ECE1387 A3

Dec 4 2023

Troy Denton 1009600932

1. Description of program

The program models the decision tree as a binary tree. Each edge corresponds to assigning a given node's cell to a partition. More specifically, From a given node, the left child node is the partial solution where the node's corresponding cell is added to the left partition. The right child node represents the intermediate solution where the child's cell is added to the right partition.

Each node in the tree has a partition object. This causes a great deal of memory usage. Initially, I used many vectors/maps per partition object, but this caused a lot of memory usage. I optimized the memory footprint of the partition object by using bitfields to represent allocated nodes, allocated nets, etc.

Compilation, running:

```
mkdir build

cd build

cmake -DCMAKE_BUILD_TYPE=Release ..

make
```

run with ./a3 -h for options, info

basic use case:

```
./a3 -f ../data/cct1
```

1.1 Determining initial best solution

The initial best solution is an algorithm of my own design. The program also generates 10,000 random solutions, in case any of those are better (usually not). It is in partition.cpp as a3::partition::initial_solution_heur1. The algorithm works as follows:

1. Order cells by descending fanout in list
2. Assign highest fanout cell to the left partition, remove from list
3. Find a candidate right hand cell:
    1. for all cells in list, find the one with the highest score. The score is calculated as so:

        1. calculate the number of mutual nets for candidate (rcell) with respect to the initial left cell (lcell), n_mutual_nets
        2. find number unique ‘left nets’ and ‘right nets’.
            1. n_unique_l_nets = number of nets in left cell – n_mutual_nets
            2. n_unique_r_nets = number of nets in right cell – n_mutual nets
        3. score is calculated as (n_unique_l_nets)*(n_unique_r_nets)
        4. This score is designed to find an rcell that has a high number of connections, while also having a lower degree of connectivity to the existing lcell
        5. assign the highest scoring rcell to the right partition
4. alternate assigning to left and right partitions, finding the candidate cell like so:
    1. create a “supernode” of the current partition (e.g. a netlist of all assigned nodes to that partition)
    2. select the cell with the highest degree number of mutual nets with this supernode
    3. Assign the cell to the current partitions
    4. repeat the alternating action until the list of unallocated cells is empty

1.2 Bounding functions used

The function prune_basic_cost checks the size of the partial solution’s cutset against the known best solution, and it adds to this lower bound with the functions below. Efficacy was gauged by A/B testing changes on cct1 & cct2 and measuring runtime performance changes and visited node counts per level.

- a3::partition::num_guaranteed_cut_nets
    - The idea behind this function: maintain a nets that are not yet cut (uncut_nets) and cells that are not yet assigned (unassigned_cells). For each uncut net, count the number of cells connected to it that are not yet assigned to a partition. If the number of unassigned cells on this net is greater than half of the size of unassigned_cells, it will eventually be cut in the subtree.
- a3::partition::one_partition_full_cut_nets
    - Maintain a list of nets that are assigned to each partition (e.g. a net that has a node on the partition is “assigned” to that partition). If one of the partitions is at capacity based on balance constraints, the remaining unassigned_cells must be assigned to the opposite partition. For each net in the full side, if it connects to an unassigned_cell, it must be cut in the subtree.
- a3::partition::min_number_anchored_nets_cut

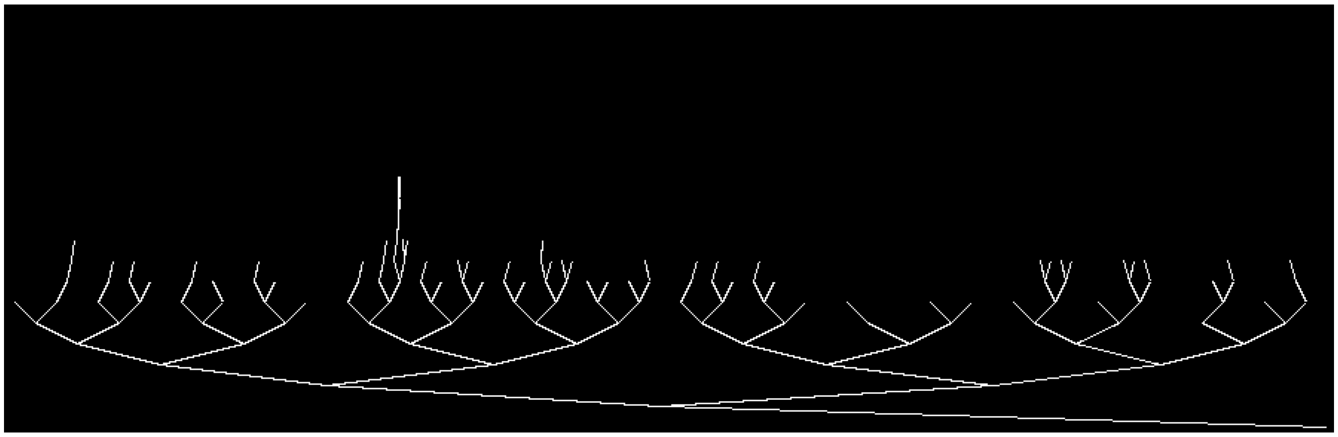
- If an unassigned cell has M nets assigned to the left partition and N nets assigned to the right partition, one of these sets will eventually be cut. We can't predict which, but we can bound it as the smaller of the two, min(|M|,|N|)
- We can only use the result from one cell, because we don't know how one cell's min(|M|,|N|) is going to combine with other cells' min(|M|,|N|) unless we explore the subtree.
    - The best we can do is track the largest min(|M|, |N|) and use that as a lower bound
- Note: this function cannot be used with one_partition_full_net_cuts, because we cant guarantee which nets are the ones that will be cut, just a count of them. Therefore we have no way to determine how the cutsets would combine (e.g. a total overlap would cause the nets to be double counted and it would not be a true lower bound)

2. Results

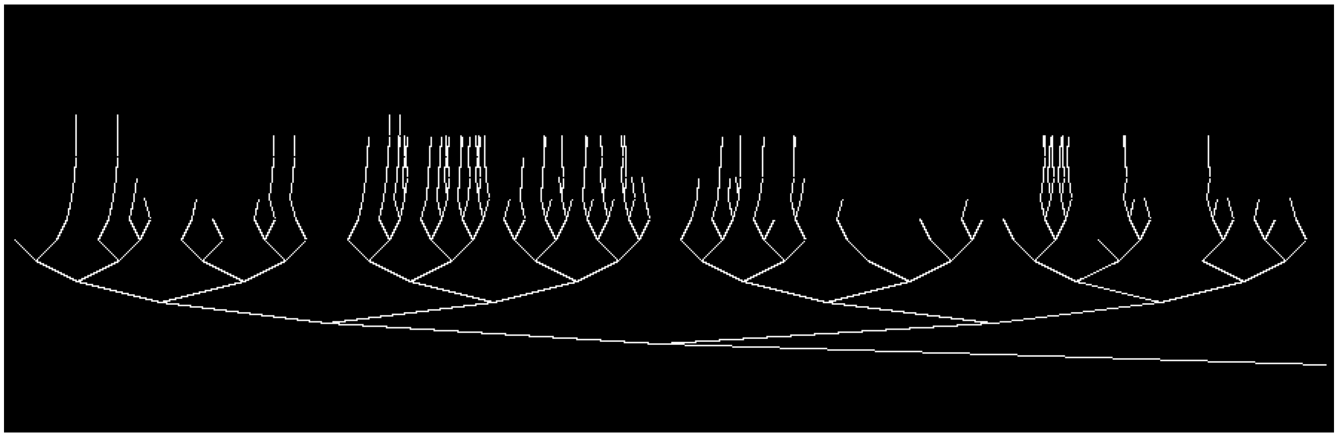
Most of these plots had to have specific source code modifications to get a reasonable screenshot, but you can try to see them in realtime by running with the interactive (-i) option. Note – cct3 BFS in interactive mode will consume an unreasonable amount of memory.

2.1 Plots from test circuits

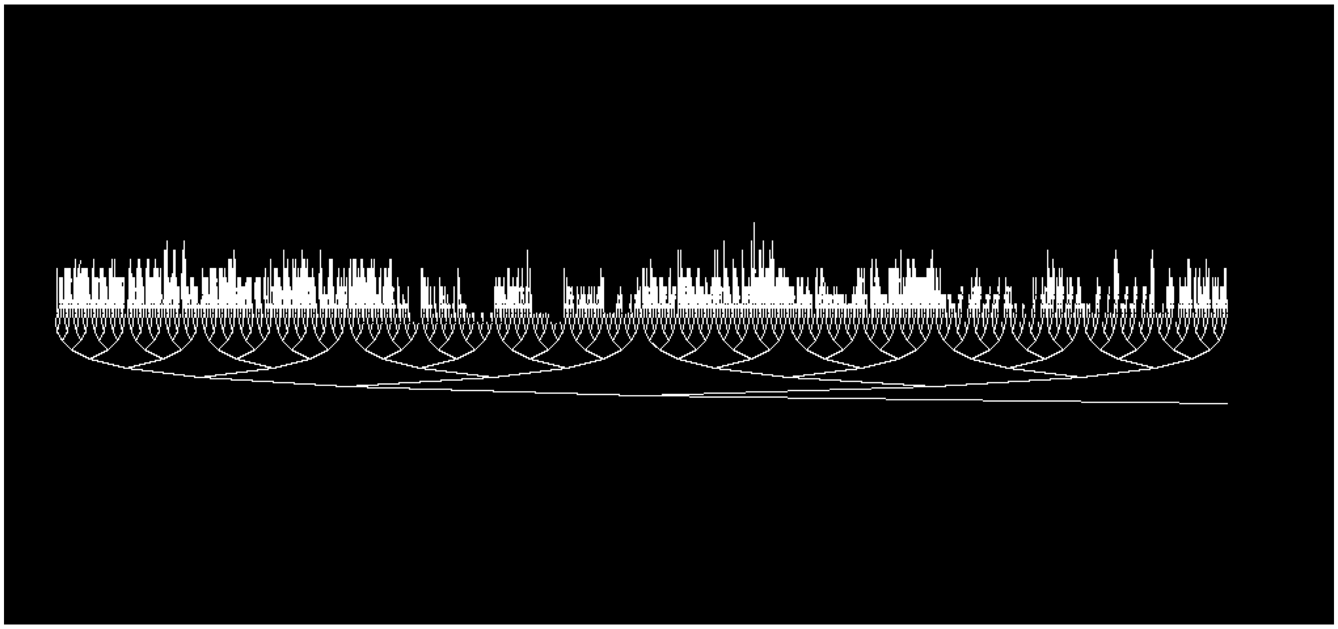
cct1 (LBF)



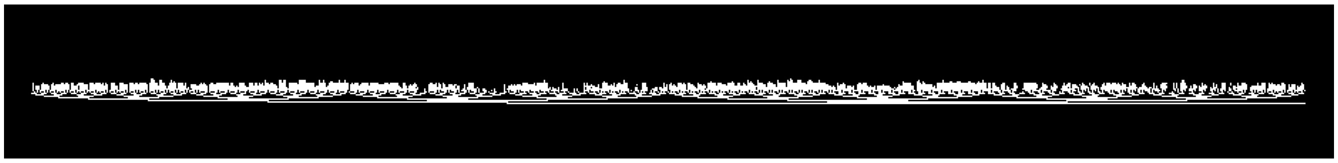
cct1 (BFS)



cct2 (LBF)



cct2 (BFS)



cct3 (LBF)

this and further diagrams omitted as they are difficult to generate, and appear as a generally useless thick white line due to their density

2.2 Numerical statistics from test circuits

These figures were obtained from my personal computer (Apple M2)

Circuit	**Count of nodes visited**		**Run-time**		**Net crossing count**	
	BFS	**LBF**	**BFS**	**LBF**	**BFS**	**LBF**
cct1	291	138	0.6 s	0.5s	28	28
cct2	12679	6587	1 s	1 s	42	42
cct3	89955915	1727790	31 minutes	51 s	74	74
~~cct4~~						

3. Results from 20% off-balance allowance

I could not get the program working properly with 20% off-balance allowance. I spent a very large amount of time getting BFS to run without getting killed by the OS, and could not get to this. I can speculate that runtimes and visited nodes get larger, and net crossing counts get smaller.

~~**Circuit**~~	~~**Count of nodes visited**~~		~~**Run-time**~~		~~**Net crossing count**~~	
	~~**BFS**~~	~~**LBF**~~	~~**BFS**~~	~~**LBF**~~	~~**BFS**~~	~~**LBF**~~
~~cct1~~						
~~cct2~~						
~~cct3~~						
~~cct4~~						

4. Discussion and summary

The most meaningful impact in terms of performance improvement was to perform the lowest-bound-first traversal – this has the benefit of finding good solutions faster than the breadth-first, which then allows for more pruning opportunities.

While I could not get the 20% off-balance solution working correctly, I expect this would increase the solution space by a large degree. It changes some of the heuristic logic to be less effective as well, impacting the amount of pruning that can be done. I would expect this relaxed constraint would result in some lower cut sets and a likely doubled runtime.

Other observations:

- My solution struggles with memory performance when performing the breadth-first search. I was able to run my solution on a pretty powerful machine at home to mitigate the effects of this. I was only able to get cct3 to run to completion after many, many hours of optimization and bug fixing.

- There are probably data structure improvements that can be made; I’m allocating a new partition object for every node in the decision tree, which really adds up.
- Given more time, I would focus my efforts the following areas:
    - better memory optimization
    - opportunities for threading enhancements