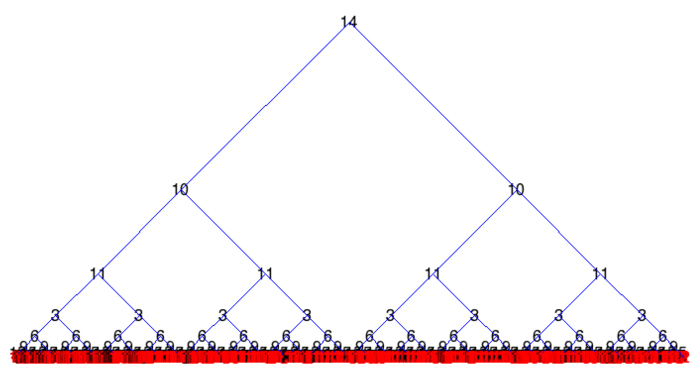
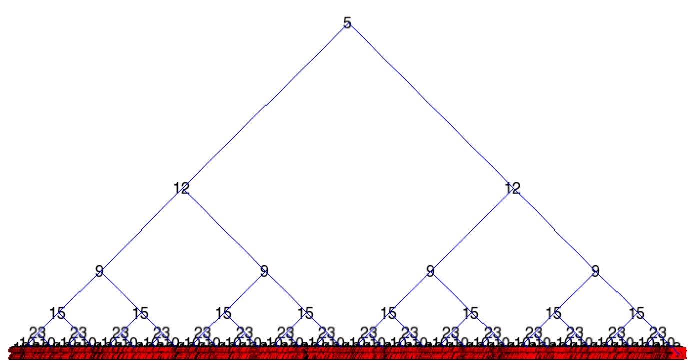
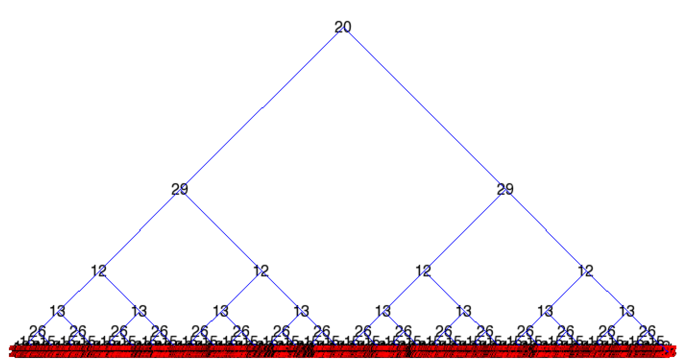
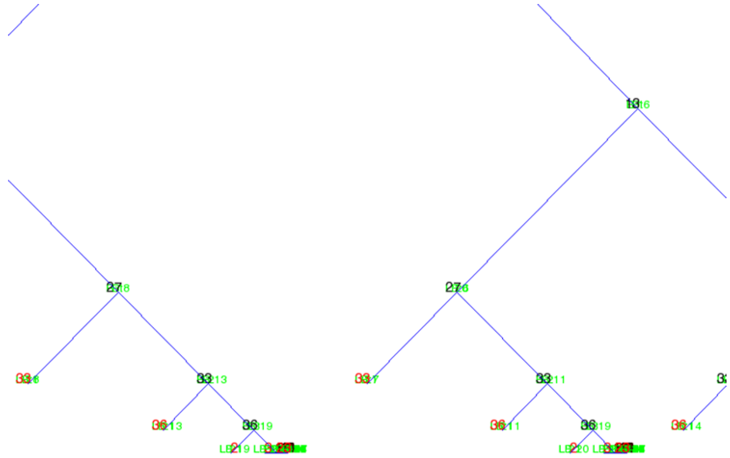
ECE 1387 – Assignment 3

Minghua Zhao
996789355
Due Date: November 13, 2015

Item 1

Circuit	*Min. Cross Count*	*Run Time*	*Total Visited Nodes*	*Plots* *(cct4 has the top redacted for a detailed view)*
cct1	21	0.3	5949	
cct2	33	14.25	179847	
cct3	34	82.98	741241	

cct4	42	1459.58	11399013



Item 2

Circuit	*Run Time**	*Total Visited Nodes*
cct1	0.370922	5948
cct2	17.4552	190416
cct3	190.49	1317738
cct4	2667.53	15443574

Item 3

The decision tree for this program is a binary tree, similar to the one shown in class.

- Each “tier” of the tree represents the position of a ‘block’ at a stage of the solution.
- Nodes have two children, corresponding to Left and Right side of the split.
- The “tiers” of blocks were ordered in descending fan-out.
    - The root node is the block belonging to most nets.

The initial best solution was partitioned as such:

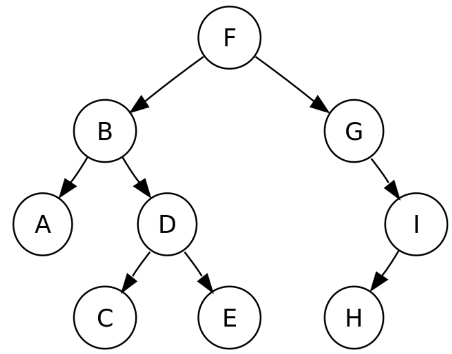
```
balance = total number of Blocks / 2
For each Net in Net_List
        For each Block in Net
                if Block number < balance
                        put Block in Right Side
                else
                        put Block in Left Side
        end For
end For

best solution = count connections between Right and Left sides
```

*Results implement Parallelization option #1 and #2

The Lower Bound was used. As described in class, it stores the number of cuts at a node in the tree. It is very effective in reducing exploration space – specially on the larger trees (more number of nodes).

It effectively prunes sub-trees that may grow at an exponential level. This saves a lot of processing time. The Lower Bound was used at the leaf-level to find a new best cut to the circuit.

The Tree traversal was Preorder. As seen on the picture to the right, the traversal order is: F, B, A, D, C, E, G, I, H.



The tree was recursively created on the first-pass, as opposed to building a whole balanced tree with every possible leaf, and then traversing through it for pruning. This decision was made early on to improve performance.

How is it Lower-Bound pruned?
The same way as described in class. I check the current node against its parents, and see if there are cuts introduced. If there are, I add the numbers to that node's parent's Lower-Bound.

If the Lower-Bound at the node is >= best_cost,
    the node is pruned.
Else, if the Lower-Bound, at the node == leaf && < best_cost,
    best_cost = node.LowerBound
Else
    continue to next nodes

Parallelization

C++11 std::async API for thread-creation, which uses POSIX threads on the back-end. A total number of 4 parallel threads is used in this solution. The number 4 was chosen because my local machine has an intel i7 processor with 4 cores, and it would be optimal if I utilized all cores evenly, ie. One per thread. The thread number – being a hardcoded variable can be easily adjusted to fit the machine.

All threads were reading and writing to the same best_cost variable. To avoid race conditions, mutex locks were used to stop one thread from "trampling" another mid-run. The specific case scenario that required the lock is when the node is a leaf, and has decided to overwrite the best_cose (it has found a better cost).

Run-Time

- Parallelization is breadth-first instead of depth-first
    - A best_cost cannot be found until one thread has reached a leaf node.
    - Since the threads are running in parallel, there is no pruning until almost all of the threads are at the leaf-level, as the symmetrical tree allows for about the same run-time for each thread.

- Due the parallel (breadth-first) traversal of the tree, more nodes are visited before pruning starts!
    - This is really bad
    - It costs a lot to run Lower Bound function and Balance function on node
        - Becomes costlier as tree deepens
    - When a node is visited, it costs time to process the balance values and the lower bound of the node. Specially in larger circuits.

Parallel Optimization #1

- But what if I could massage the algorithm to not have them all in “tier-sync”?
    - I introduced a delay between the starting of the threads.
    - The delay is directly proportional to the total number of Blocks to be processed.
    - This way, Thread#1 starts some milliseconds ahead of Thread#2, which starts ahead of Thread#3, which starts ahead of Thread#4.
    - This greatly improved the performance of the parallel version by reducing the total number of nodes visited.

Parallel Optimization #2 (only working on OSX)

- But what if I can give one thread a chance to truly finish?
    - Staggering Thread start-times might not be very accurate.
    - Ie. Thread#2 would still have processed a lot of nodes before Thread#1 found a best_cost
    - Ok, so the solution is to wait for the very first thread to finish, then start and stagger the other Threads.

With these two optimizations, Run-time for threaded achieved much better performance.