# VLSI Assignment – 1

**Duy Tran**

**M0744762**

1. **Program structures:**

Program data structures and functions used in this Simulated Annealing program is prototyped in partprogh.h header file. The main cpp file is SA.cpp

1. **NetIn**

\_ Simple read in from the net list files. Store the number of cells in c, number of net in n, the nets in the net vector.

1. **CellDistArray**

\_ Structure to hold information about cell distribution to 2 sets. Implemented as a vector that hold boolean values (set 0 or 1) at each location (cell).

1. **NetMatrix**

\_ Structure to hold information about the cell’s nets. Take in net vector from Net In and break down the 2 nodes on each net. Implemented as an adjacency list (vector of vectors). Each vector of a row holds all the nodes that is connected to the cell of that row location. Ex: Vector[i] holds the list of nodes connected to cell i.

\_Initally, a matrix of int was implemented to hold the nets and net weights. However, in trying to tackle B9 the matrix memory required was 250000 x 250000 x 4 bytes = 125 GB. This was unfeasable. Using an adjacency list, the memory required was 250000 x 2 x 4 bytes = 2 MB.

1. **costFunc**

\_The cost function take the new solution, the NetMatrix’s adjacency list and the 2 cells used in moveFunc and return delta cost. The logic is as follow:

+ Suppose a cell has

**a** nets connected to cells of different set.

**b** nets connected to cells of the same set.

**c** nets connected to the cell that will be swapped with it (assuming 2 swapping cells are on different sets)

+ The old solution will have cost of: a1 + a2 + c

+ The new solution will have cost of: b1 + b2 + c

* The delta cost will be: b1 – a1 + b2 – a2

\_ The cost fucntion go through the net list at the input cells in the adjacency list. Because the input is the new solution, if the nets connect cells on the same side, they will be counted to a1 and a2. If the nets connected to different sides, they will be counted to b1 and b2.

1. **moveFunc**

\_Swap 2 randomly generated cells taken from the main program.

1. **acceptMove**

\_ Implemented as shown in lecture. K = 2.72

1. **Tuning**

\_ Tuning method is as followed:

+ Run the program from 40000 to 4 with cool rate of 0.00001 and 250 moves each step. This way the program will run through 1 temperature loop. Check the execution time T

+ T/250 = T\_ave is average time to execute 1 loop

+ Solve for x in 40000 \* x^[(Desired time / T\_ave) / moves] = 0.1

\_In testing bench\_16, using 40000 to 0.1 with cool rate of 0.95 and 250 moves each step resulted in the program ending too early and produce suboptimal result.

\_ Thus, 2 minutes was chosen as Desired time. 500 moves per temp test was chosen to increase the cooldown rate. Thus the cool down rate is approximately 0.998 on my laptop.

\_ These tuning parameters was chosen also used for benches and the test bench B1, B2, B3 with less cell and net numbers because bench\_16 was the most challenging in the example nets.

Bench\_2: 8 seconds

Bench\_4: 10 seconds

Bench\_6: 10 seconds

Bench\_11: 16 seconds

Bench\_12: 38 seconds

Bench\_16: 2 minutes

\_ From B4 forward, running with the above parameters resulted in the programming ending too early. Thus from tuning B10 for 3 hours the cooldown rate is changed to 0.9999. These parameters were used for B4 to 10

\_ Final parameters:

B1, B2, B3: T0 = 40000; T\_freeze = 0.1; T = 0.998 \* T; moves = 500

B4->10: T0 = 40000; T\_freeze = 0.1; T = 0.9999 \* T; moves = 500

\*Notes: In B9 and B10, it might be better if Temp is larger, as delta\_cost might be much larger than Temp resulting in affecting Boltzman number (spending several initial steps not accepting bad result)

1. **Statistic**

The below chart is of the test in bench\_16 with the tuning parameters mentioned above

1. **Test bench execution time:**

B1: 9 seconds

B2: 36 seconds

B3: 1.8 minutes

B4: 5 minutes

B5: 6 minutes

B6: 6 minutes

B7: 8 minutes

B8:

B9: 1 hour

B10: 3 hours