CPEN513 Assignment 3 - Partitioning

Brandon Freiberger
86799228

I. Description

The goal of this assignment was to implement an efficient version of the Kernighan-Lin based bi-partitioning algorithm capable of dealing with multi-sink nodes. The final algorithm should achieve both low cost with an emphasis on runtime performance.

II. Implementation

A. General Structure

There are 2 main structures in the code: nodes and edges. Both structures are stored as dictionaries, where the key is the node number in the nodes dictionary and the edge or net number in the edge array.

The first entry for each key of the node list stores the edges that the node is associated with. The second entry stores the current cost, and the final entry stores if the node is unlocked or not.

In the edge list, the first entry for each key corresponds to a list containing the constituent nodes as provided in the text input file. The second entry stores whether or not there is a cut in the edge, while the third and final entry stores the number of nodes on the a side of the partition (used in section IV).

B. Efficiency Improvements

There were several optimizations made to make the algorithm run more efficiently:

1) The cost of each edge was stored in the edge structure to avoid having to recalculate all of the edge lengths each time 2 nodes were swapped. This list also contains the amount of nodes that are stored in the ”a” partition.
2) A sublist was created for each node containing the nets it was associated with, which meant that each time a node was swapped, only those nets' lengths would have to be updated.
3) Each time 2 nodes are swapped, only the edges containing those nodes will have their costs updated.
4) The partitioning algorithm ends if there are no better cuts found in a given pass.

C. Cost Improvements

In order to lower the cost of the algorithm, several different gain functions were tried. Initially, the algorithm used the naive approach of adding 1 to the gain for each edge associated to a node that crossed the partition.

As suggested in class, this was changed to using the function 1/(#edges - 1) for the cost of each edge that crossed the partition. However, the results were further improved by changing it to 3/#edges, which is what was used for the final version of the algorithm.

To extract a bit more performance, a correction factor (described in section IV) was added.

III. Results

The average achieved across the 11 benchmarks was 73.3. It is interesting to note the variance on each of the outputs; for example, when running paira, the partitioning algorithm would output a cost of anywhere between 50 and 220. It is likely that with a more careful process of choosing the initial values, the algorithm would have less variance.

The number of iterations the algorithm takes to converge largely depends on the random seed it is run on. Often, it will take a small jump and plateau and exit early. At other times, it will be almost linear with its decent in cost and will not show signs of reaching a local minima, as seen in Fig. 1. Interestingly, running the random placement 50-100 times and taking the lowest initial cost does not seem to have an effect on the final cost.

TABLE I
Cost of Placement

Benchmark	Optimized K&L
cm138a	4
cm150a	6
cm151a	5
cm162a	6
c880	34
alu2	30
e64	85
paira	51
apex1	199
cps	151
apex4	236

IV. Extra

To try and further optimize the K&L algorithm, I implemented a correction factor, as seen on slide 22 of [1]. The correction factor adds a heavier penalty to swaps that have both nodes in the same net. An example can be seen in 2. In the case that both of the red nodes are to be swapped, there will be a factor of 1 added to the gain of both of the red nodes.

This gave a small amount of improvement: the lowest cost of paira decreased from 64 to 51.

V. Running the Code

There is a README file in the github (https://github.com/Undeadpapaya/CPEN513_A3) that provides a guide on how to run the program.

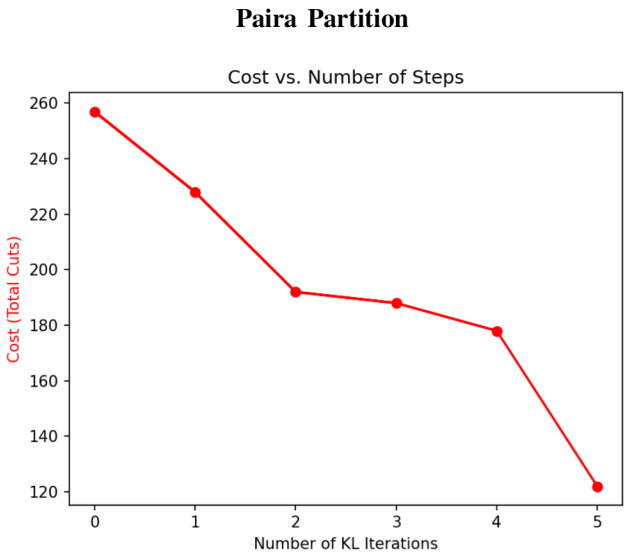

Fig. 1. Example of the partitioning output.

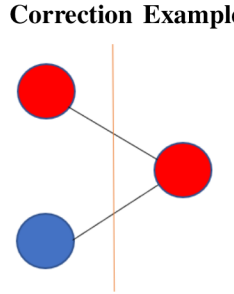

Fig. 2. Example case of the correction term

To get up and running quickly, run the command ”python3 tests.py”. The test file can be changed on line 27 of tests.py - by default, it runs performs partitioning on paira.

The code is placed in the test file in order to verify that the output cost is correct. It should not effect run time by more than a second. There is also a test to ensure the swap function is working correctly.

Running on a Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz with 16GB of ram, Paira takes around 4 seconds to run.

After the last update, the graph will pause for 5 seconds before disappearing. The graph shows the number of cuts made between the partitions as seen in Fig. 1.

VI. References

[1] http://ccf.ee.ntu.edu.tw/ cchen/course/simulation/CAD/unit5A.pdf