Dustin Kut Moy Cheung - 996881147

ECSE 1387 - Assignment 3 : Branch And Bound

1.

Circuit	Runtime / s	Decision tree nodes visited	Minimum crossing count
cct1	0.00	2,005	21
cct2	0.57	93,473	33
cct3	1.64	222,077	35
cct4	48.98	5,535,181	42

The plots are found in the end of this report.

2.

Circuit	Runtime / s	Decision tree nodes visited
cct1	0.00	2,080
cct2	0.31	195,210
cct3	0.64	328,894
cct4	29.62	9,783,324

3.

The program is organized as follows:

File	Purpose
Netlist.cpp	Parsing of the netlist file, and populating variables `BLOCK_NETS`, `NET_BLOCKS`, and `BLOCK_EDGES`
Node.cpp	Used to represent each node explored in the B&B algorithm. Each node has a pointer to its parent, and pointers to its nearest left and right ancestor nodes.
BranchAndBound.cpp	Implementation of different possible B&B methods (Depth-First search, parallel Depth-First search, Breadth first search)
main.cpp	Main program

Algorithms used, and investigated

Throughout the process of working on this assignment, several implementation of B&B were investigated, namely the Depth-First search (DFS) and Breadth-First search (BFS). In the end, the author chose to use DFS because the algorithm explores less nodes than BFS while being memory-efficient. Running BFS on the cct4 netlist easily consumed more than 10 GB of RAM memory.

Branching structure, levels of trees and ordering

Most of the logic of this B&B implementation are found in Node.cpp and BranchAndBound.cpp. For each node, we generate the left and right children of that node in '`Node.cpp::get_branch`' method. The block to use at each level has to be provided to that method.

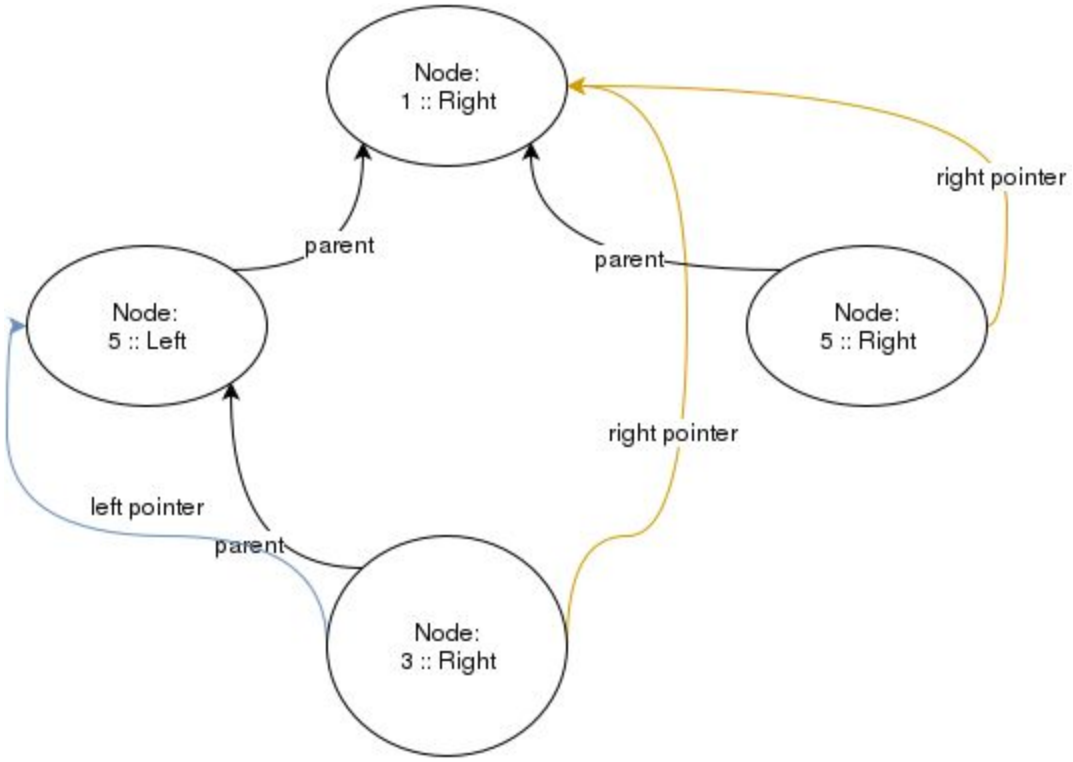


Diagram 1: Node representation

The diagram above illustrates how each node are connected to each other. For example, let's consider Node 1, with direction Right as the root node. A call to `get_branch` will generate the left and right nodes (*Node:5 :: Left* and *Node:5 :: Right*).

Given the root node, and the blocks to use at each level, we can generate the whole decision tree by just calling the `get_branch` method. This method is used a lot in our different implementations for B&B..

Each level of the tree represents a placement of a node/block in either the left or right direction.

The block to use at each level is determined inside '`main.cpp::get_order_evaluation`'. The blocks are ordered from highest number of edges to lowest number of edges. We'll pick the block with the highest number of edges as root and lowest number of edges as leaf. This is done to increase the amount of pruning occurring during B&B to reduce the number of nodes to explore in the decision tree.

Initial "best" solution

The initial "best" solution was determined using a simple heuristic. Each net is sorted in decreasing order according to how many blocks are connected to that net. Then we pick the first net (the first net in sorted manner will be the net with the highest number of blocks to it) and add each of its block to the left side, making sure we are not violating any balance conditions. We then pick the next net and repeat the process until we have the requirement number of left blocks. The rest of the blocks are added to the right blocks.

Circuit	cct1	cct2	cct3	cct4
Minimum crossing cut	21	33	35	42
Initial solution cut	25	40	47	56

This initial solution heuristic does a good job for small netlists, but start to get worse as the number of blocks increase.

Bounding function

The bounding function used is the number of crossing count for that node. This is done as such:

1. When a new node is created from its parent, the new node will copy the set of nets intersected from its parent
2. When the `calculate_lower_bound` is called on node *n*,, it will visit all its ancestors in the opposite direction of *n* and see if it intersects any nets from them. If it does, add the nets to the set of nets intersected.
3. The crossing count is the size of the set of nets intersected.

This *left* and *right* pointers (seen in Diagram 1) are used so that we don't have to visit all the ancestors of node *n* to know if they are in opposite direction. Instead, we can use the *left* and *right* pointers to just visit ancestors in the opposite direction of node *n*.

One reason why the bounding function was designed as such is because it doesn't involve any locking required while calculating the lower bound / crossing count since it doesn't modify any of its ancestors. This is useful in the multithreaded version of B&B.

To illustrate how effective the bounding function is, we ran the DFS with and without the bounding function for cct1 and cct2 netlists.

Circuit	cct1	cct2
Nodes visited with bounding function	2,005	93,473
Nodes visited without bounding function	35,749	29,715,999

Circuit	cct1	cct2
Runtime with bounding function / s	0.04	0.81
Runtime without bounding function / s	0.21	244.80

Note: the runtime was done using the author's computer, not the ECF server.

The bounding function significantly reduces the number of nodes to visit, and as a result, this dramatically improves runtime.

Traversing the tree and pruning

For each node popped from the stack in DFS, we generate the children for the next level based on the ordering of the level.

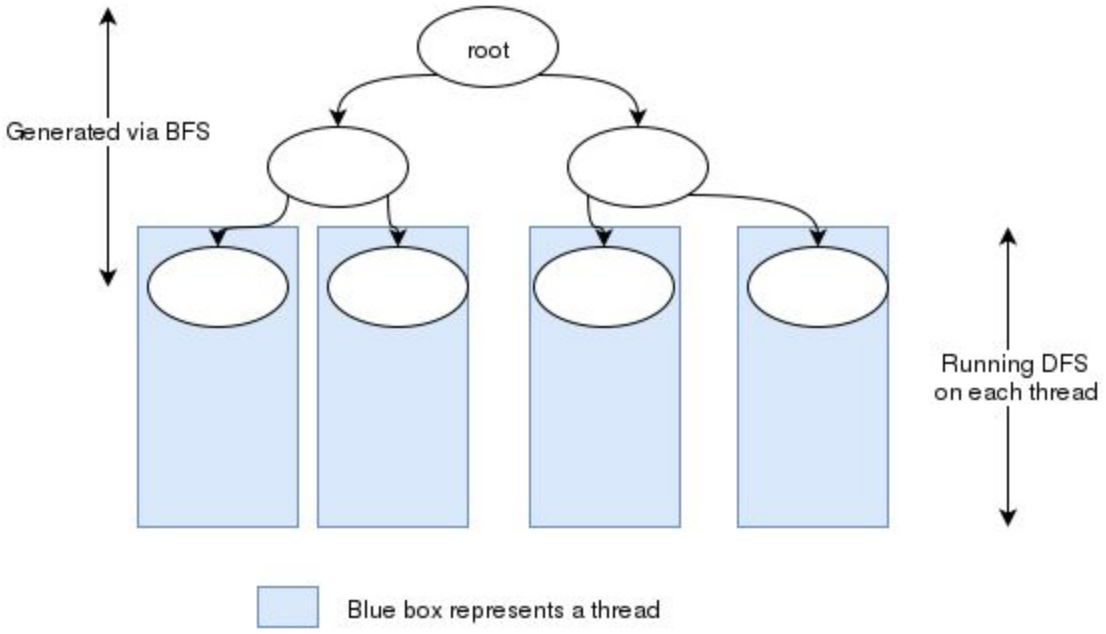
The children nodes are evaluated first to make sure they don't violate the balance conditions. After that their `calculate_lower_bound` function is called to see if the crossing count is bigger than the actual best crossing count. If yes, then the child is discarded, else the child is added to the stack.

Parallelization Approach

The initial approach for parallelizing B&B was to parallelize BFS (implemented using *OpenMP*). Since BFS operates using a queue, it was fairly easy to create thread workers that would pop an item in the queue, process it, and add the child nodes to explore back in the queue. This is implemented in '`BranchAndBound.cpp::parallel_breadth_first_search`'. This approach has a lot of advantages in that the thread workers are always doing work as their only tasks is to pick the next item in the queue and process it. However, the algorithm consumed a huge amount of memory and this approach had to be abandoned for a parallel version of DFS instead (implemented using *C++11* threads).

The parallel version of DFS works as follows.

1. Run BFS on the decision tree up to a certain level of the tree.
2. Given the queue of nodes to process for that level, create a thread for each item in the queue and run DFS on it.
3. Have global variables to track the best lower bound, best solution and number of nodes visited



This approach has the advantage of being vastly better memory wise compared to parallel BFS. However, we notice that some threads end sooner (because more pruning occurs in the region it is responsible) than others and therefore, this parallel version of DFS does not split the work evenly. This could perhaps be fixed in another iteration of the implementation by using work stealing.

The number of threads used depend on which level of BFS we stopped. Since the ECF server has 32 cores, we stop at level 5 of the decision tree, and create 32 threads to run DFS on the nodes in the queue. This is done so as to not create too much swapping among the threads, while exploiting all the cores in the ECF server.

In this parallel version, we have to make sure that shared memory is synchronized properly. Since each thread has its own copy of a stack, we do not have to synchronize on this.

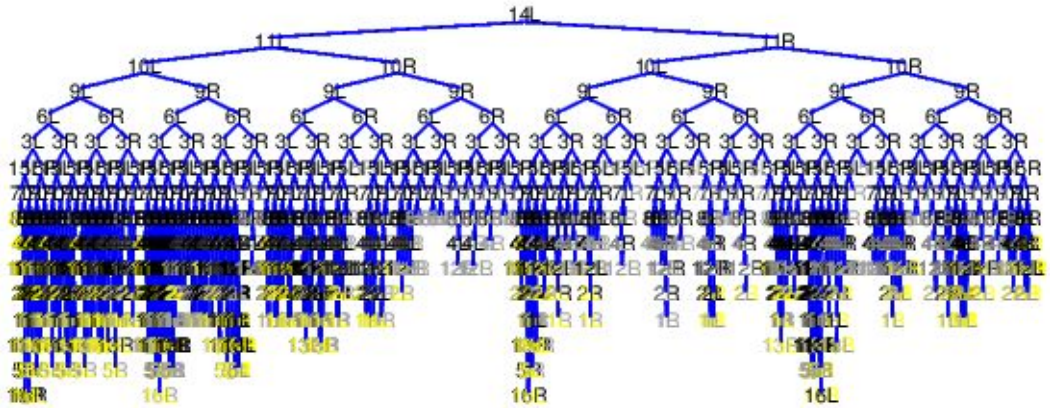
The `nodes_visited_parallel` variable (defined in BranchAndBound.cpp) is an atomic int variable. We use the *std::atomic* wrapper to make sure that only one thread at a time can increment this variable.

Similarly, we use *std::atomic* to update the best lower bound (`best_parallel` variable). Finally, to update the best solution variable, we apply a mutex to make sure only one thread can update the `best_parallel` variable and the best solution (`best_node_parallel` variable) at a time.

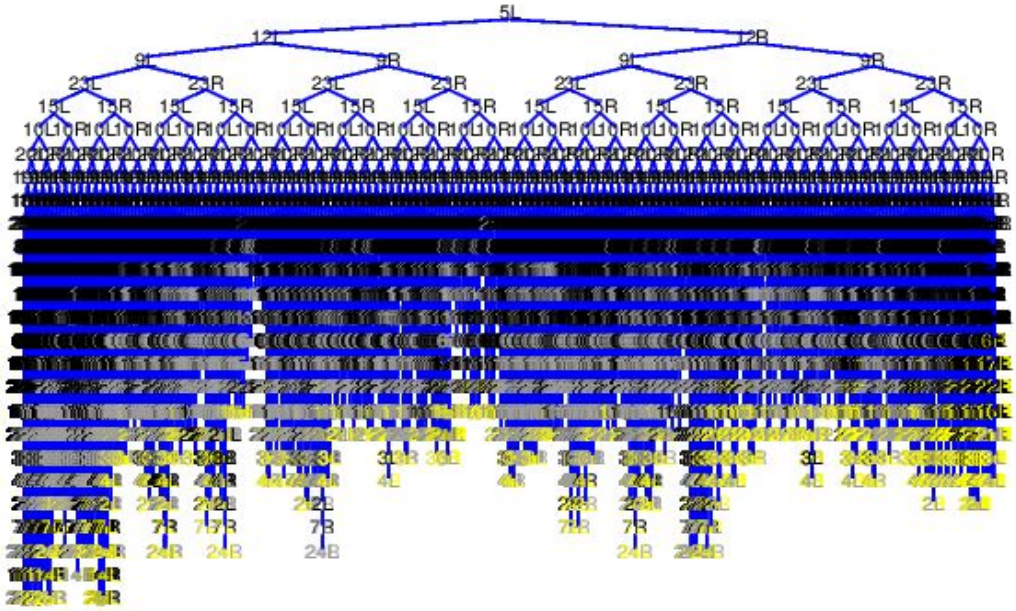
Plots for part 1

Note: dark grey nodes are pruned nodes, and yellow nodes are out-of-balance nodes.

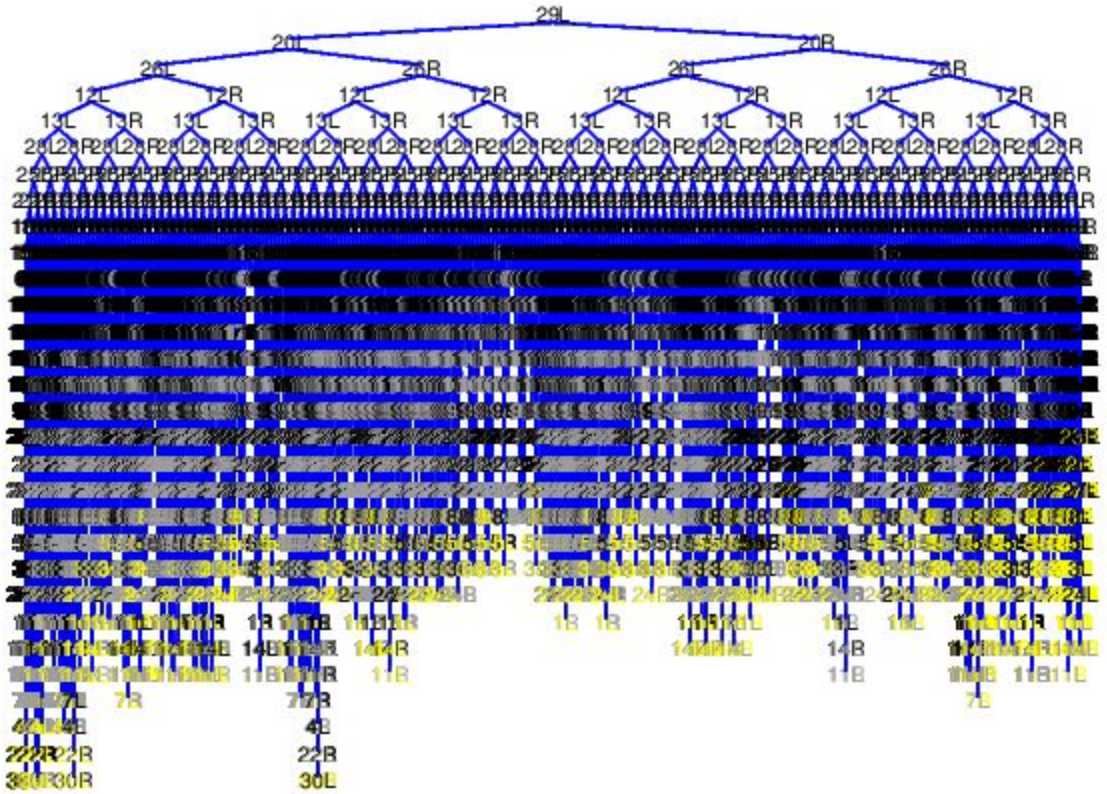
cct1:



cct2:



cct3:



cct4: