ECE1387 – CAD for Digital Circuit Synthesis and Layout

Assignment #3 Report

Branch and Bound Partitioning with Parallelization

Lawrence Park (Student # 995936395)

11-15-2015

Lawrence Park
Nov 15nd, 2015

1. Overview of Software Flow

The branch-and-bound (B&B) partitioning presented for this assignment performs a minimum-cut bisection of a given graph. The program assigns the vertices of a graph to either left set or right set, and the nets that connect vertices across the sets are added to the cut cost.

The B&B partitioning constructs a binary decision tree at which its nodes represent partial solutions to the bisection problem. As the program traverse down a level on the decision tree, a free vertex is assigned to a set, and the cut size of the partial solution is updated. Once all the vertices are assigned, a bisection of the given graph is completed, and its total cut size is compared to the current best cut size. The B&B partitioning ensures that the minimum-cut solution is found by considering the entire solution space of the bisection problem. Since the solution space grows exponentially with the size of graph (i.e. number of vertices), it is essential to detect a tree node that is determined to yield a sub-optimal solution, and prune it as early as possible. The following subsections in this report discusses some key aspects of the minimum-cut B&B partitioning that affects its performance metrics: program runtime and number of visited tree nodes.

1.1 Branching rule

To improve the performance of the B&B partitioning, we want to detect a partial solution that will not result in a better cut-size as early as possible in the decision tree. Therefore, vertices that are likely to lead to higher cut are favored, and should be branched early on. The past work on the graph partitioning problem suggest that the degree of a vertex (i.e. number of adjacent nets) is a good proxy for higher cut sizes. Therefore the vertices are sorted by the decreasing order of their degrees, and the decision tree branches on the sorted order. Furthermore, the initial solution is obtained by assigning the half of the sorted vertices to one set, and the remaining vertices to the other. This approach resulted in a reduced number of visited tree nodes compared to breath-first-search approaches.

Another important aspect of the B&B branching rule is the tree traversal order. The goal is to reach a complete solution with a “good” cut size early, so that the upper bound (i.e. current best solution) can be updated, because a good upper bound allows more sub-optimal solutions to be pruned. Therefore, the decision tree is traversed in the depth-first order. This is implemented via a recursive call to the B&B routine that performs a single binary tree expansion on a given node. Since there are two children nodes to every node (i.e. assign a vertex to either left or right), the program recurses on a child node with a better lower bound.

1.2 Lower Bound Computation

To process a decision node, we must compute a lower bound cut-size of the given partial solution; if the computed lower bound is greater than the current upper bound, the program can prune the decision node. Therefore, the quality of lower bound computation greatly affects the total number of visited nodes.

To improve the lower bound computation, the program performs a look-ahead on the free vertices of a given decision node. If a free vertex is adjacent to nets that are currently confined to the left or the right set, it is guaranteed to increase the cut size by *min(left-confined nets, right-confined nets).* These nets are added to the cut size of partial solution, and are recorded on an array to prevent duplicate counts in the lower bound computation. Through experiments, it was determined that the look-ahead technique significantly reduced the number of visited nodes for the given test circuits.

Another valid optimization technique is leveraging the balance constraint of the bisection problem. If a decision node has a set that cannot assign another vertex due to the balance constraint, the complete solution can be determined at the node because the remaining vertices must be assigned to the other set. This prevents unnecessary branching at its children nodes, thereby reducing the total number of visited nodes.

1.3 Parallelization

The C++11 concurrency library is chosen to parallelize the B&B partitioning; the C++11 standard provides a high-level abstraction to spawn an asynchronous thread that execute a given task in parallel. A task can be spawned to each recursive call to the B&B branching function that will traverse down on a given tree node independently to the other children nodes in a given tree level. Since each tree node keeps a distinct copy of its partial solution – current partition, free vertices, and cut-size, each thread can complete the B&B partitioning on its respective subtree. The number of spawned thread is configurable via a command line argument. The variable that must be shared across the threads is the current upper bound cut. Since the variable is of a primitive integer type, the mutual exclusion can be achieved by using the atomic datatype supported by the C++11 standard.

2. B&B Partitioning Results

The following tables summarize the B&B partitioning results on the 4 test circuits. The runtime is computed by averaging samples from 10 consecutive runs. The code is compiled with O2 optimization level. The total number of visited nodes are deterministic for single-threaded partitioning.

	Min. crossing count	**Runtime on ECF**	**Total # of visited nodes**
cct1	21	4.4 ms	1679
cct2	33	0.15 s	46570
cct3	35	0.33 s	140801
cct4	42	14.3 s	4646061

Table 1. *Single-threaded B&B partitioning results*

	Min. crossing count	Runtime on ECF	Total # of visited nodes
cct1	21	2.2 ms	1452
cct2	33	0.12 s	57740
cct3	35	0.28 s	185099
cct4	42	13.2 s	5631970

Table 2. *4-threaded B&B partitioning results*

It is worthwhile to note that the runtime deviation of samples were much higher for multi-threaded results, and the speed-ups gained from multi-threaded plateaued at threads = 4. I observed that the runtime speedup is much more consistent at a local machine; this is probably due to the shared workloads on the ECF machine. The non-linear speedup can be attributed to the increase in the number of visited cells, and the threading overheads.

The following figures are the plots of the test circuits.

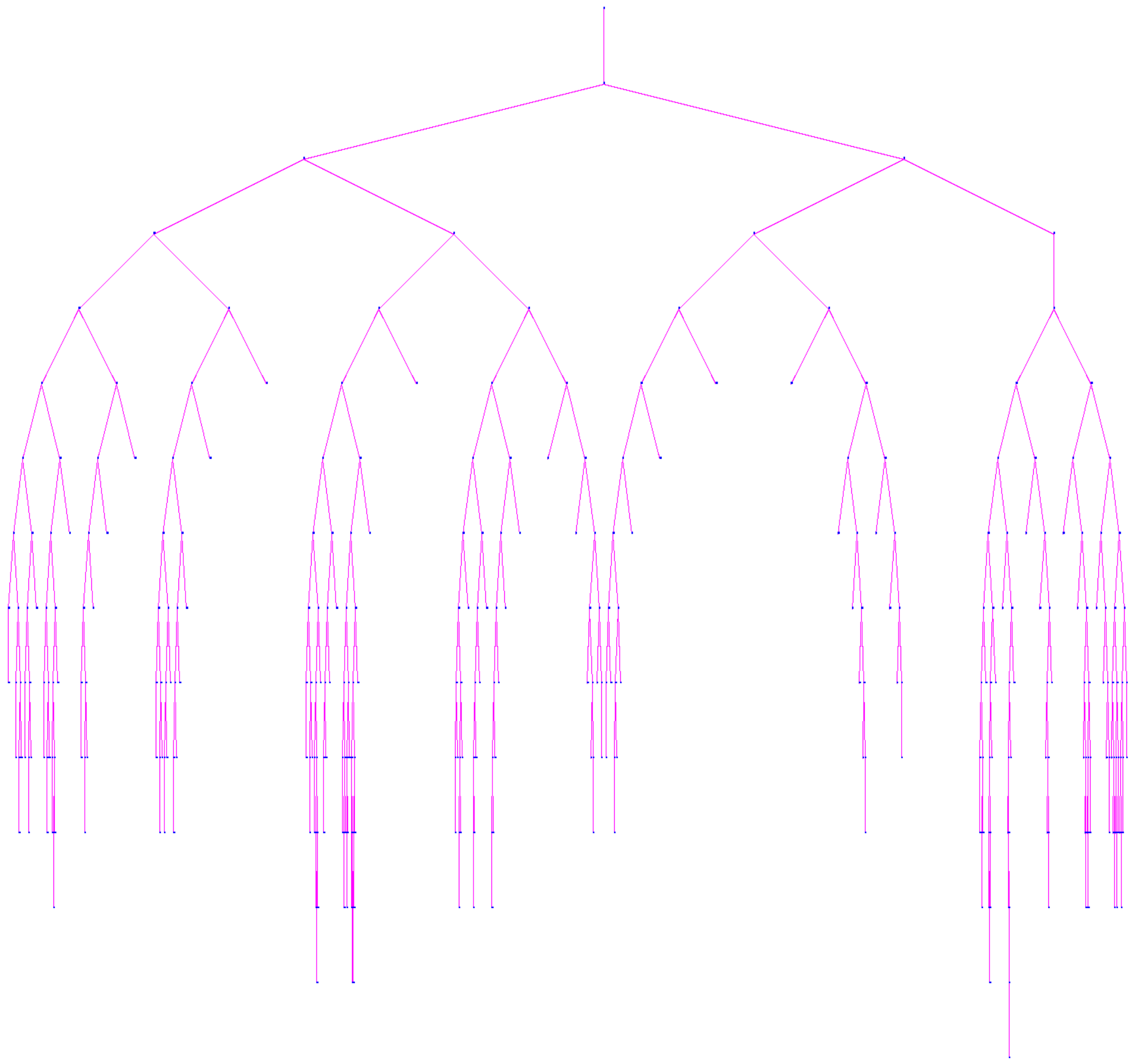

Figure 1. *cct1 decision tree plot*

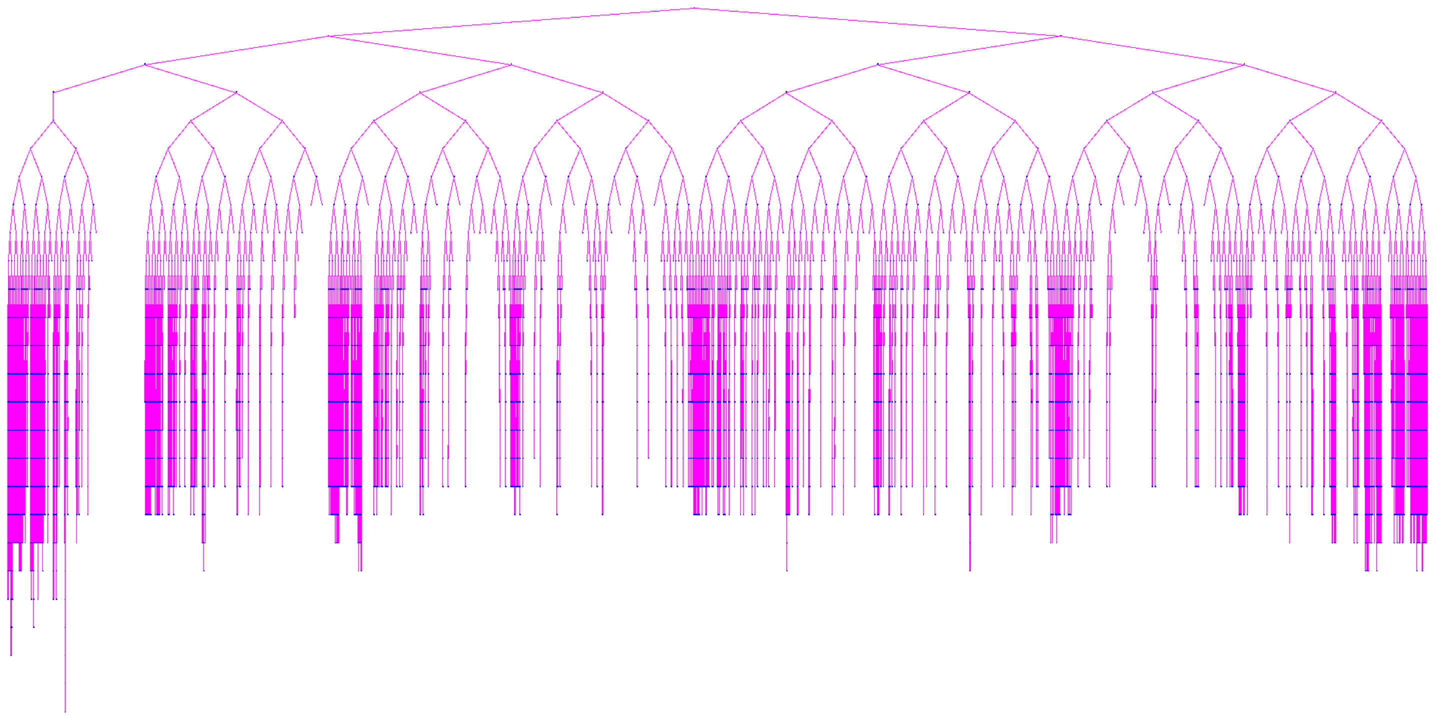

Figure 2. *cct2 decision tree plot*

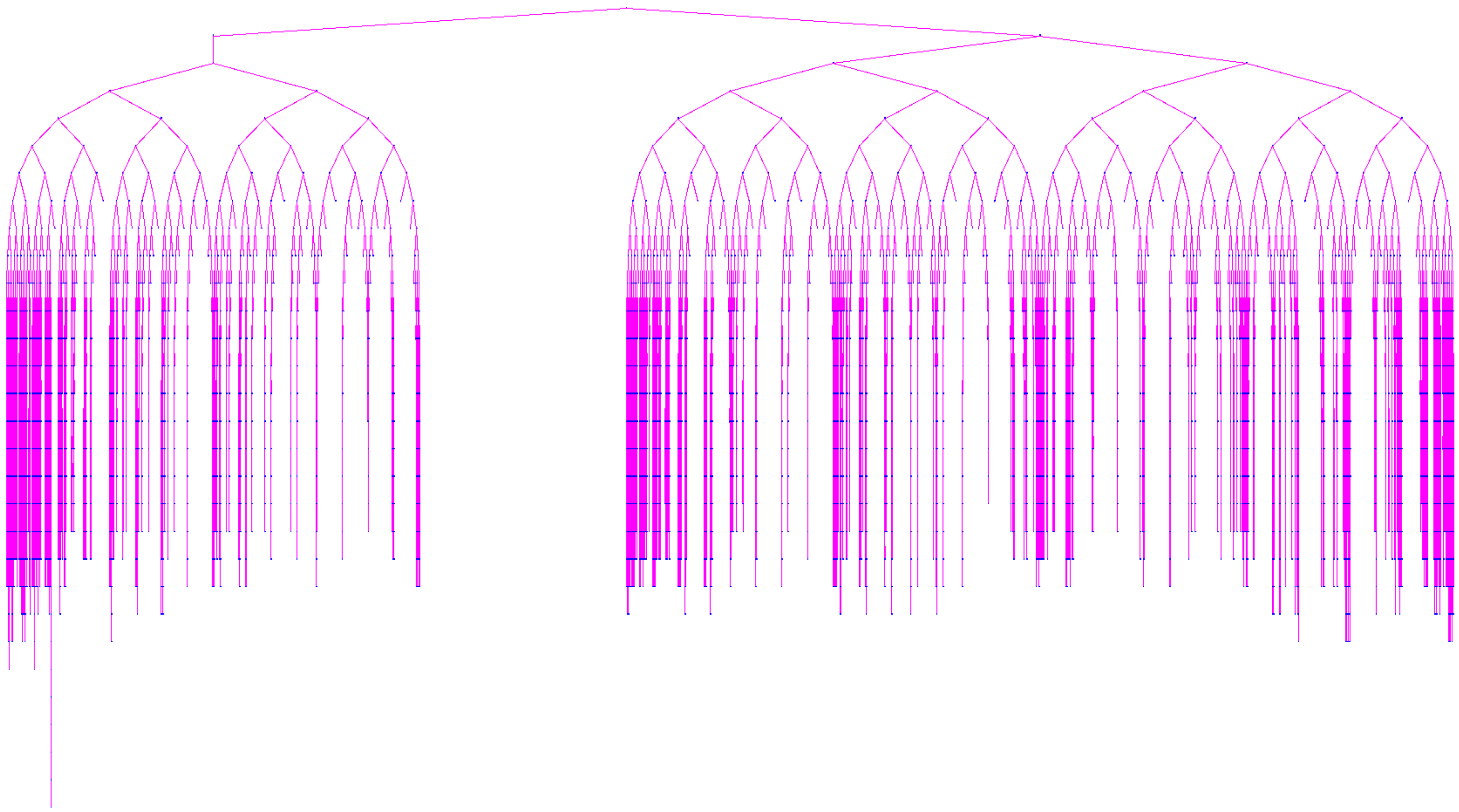

Figure 3. *cct3 decision tree plot*

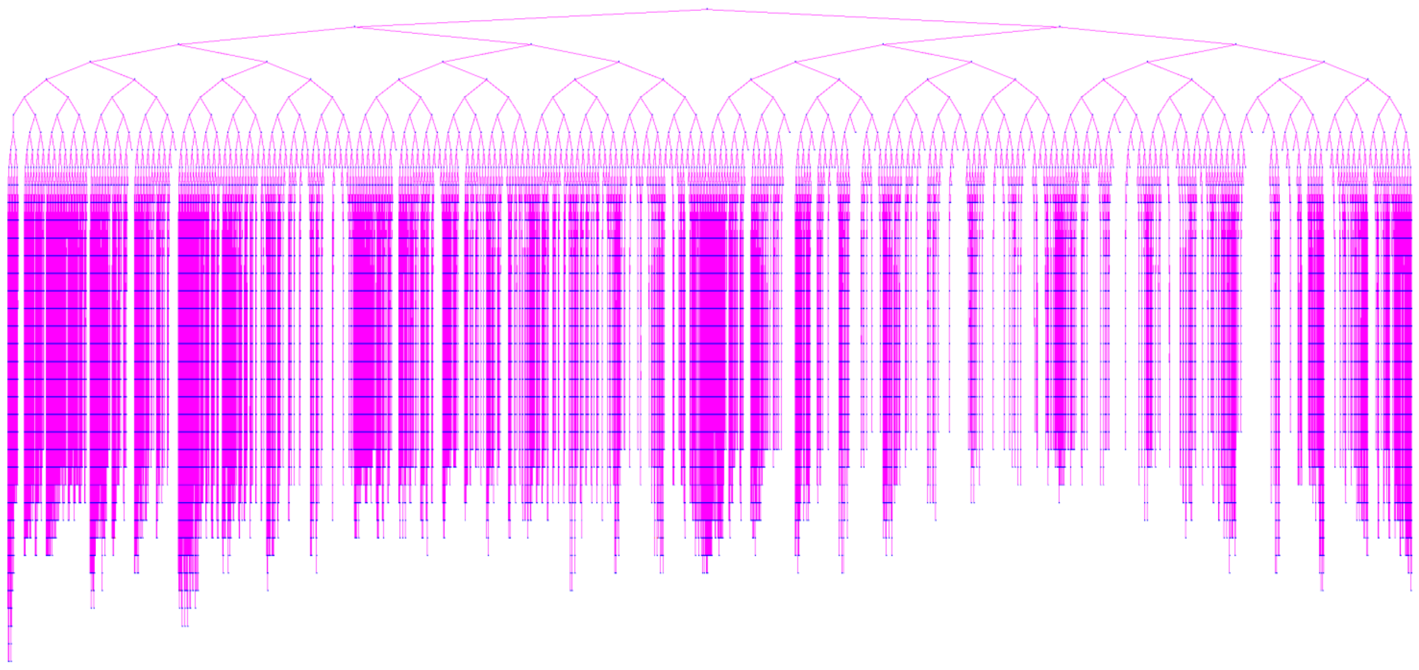

Figure 4. *cct4 decision tree plot*