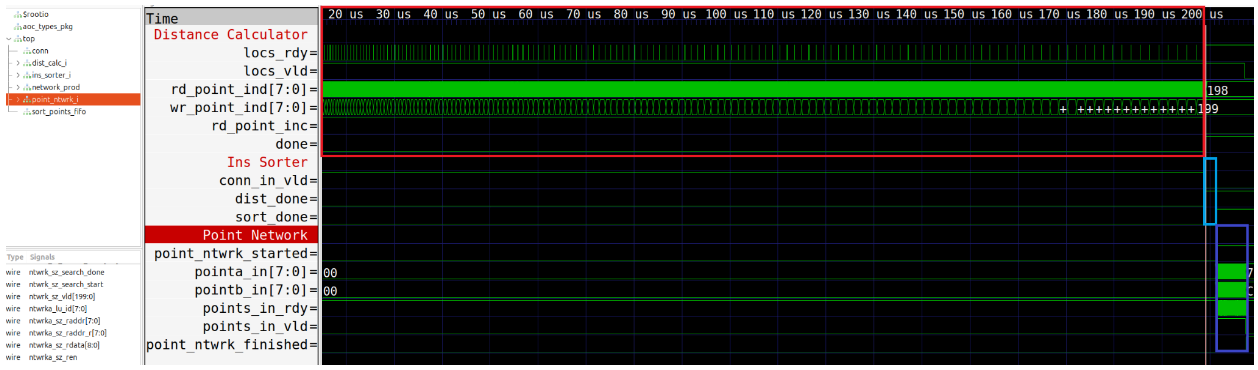


```
+-----------------------------------------+---------------------------+------------+------------+---------+------+--------+--------+--------+------+------------+
|                 Instance                |           Module          | Total LUTs | Logic LUTs | LUTRAMs | SRLs |   FFs  | RAMB36 | RAMB18 | URAM | DSP Blocks |
+-----------------------------------------+---------------------------+------------+------------+---------+------+--------+--------+--------+------+------------+
| top                                     |                     (top) |      70998 |      69450 |    1548 |    0 | 123575 |      0 |      7 |    0 |          5 |
|   (top)                                 |                     (top) |          2 |          2 |       0 |    0 |      0 |      0 |      0 |    0 |          2 |
|   dist_calc_i                           |                 dist_calc |        123 |        123 |       0 |    0 |    206 |      0 |      3 |    0 |          3 |
|     (dist_calc_i)                       |                 dist_calc |        120 |        120 |       0 |    0 |    206 |      0 |      0 |    0 |          3 |
|   ins_sorter_i                          |                ins_sorter |      59003 |      59003 |       0 |    0 | 118989 |      0 |      0 |    0 |          0 |
|     sort_node_loop[9].next_nodes.node   |             sort_node__990 |         59 |         59 |       0 |    0 |    119 |      0 |      0 |    0 |          0 |
```

One way to get this size down dramatically would be to implement a square root function on the distances generated as that would reduce the size of each distance from 38bits to 19bits cutting the number of FFs by about half as well as reducing the number of LUTs required for muxing.

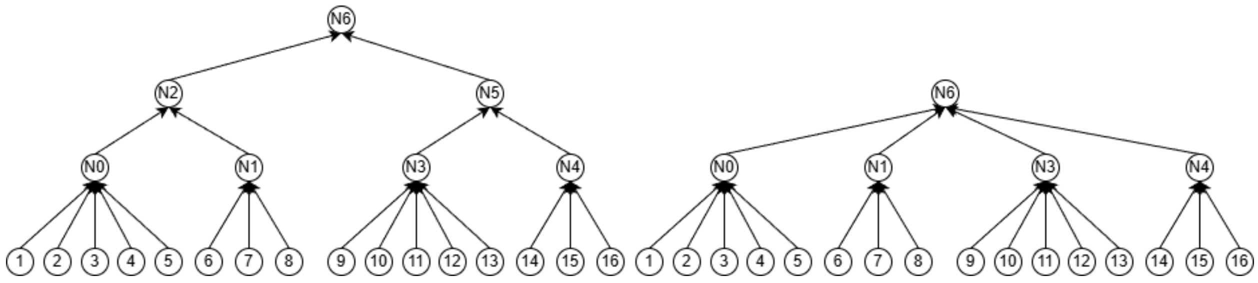
Performance Analysis

The overall performance of the design is heavily limited by the distance calculator block as it takes up the majority of the time for this module. As mentioned earlier the distance block has to calculate almost 500k distances when using 1k points so it will take the majority of the time to run since the ins_sorter block and point_ntwrk block only have to work on 1k points. Shown below is with only 200 points but the drastic difference between the three blocks is very apparent.



One way to save time here is to increase the number of distances calculated per cycle. If we double or even quadruple the number of distance calculators we can cut this time in ½ or ¼. The caveat here though is this will blow up the resource count for flops for the sorter block as it would essentially widen it to be able to handle multiple distances at a time. Additionally there would be many more LUTs per node used as well as you would have to do multiple comparisons every cycle at every node to decide whether to evict that node or not.

For the point network I already included some optimizations to reduce how many reads it takes to find the final network ID in the remapping table but there is an alternative approach which involves collapsing network IDs so that the lookup is always only 2 hops worse case. See diagram below for how that would look visually.



In the example above when N6 is generated you can reroute N0-N4 to point to N6 rather than updating N2 and N5. You can repeat this process anytime two hops are used to get to a final network ID and a merge action is generated. Doing this method ensures you only ever hop two times to find the final network ID. The downside to this is you can never update points to a new network ID as you may strand those points when collapsing network IDs so they must always point to the same starting network.

It would be interesting to try both methods and see what has better average lookup time for network IDs.

Finally one last note about having the point_ntwrk block wait until the dist_calc and sorter finished was to optimize overall run time. In a normal data pipeline I would not hold off downstream blocks from taking input data but the issue here is that if the point_ntwrk block were to take in every generated connection it would eventually backpressure the upstream blocks which is where most of the time is being spent. Additionally it would add more complexity to the point_ntwrk block as you would now have to invalidate points in a network as well which could ultimately result in deleting existing networks. If all of this logic can not be pipelined it would eventually backpressure the dist_calc block which would slow down the overall time it takes to get the final answer.