Harsh Patel- 04/28/2018

ECE 4930 EDA Project 2 Report

Project Environment Details:

This project was developed in Visual Studio 2015 and should be opened using the "EDA2.sln" file provided in this archive.

To build the project:

1. Select your machine architecture at the top using the drop down (x86/x64).
2. Next add your input file to the /EDA2/EDA2/benchmarks folder.
3. In Visual Studio go to Projects>Properties>Debugging and in the field next to Command Arguments add the input file name as argument like this:
    a. benchmarks\<filename>
4. Now click on Local Windows Debugger (found at the top next to a green play arrow)
5. After program builds and runs, <filename>.out will be found in the benchmarks directory.
6. Steps 2-5 need to be repeated for each new input file or alternatively you can find the executable, place inputs in that directory and use the console window.
    a. > EDA2.exe <filename.in>

Project Organization:

The project is broken into 4 modules which contain either a class for data structuring, some algorithms or the main driver.

1. gate.h/gate.cpp – Contains the data structure for a Gate.
    a. Data members:
        i. string type – NAND, NOR, AND etc.
        ii. vector<Gate*> in-/out – These two vectors serve as hyper edges to other Gates that either fan in to or out of this Gate.
2. circuit.h/circuit.cpp – Contains the data structure for all of the external inputs, outputs and internal gates of the circuit.
    a. Data stored in unordered_map<string, Gate*> which is a C++ STL container that is implemented as a hash table. This is a change from project 1's use of "map".
        i. This was used to improve access/insert time from O(logn) to O(1).
        ii. Note about unordered_maps: they store STL containers called a pair which is basically a <key, data> pair as the name suggests. In this case key is the gate id (gate 10 has id G_10).
    b. Data members:
        i. Unordered_map<string, Gate*> inputs/gates/outputs – These three maps are used to keep the inputs, outputs and gates separate while with the use of pointers still maintaining a hyper graph structure for later projects.
3. partitioner.h/partitioner.cpp – Contains a data structure to keep a bi-partition of the provided circuit.
    a. Data is kept as the 2 partitions A and B which are of type unordered_map<strong, Gate*> as well as a vector of keys (gate IDs) for direct access during simulated annealing.

   b. Along with these, there are some data members that keep track of current cut size, and partition size.
4. main.cpp – basically just the driver program that calls methods from algorithms.cpp to retrieve statistics on the inputted circuit.

Algorithms Runtime Analysis:

1. parseFile – found in circuit.cpp this private method is called by the constructor when instantiating a Circuit.
   a. Run time complexity O(n) – Each time a gate is found you need at least 1 search of an unordered_map, which we said was a hash table therefore this is O(1). You also need to iterate through all n gates in the file so total complexity is O(n).
   b. The remaining functionality is the same from EDA 1.
2. PartitionSA – The Simulated Annealing partitioner. When a partitioner object is created, all of the elements are randomly distributed into the two partitions while keeping track of balancing restrictions.
   a. Run time complexity of O(MN) where M is moves per temperature step and N is iterations required to get to the freezing temperature. Both are dependent on the number of total gates in the circuit. (More on this below)
   b. A lot of measure were taken to avoid unnecessary computation.
      i. Functions to compute cost without moving the gates. This helps so that if a move is accepted, you can move the gates and then just add the previously found Δcost to the previous cut size. If the move is not accepted then just continue.
      ii. Use of vectors to hold the keys to allow for direct random access of the keys. Even though the unordered_map has access time of O(1), it isn't index accessible.

Simulated Annealing Framework:

1. **Overview:** The algorithm is show below in **Figure 1**. It is the same algorithm that was given as pseudocode in the lecture notes (S3 – Floor planning).
   a. Each iteration (within each temperature step) the algorithm randomly selects one of the 3 possible moves (shown below) and then selects random gates from each partition to perform the moves on.
   b. The cost of each move is determined without doing the move to save some time.
      i. There is a bug that causes the cut size and some cost values to be calculated incorrectly. I couldn't debug it completely, so the solution instead was to recalculate the cut size after the algorithm completed. Completely recalculating cut size is expensive and so is only called once.
2. **Allowed Moves:**
   a. There are 3 different moves that the annealer can choose from:
      i. Move from A to B
      ii. Move from B to A
      iii. Swap one from A and one from B
   b. In order to maintain a balance of at most 60 to 40, the acceptance function only accepts moves if they move would maintain the required ratio.

```
// Param init
To = Ti = 40000.0;
Tf = Ti / pow(10, (floor(log10(Ti) + 1)));
int totalNodes = getNodesA() + getNodesB();
coolFactor = 0.97 + (0.001 * log(totalNodes)) ;
k = .00025;
moves_per_temp = (int)ceil(sqrt(totalNodes)*log(totalNodes));
double alpha = 0.9995;

double accepted = 0.0;
while (Ti > Tf) {
    accepted = 0.0;
    for (int i = 0; i < moves_per_temp; i++) {
        a = (int)floor(uniform(mt) * this->getNodesA() );
        b = (int)floor(uniform(mt) * this->getNodesB() );
        move = (int)floor(uniform(mt) * 3);
        cost = getCost(move, a, b);
        if (acceptMove(k, Ti, cost, uniform(mt), move)) {
            makeMove(move, a, b);
            accepted++;
            cutSize += cost;
        }
    }

    Ti *= coolFactor;  // cool down
    moves_per_temp = (int)((double)moves_per_temp / (alpha));
```

Figure 1. Simulated Annealing Algorithm

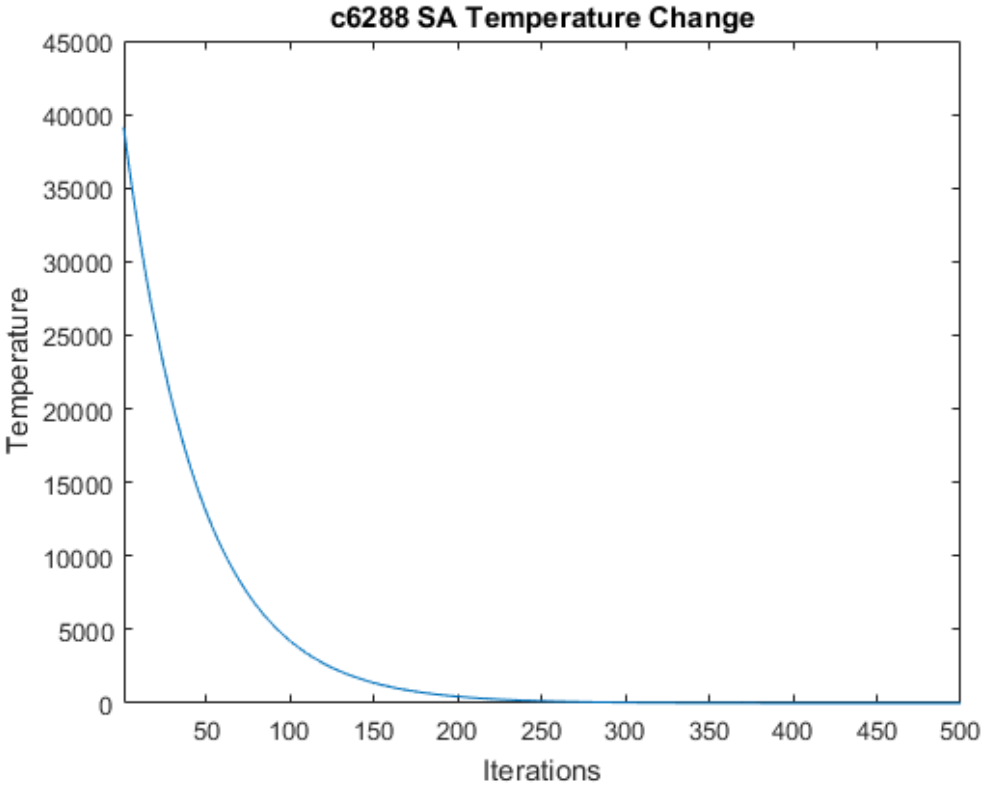
3. **Parameter Tuning:**
    a. **To** (initial temperature) – This was set at 40000. I initially started at 1000 and worked up to here. I found that the temperature range didn't have as much effect on the results as did the cooling rate. This was a good starting point if cooling rate varied for smaller circuits.
    b. **Tf** (freezing temperature) – This was set using the formula shown on line 2 of **Figure 1**. For example, for To = 40000 -> Tf = 0.4. (1000 -> 0.1) This is sufficiently small enough, it won't have any more effect in the Boltzmann function so no sense in going any lower.
    c. **k** (Boltzmann constant) – originally, I had a similar formula to the one for Tf (mapping is Ti = 40000 -> 0.0025) This was the Boltzmann value for acceptance was about 0.99 at Ti = To, then about 0.9 at Ti = To/10, 0.6 for Ti = To/100 and so on approaching 0 from there. I changed it to a constant 0.00025 to allow a little more acceptance at lower temperatures. Changing this number from here made the results worse overall (leaving other parameters unchanged).
    d. **coolFactor** – the cooling rate is related to the size of the circuit, I found this rate had a larger effect on the outcome than did setting Ti of Tf. Setting the initial rate to 0.97 did well for most circuits. Adding the little extra as shown by the formula below, improved the results of the larger circuits. Cooling slower allowed more chances to escape local minimums in larger circuits.

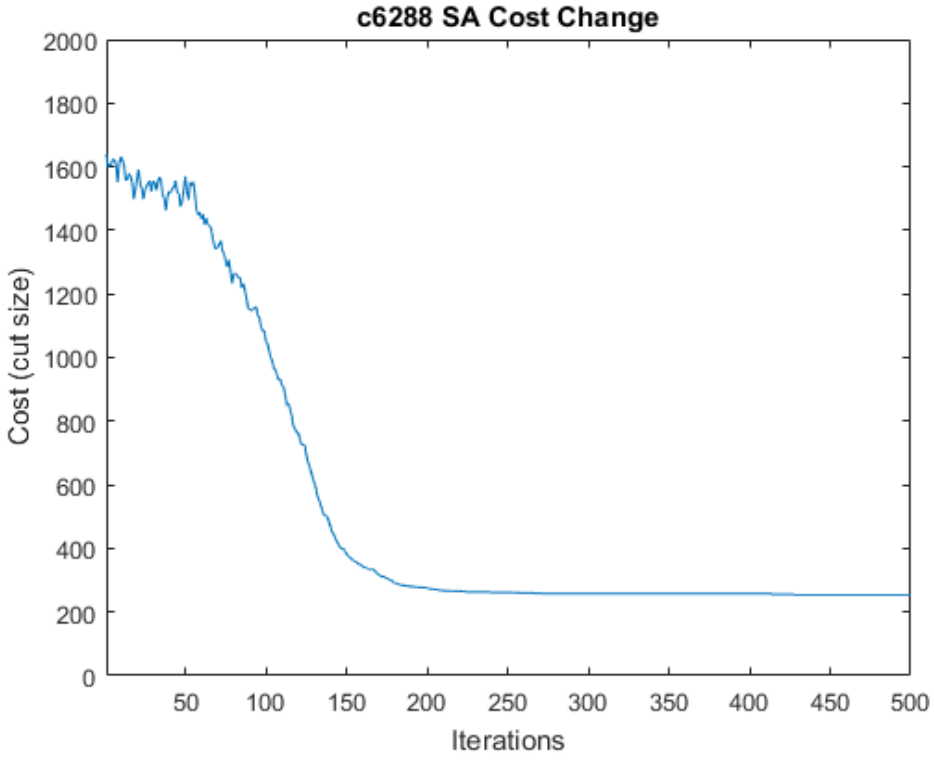
e. **moves_per_temp** – originally this was a constant (~200) however a number that large is unnecessary for c17 and too small for larger circuits. The new formula (**Figure 1**) again is based on the size of the circuit. The way I came up with this formula was I tried to do something like what I did for the cooling factor and I kept tweaking it until the output of the formula was something reasonable. (for example, for a netlist of 3500 gates about 600 iterations is good, but I don't want something massive like 1000). For smaller circuits, say c17, this number will be around 10 which is plenty for a circuit that small.
f. **alpha** – This is a constant which is used to increase the number of iterations per temperature as the algorithm runs on. This allows more chances to get out of a local minimum as the temperature cools down. The number needs to be as close to 1 as possible otherwise moves_per_temp will approach infinity very quickly. The maximum ratio between initial moves and the final moves is around 1.3 which is not too high.

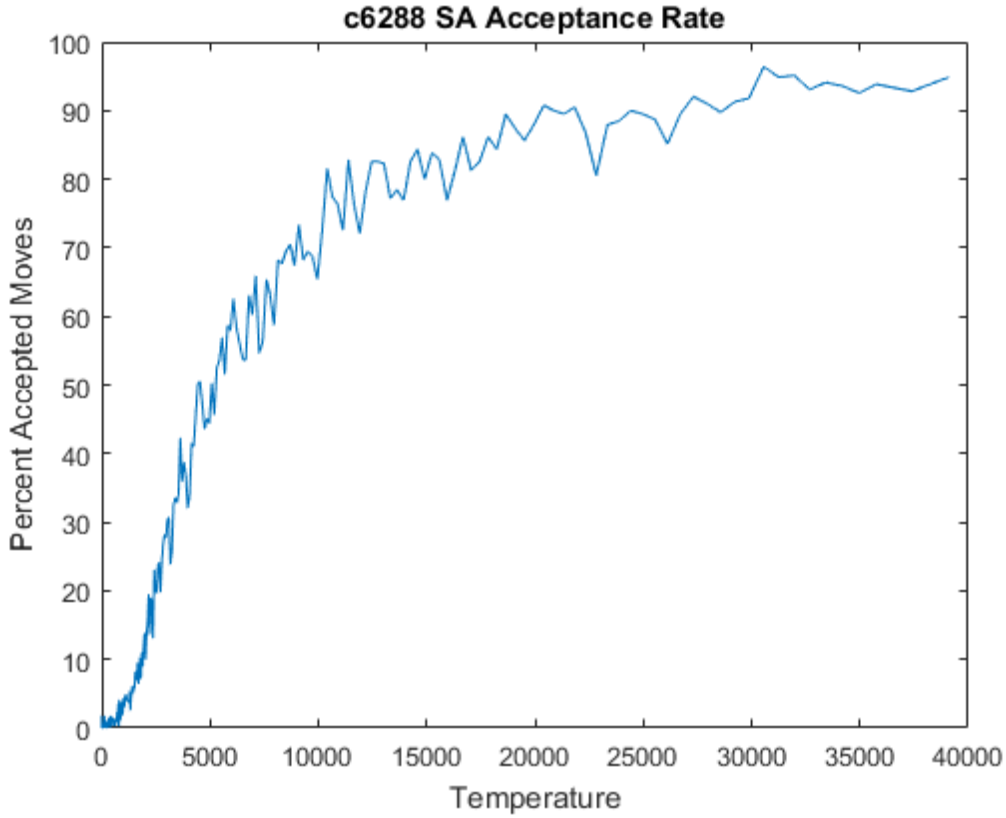
Closing Remarks:

This was a very interesting project, overall the code wasn't very difficult however parameter tuning was a chore. I still feel I could tune it better if I had more time, but I am pleased with the runtime and quality of the current solutions. While tuning parameters I determined that the 3 most important parameters are cooling rate, k and moves_per_temp. Overall the starting and ending temperature didn't make much different raised past 40000. Although this may have just been a coincidence or error on my part. I wanted to attempt the FM algorithm but I started too late on the project and decided to just complete SA. It's interesting how effective an algorithm based on randomness can still be with proper parameters. I came across several variations on algorithms for cooling, moves per temp, etc. while researching. Safe to say there is a lot of variation in this algorithm that I haven't even seen yet. In short, tuning was a little difficult but it was a fun project.

Appendix A: c6288 Result Plots






c6288 SA Acceptance Rate
Percent Accepted Moves
Temperature
0
10
20
30
40
50
60
70
80
90
100
0
5000
10000
15000
20000
25000
30000
35000
40000