Kernighan-Lin Algorithm Implementation

Jose Fraga, University of Arkansas

Abstract—**In this paper I present the implementation of the Kernighan-Lin algorithm. It is composed of the problem formulation, methodology and experimental results. I compare results obtained from the Kernighan-Lin algorithm and the Fiduccia Mattheyses algorithm.**

I. Introduction and Problem Formulation

Being tasked with implementing the Kernighan-Lin algorithm to develop a CAD tool as a final project to CSCE 4013/5013, I created a program to implement the Kernighan-Lin algorithm.

II. Methodology and Algorithm

Kernighan-Lin algorithm swaps a pair of cells, each cell from a different partition. The two cells that generate the highest reduction in cut size are swapped.

A time complexity analysis of Kernighan-Lin algorithm yields the following. Step 0, the initial arbitrary partition yields a time complexity of O(n). Step 1, computing the move cost for each cell yields a time complexity of O(n). Step 2, calculating the first cells to swap yields a time complexity of O(n^2) and later iterations yield a time complexity less than O(n^2). Step 3, return to step 1 and repeat. Step 4, finding the swap sequence with the maximum positive gain yields a time complexity of O(1). Finally step 5, executing m swaps yields a time complexity of O(n^3).

III. Implementation

I implemented the Kernighan-Lin algorithm by creating a program in the Java language. I specifically implemented the algorithm described in [1]. I created various functions. Some functions served directly in the implementation of the algorithm while other functions aided in implementing the algorithm.

Kernighan-Lin algorithm begins with an arbitrary initial partition. My program has many functions just to perform this first task. Function 'readHgrFile()' reads the hypergraph file to extract information pertaining to connections between cells. Then depending on whether a '.part' file was provided or not, function 'readPartFile()' reads or creates a '.part' file. By this point my program has implemented the first part of the algorithm and begins with an arbitrary initial partition.

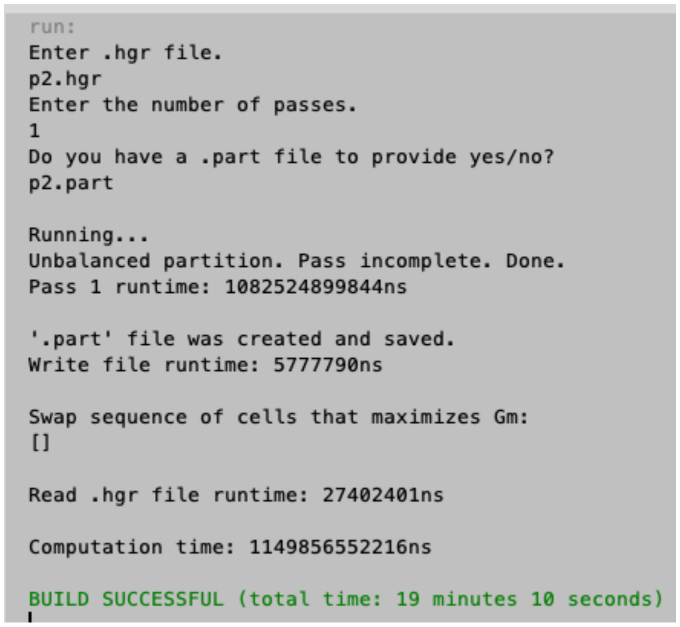
The next step in the algorithm is to set every cell as free. My program iterates through the linked lists of type double arrays containing information of partition zero and partition one. Function 'setFree()' accomplishes this by setting the established index in the double array equal to -1 to mean that the cell is free.

The algorithm proceeds to calculate "D(v)" (move cost) for each cell. Function 'allCostDv()' iterates through all cells and computes the move cost of each cell by calling another function 'costDv()' and finally stores the move cost. Function 'costDv()' receives the pertinent node, searches the data structure containing the nets, retrieves the pertinent net, calls function 'graphConnections()' to find external and internal connection values, and finally calculates the move cost. Function 'graphConnections()' compares the pertinent node to other nodes within the pertinent net and if certain criteria is met, the function returns an appropriate value. Next, the algorithm calculates the max gain obtained from computing the equation $G = D(x) - D(y) - 2 * c(x,y)$. Function 'maxSwapGain()' iterates in a fashion that computes the gain for each cell and saves the max gain along with the associated pair of cells. Once all gains have been calculated, the function calls function 'swapCells()' and passes as parameters the max gain and cells associated with the max gain. Implementing 'maxSwapGain()' in this way was one particular implementation optimization I made. Rather than storing all the calculated swap gains, function maxSwapGain() finds the pair of cells with the max swap gain and updates variables until the max is obtained. I presume this improves the runtime and memory footprint, even if only slightly. It would have been more computationally expensive storing all swap gains, searching the data structure, comparing the values to one another and finally retrieving the max swap gain.

In general the final step the algorithm does is execute the move sequence and swap the cells that maximize the maximum positive gain. In the previous step when my program calls the function 'swapCells()' it executes the final step in the algorithm, swaps the cells and stores the sequence (i.e., it stores the nodes that were swapped). More specifically, the function moves each cell to the opposite partition and updates the data structures accordingly.

IV. Experimental Results

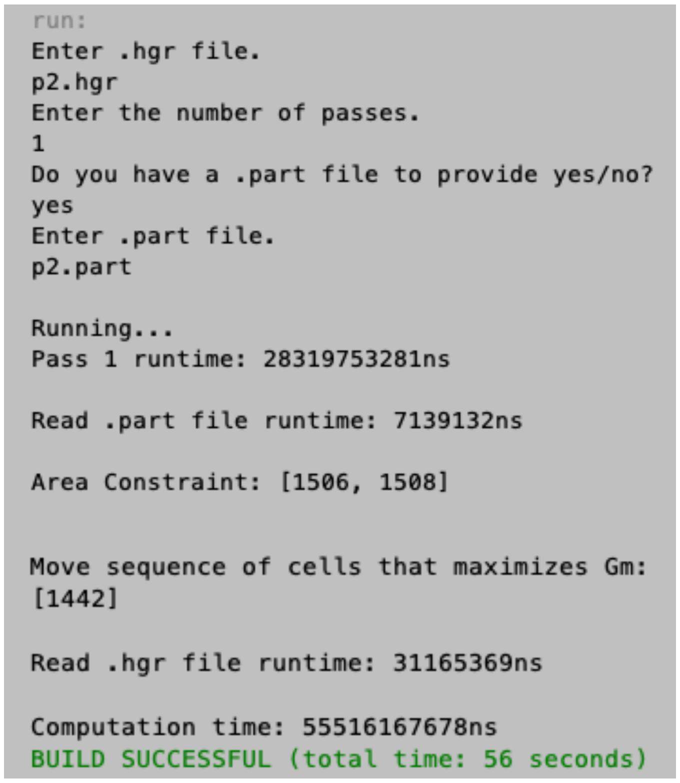
I ran my program on the provided benchmarks 'p2.hgr' and 'structP.hgr' and obtained the following results. I ran the benchmark 'p2.hgr' on the Kernighan-Lin algorithm with an input of 1 pass and obtained a runtime of nineteen minutes with ten seconds.



```
run:
Enter .hgr file.
p2.hgr
Enter the number of passes.
1
Do you have a .part file to provide yes/no?
p2.part

Running...
Unbalanced partition. Pass incomplete. Done.
Pass 1 runtime: 1082524899844ns

'.part' file was created and saved.
Write file runtime: 5777790ns

Swap sequence of cells that maximizes Gm:
[]

Read .hgr file runtime: 27402401ns

Computation time: 1149856552216ns

BUILD SUCCESSFUL (total time: 19 minutes 10 seconds)
```

Fig. 1. Benchmark 'p2.hgr' on Kernighan-Lin algorithm implementation.

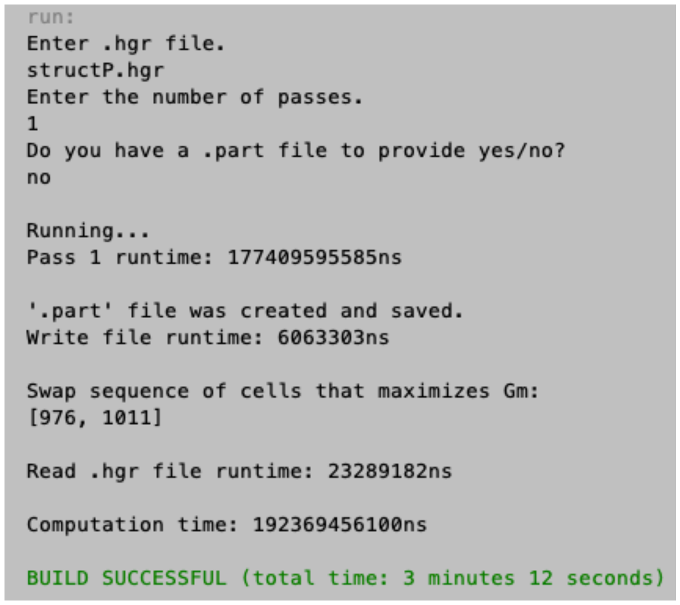
I ran the benchmark 'p2.hgr' with an input of 1 pass on the Fiduccia-Mattheyses algorithm and obtained a computation time of 56 seconds.



```
run:
Enter .hgr file.
p2.hgr
Enter the number of passes.
1
Do you have a .part file to provide yes/no?
yes
Enter .part file.
p2.part

Running...
Pass 1 runtime: 28319753281ns

Read .part file runtime: 7139132ns

Area Constraint: [1506, 1508]


Move sequence of cells that maximizes Gm:
[1442]

Read .hgr file runtime: 31165369ns

Computation time: 55516167678ns
BUILD SUCCESSFUL (total time: 56 seconds)
```

Fig. 2. Benchmark 'p2.hgr' on Fiduccia-Mattheyses algorithm implementation.

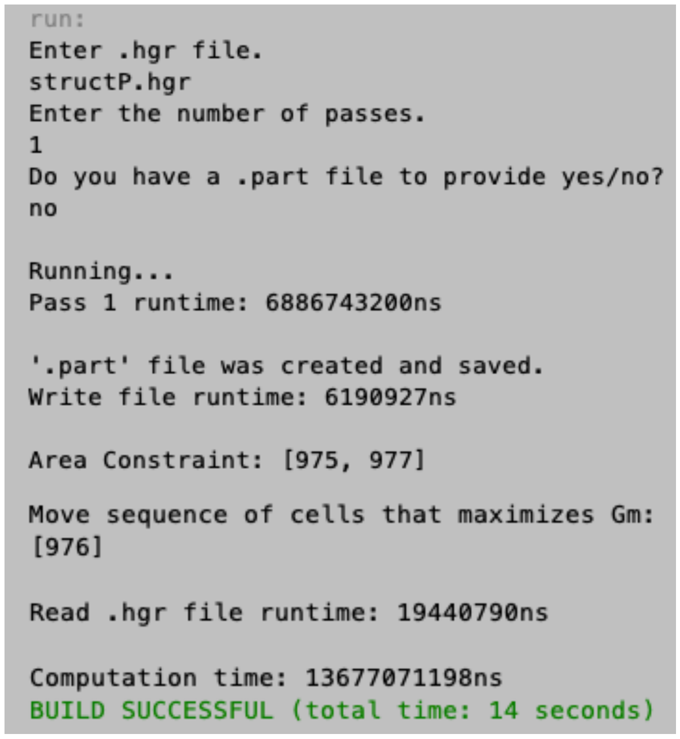
I ran the benchmark 'structP.hgr' on the Kernighan-Lin algorithm with an input of 1 pass and got a computation time of three minutes and twelve seconds.



```
run:
Enter .hgr file.
structP.hgr
Enter the number of passes.
1
Do you have a .part file to provide yes/no?
no

Running...
Pass 1 runtime: 177409595585ns

'.part' file was created and saved.
Write file runtime: 6063303ns

Swap sequence of cells that maximizes Gm:
[976, 1011]

Read .hgr file runtime: 23289182ns

Computation time: 192369456100ns

BUILD SUCCESSFUL (total time: 3 minutes 12 seconds)
```

Fig. 3. Benchmark 'structP.hgr' on Kernighan-Lin algorithm implementation.

I ran the benchmark 'structP.hgr' on the Fiduccia-Mattheyses algorithm with an input of 1 pass and got a computation time of fourteen seconds.



```
run:
Enter .hgr file.
structP.hgr
Enter the number of passes.
1
Do you have a .part file to provide yes/no?
no

Running...
Pass 1 runtime: 6886743200ns

'.part' file was created and saved.
Write file runtime: 6190927ns

Area Constraint: [975, 977]

Move sequence of cells that maximizes Gm:
[976]

Read .hgr file runtime: 19440790ns

Computation time: 13677071198ns
BUILD SUCCESSFUL (total time: 14 seconds)
```

Fig. 4. Benchmark ‘structP.hgr’ on Fiduccia-Mattheyses algorithm implementation.

The obtained results indicated that the Fiduccia-Mattheyses algorithm was much faster than the Kernighan-Lin algorithm.

V. Conclusion

After successfully implementing Kernighan-Lin algorithm in the Java language I was able to create my CAD tool as my final project successfully. Implementing the algorithm helped me understand at a much deeper level the inner workings of the algorithm. I learned how this algorithm finds high-quality circuit partitioning solutions. Also, the successful implementation allowed me to obtain experimental results that allowed me to compare to the Fiduccia-Mattheyses algorithm experimental results. The experimental results made clear the difference between the Kernighan-Lin algorithm and the Fiduccia-Mattheyses algorithm in terms of efficiency.

References

[1] Kahng, Andrew B. *VLSI Physical Design: from Graph Partitioning to Timing Closure.* Springer, 2011.