Lab 4 Report

Matthew Crump A01841001
Nicholas Williams A02057223

Objectives

Implement a simulated annealing program in C or C++ that solves the FPGA placement problem. Provide statistical analysis of program execution and results.

Procedure

We started out by reading the lab-4 requirements for the simulated annealing program and reviewed lecture slides about the annealing algorithm. We chose to make the program in C++ as we wanted to use classes and the sstream library to read in input data files. We broke down the program into 5 main sections: reading the input file, calculating the cost, storing location of nodes, implementing the annealing algorithm, and moving nodes in various ways. Though not a requirement, we wanted to print out a grid with the node locations for easy visualization of the result.

We used the sstream library to read in each row of the input file then cast them to integers. The first numbers were used to instantiate our class, taking in the number of nodes and the grid size as parameters. Then the edges were added one at a time to a vector. For the final part of the setup, we needed to place each node at some starting location. We were going to randomly place them, but found that placing them one after another along the top of the grid worked just as well.

As discussed in lecture, the cost of each edge should be the square of the length between the two nodes (without diagonal paths). To find the length we subtracted the x and y values for each node and then took the absolute value. Each edge also kept track of its length. If the edge's length was the shortest found, its corresponding nodes would be locked. This way nodes that are placed optimally would be left alone while edges with longer lengths would be targeted. If an edge was no longer the smallest, then its corresponding nodes were unlocked.

We had three main ways of moving nodes around to potentially decrease the cost of each edge. Our program randomly selects one of the three ways of moving nodes every iteration. To cause some uphill moves, we randomly selected two nodes and swapped them or picked a random node and moved it to a random empty spot. The last option to create a more optimal placement was to take the longest edge and move its two nodes closer together. Each of these three methods only picked nodes that were not locked.

Results

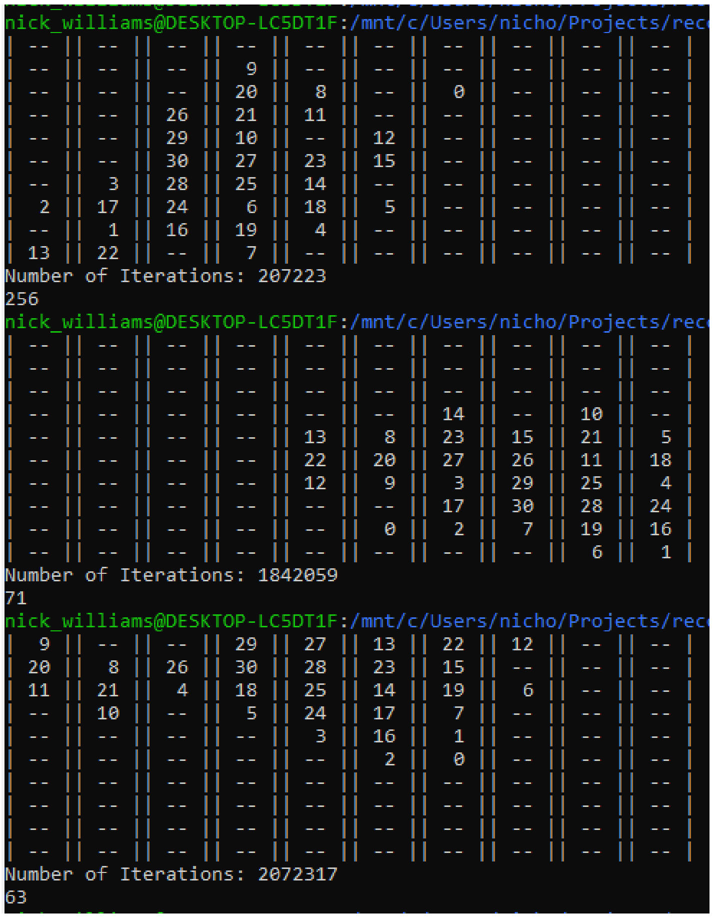
Once we got our code successfully reading in the data and placing it into the grid we started working on our adjustment functions. Our code initially places the nodes into the top of the grid. Our cost function successfully computed the sum of all the edges squared. To make sure that our adjustment functions were helping we compared the average final cost from before and after implementing the adjustment function.

The first adjustment function we created picked two nodes at random and swapped their location. After implementing the first adjustment function, the nodes continued to occupy the top of the grid, but the nodes were no longer in sequential order. The average final cost decreased slightly after implementing the first adjustment function.

The second adjustment function we created picked one node at random and moved the node to a random empty space in the grid. This was an important step because it opened the grid up so the nodes were no longer only occupying the top couple rows of the grid. The average final cost stayed about the same as it was after implementing the first adjustment function.

The third adjustment function took the two nodes with the longest edge and placed them next to each other in the grid. In addition to adding the third adjustment function, we created a shortest length threshold so that nodes with edges smaller than the shortest length threshold were temporarily locked. Then the three adjustment functions were able to focus on the nodes with the largest edges. With these changes and all three adjustment functions in action, the nodes were suddenly regrouping together in the grid with a much lower final cost on average.

Figure 1 shows three examples of all three adjustment functions working with different cooling rates. The first output shows that the simulated annealing algorithm went through 207,223 iterations, and the final cost was 256. The second output shows the final cost of 71 after 1,842,059 iterations. The third output shows a cost of 63 after 2,072,317 iterations. This helps to show that as the number of iterations increase, the total average cost tends to decrease. The relationships of cooling rate, time spent, and solution quality will be analyzed further in the analysis section.



```
nick_williams@DESKTOP-LC5DT1F:/mnt/c/Users/nicho/Projects/rec
| -- || -- || -- || -- || -- || -- || -- || -- || -- || -- |
| -- || -- || -- ||  9 || -- || -- || -- || -- || -- || -- |
| -- || -- || -- || 20 ||  8 || -- ||  0 || -- || -- || -- |
| -- || -- || 26 || 21 || 11 || -- || -- || -- || -- || -- |
| -- || -- || 29 || 10 || -- || 12 || -- || -- || -- || -- |
| -- || -- || 30 || 27 || 23 || 15 || -- || -- || -- || -- |
| -- ||  3 || 28 || 25 || 14 || -- || -- || -- || -- || -- |
|  2 || 17 || 24 ||  6 || 18 ||  5 || -- || -- || -- || -- |
| -- ||  1 || 16 || 19 ||  4 || -- || -- || -- || -- || -- |
| 13 || 22 || -- ||  7 || -- || -- || -- || -- || -- || -- |
Number of Iterations: 207223
256
nick_williams@DESKTOP-LC5DT1F:/mnt/c/Users/nicho/Projects/rec
| -- || -- || -- || -- || -- || -- || -- || -- || -- || -- |
| -- || -- || -- || -- || -- || -- || -- || -- || -- || -- |
| -- || -- || -- || -- || -- || -- || -- || -- || -- || -- |
| -- || -- || -- || -- || -- || -- || 14 || -- || 10 || -- |
| -- || -- || -- || -- || 13 ||  8 || 23 || 15 || 21 ||  5 |
| -- || -- || -- || -- || 22 || 20 || 27 || 26 || 11 || 18 |
| -- || -- || -- || -- || 12 ||  9 ||  3 || 29 || 25 ||  4 |
| -- || -- || -- || -- || -- || -- || 17 || 30 || 28 || 24 |
| -- || -- || -- || -- || -- ||  0 ||  2 ||  7 || 19 || 16 |
| -- || -- || -- || -- || -- || -- || -- || -- ||  6 ||  1 |
Number of Iterations: 1842059
71
nick_williams@DESKTOP-LC5DT1F:/mnt/c/Users/nicho/Projects/rec
|  9 || -- || -- || 29 || 27 || 13 || 22 || 12 || -- || -- |
| 20 ||  8 || 26 || 30 || 28 || 23 || 15 || -- || -- || -- |
| 11 || 21 ||  4 || 18 || 25 || 14 || 19 ||  6 || -- || -- |
| -- || 10 || -- ||  5 || 24 || 17 ||  7 || -- || -- || -- |
| -- || -- || -- || -- ||  3 || 16 ||  1 || -- || -- || -- |
| -- || -- || -- || -- || -- ||  2 ||  0 || -- || -- || -- |
| -- || -- || -- || -- || -- || -- || -- || -- || -- || -- |
| -- || -- || -- || -- || -- || -- || -- || -- || -- || -- |
| -- || -- || -- || -- || -- || -- || -- || -- || -- || -- |
| -- || -- || -- || -- || -- || -- || -- || -- || -- || -- |
Number of Iterations: 2072317
63
```

Figure 1. Sample output when using input.txt

Analysis

Using Python, we ran our program over 64,000 times with various cooling rates. The initial temperature was 1 billion with a stop threshold of 0.01. We started the cooling ratio at 6.103e-5 and for each iteration, increased it by that same amount with a maximum value of 0.9999. Figure 2 shows the plotted data with the best cooling ratio at approximately 0.94 which gives a runtime for input3.txt of 200 milliseconds. A larger value of 0.94 for the cooling ratio did not seem to yield any better results making it not worth the extra time to compute a solution.

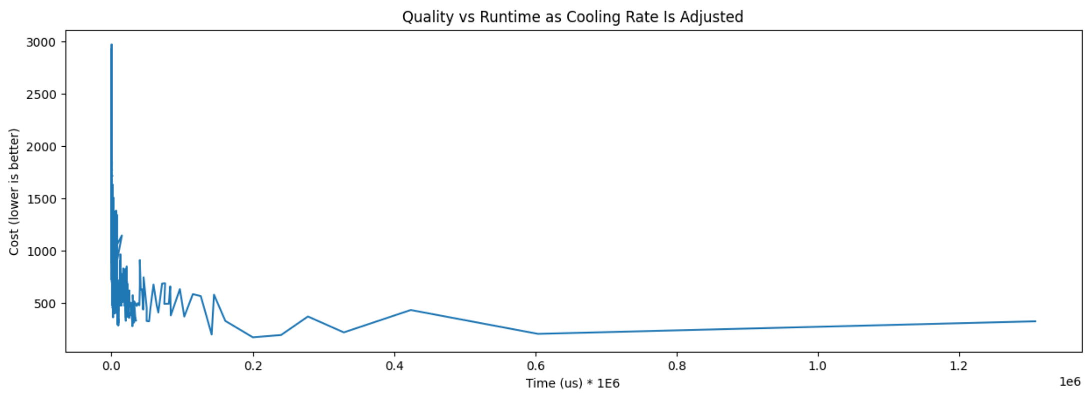


Figure 2. Annealing plot

Due to the way we moved nodes around randomly to find new solutions, the results had a lot of noise in them when under a few iterations. After many iterations, the quality of the solution would actually slightly decrease. Sometimes the optimal/best solution is not found because the random movements lock “good” nodes which can set the solver down a bad path that it will not recover from. In other words, this is probably because we locked nodes that were near the shortest length possible, making it so that there are only a handful of possible solutions to explore.

If we implemented more strategy and tactics to how nodes are moved around, we would probably see a slow increase to the quality as the runtime increases rather than a slow decrease. Despite these flaws, our implementation did trend towards a good solution after a reasonable amount of iterations.

Conclusion

The purpose of this assignment was to gain a better understanding of how Quartus potentially handles the placement of nodes and edges onto our FPGAs through simulated annealing. The algorithms that we used in our design found good solutions but potentially not the best solution. We used the simulated annealing algorithm that we discussed in class to randomly call one of three adjustment functions. The adjustment functions randomly moved a node, swapped two nodes, or moved the nodes with the biggest edge together.

One small improvement could be to add a final pass over the grid, check for nodes that are all by themselves, and move them closer to their connected nodes. Occasionally, a node would be out by itself in the grid at the end of the program execution. Fixing this problem could potentially decrease the final cost by 25 to 150 points. It was very satisfying to see simple random movements produce clusters of optimal node placement once we implemented the locking/unlocking node feature.