



THE UNIVERSITY OF TEXAS AT DALLAS

Erik Jonsson School of Engineering and Computer Science

Standard cell placement engine

EEDG 6375 : Design Automation of VLSI Systems

Date,
December 8, 2018

Submitted by,
Akash Anil Tadmare (aat171030)
Akshay Nandkumar Patil (anp170330)

TABLE OF CONTENTS

1. INTRODUCTION:	1
1.1 IMPORTANCE OF PLACEMENT:	1
2. CIRCUIT PLACEMENT USING SIMULATED ANNEALING:	2
2.1 ALGORITHM DETAILS:	2
2.1.1 <i>Hill Climbing:</i>	3
2.1.2 <i>General Algorithm:</i>	3
2.1.3 <i>Placement problem:</i>	3
2.2 IMPLEMENTATION DETAILS:	4
2.2.1 <i>Data Structures:</i>	4
2.2.2 <i>Implementation:</i>	6
2.2.3 <i>Internal loop criteria:</i>	10
2.2.4 <i>Exit Criteria:</i>	10
2.3 TESTING AND RESULTS	11
2.4 CONCLUSION:	14

LIST OF FIGURES

FIGURE 1: SOLUTION SPACE	2
FIGURE 2: WIRELENGTH CALCULATION	5
FIGURE 3: DIMENSION DETAILS OF THE DIE	6
FIGURE 4: COST V/S NUMBER OF MOVES	10
FIGURE 5: IMPROVEMENT V/S TIME FOR BENCHMARK IBM01	12

LIST OF TABLES

TABLE 1: NODE CLASS	4
TABLE 2: STRUCTURE OF 'CELL' MAP	4
TABLE 3: NET CLASS	4
TABLE 4: STRUCTURE OF MAP NET	5
TABLE 5: COOLING SCHEDULE	11
TABLE 6: BRIEF DESCRIPTION OF BENCHMARKS	11
TABLE 7: EXECUTION RESULT FOR BENCHMARK IBM01	12
TABLE 8: ANALYTICAL RESULT FOR REST OF THE BENCHMARKS	13

1. Introduction:

VLSI world completely revolves around integrated circuits (IC). Manufacturing process of an IC follows through a lot of different steps, out of which typically IC design cycle involves following steps:

1. System Specification.
2. Architectural Design.
3. Logic Design.
4. Logic Synthesis.
5. Physical Design.
6. Physical verification and Tapeout.
7. Wafer Fabrication.
8. Packaging.

Amongst all these steps, we will be dealing with Physical Design. Once the logical description of the design has been completely synthesized into a netlist, the physical design converts this netlist based circuit representation into a geometrical representation.

Physical design further involves following steps:

1. Partitioning.
2. Floor-planning.
3. Placement.
4. Clock-tree synthesis.
5. Routing.
6. Timing verification.

It would be very time consuming or even impossible to hand-craft the circuit design of large complexity without any assistance of computer programs during all the steps discussed above. According to Moore's law, number of transistors on a single chip doubles every 2 years, the state of the art chips consists of an entire system with millions of transistors. For example, Pentium processors from Intel has 3 million transistors. Computer programs are used widely into the industries to automate these physical design processes with no or very little human intervention. As a part of Design automation of VLSI systems course. We have implemented automatic partitioning and placement engines.

1.1 Importance of Placement:

Placement is another very important stage in physical design. In this stage, the geometrical locations are assigned to each cell in the circuit. The factors taken into consideration for placement mainly involves, interconnections between the cells e.g. nets. The aim of the placement is to ensure that every cell in the circuit is connected to the other cell through minimum required wire length which will minimize the total wire length of the circuit. Minimum wire length reduces the routing area which will reduce the cost of the system. It will also reduce the complexity for the routing stage of physical design process.

2. Circuit placement using Simulated Annealing:

Traditionally, Placement is the step after logic synthesis and before routing in the VLSI circuit design flow. Using the netlist generated in logic synthesis, locations of the cells in the netlist are determined in placement step. Placement of the cells plays a very important role in circuit design. Badly placed designs results in unnecessarily increased wire-lengths. Interconnect delays are directly proportional to the wire lengths so increased wire length results in long interconnect delays. Increased wire length also increases the capacitive load which increases the switching delays of the components thereby increasing their power consumption.

As routing stage is followed by placement, the routing solution highly depend on the final placement solution. Well placed circuit can reduce the routing effort and also reduces the chances of having non routable wires in the design. Placement is also critical for distribution of heat dissipation on the die. An uneven routing profile can badly affect the reliability and timing closures.

The size of the placement problem can be very large even for small circuits. The solution space is huge as shown in figure 1. E.g. consider a circuit with only 10 cells which are to be placed in 10 locations can have roughly 3.5 million solutions and coming up with a best placement solution can be very tedious task. In general case there will be circuits with millions of cells which will have humongous number of solutions. In problem of such large complexity we can try to find the good and acceptable (not necessarily perfect) solution. For this purpose fast search algorithms like Simulated Annealing is widely used.



Figure 1: Solution space

2.1 Algorithm details:

Simulated annealing is a global optimization technique in which the problem of interest is treated to be analogous to a real-world problem of obtaining a perfectly homogeneous crystal at global minimum energy. A perfect or close to perfect crystal can be obtained using a process called annealing. In this process, a solid-state material is heated to a high temperature at which it reaches an amorphous liquid state. Then the material is allowed to cool down with a very slow and scheduled rate. If the initial temperature is high enough to make sure that, the atoms in the material are at “random” state and thermal equilibrium is reached at each step while cooling down, then the atoms will arrange themselves in a pattern that resembles the perfect crystal at global minimum energy.

The states of different energy E are created by perturbing the atoms multiple times at each temperature step. If the energy of the generated state is lower than the previously accepted state then it is accepted. If the energy of the generated state is more than the energy of the previously accepted state then this state is accepted with a Boltzmann probability. The Boltzmann probability is given by $e^{-\Delta E/BT}$ where, ΔE is the deviation of the energy of current state from the energy of the previously accepted state, T is the current temperature, and B is the Boltzmann constant. It is clear from the probability function that, the probability of higher energy state being accepted is high at high temperature value and at lower temperature value this probability is low, i.e. at lower temperature only those states with energy close to or less than previously accepted state will be accepted. Accepting a bad solution (higher energy state) at high temperature enables the ability of 'hill climbing', this concept is explained in brief in the next section.

2.1.1 Hill Climbing:

The solution space shown in figure 1 shows an instance where the current solution is in the local minima. The bad implementation of the algorithm will lead to get stuck in this local minima. As mentioned before the Boltzmann probability function allows to accept few worse states. This enables the system to come out of the local minima moving towards the intended global minima.

2.1.2 General Algorithm:

```

Begin
  S: Initial random solution  $S_0$ 
  T: Initial temperature  $T_0$ 
  P: probability of acceptance
  While (T>0.1)
    Begin
      While (no equilibrium)
        Begin
          S'=neighboring solution to S;
           $\Delta E = \text{Cost}(S') - \text{Cost}(S)$ ;
           $P = \min(1, e^{(-\Delta E/T)})$ ;
          If (random(0,1)<=P)
            S = S';
          End
        Update T;
      End
    End
  End

```

2.1.3 Placement problem:

It deals with the allocation of geometrical locations to the cells in a circuit with complex interconnections such that the total wire length in the network is minimized. This problem can be addressed using simulated annealing algorithm considering following analogy:

State: The current legal placement of cells.

Energy: The cost of total wire length for current placement.

Temperature: A control parameter in Boltzmann probability function.

Perturbation: Generation of new placement by swapping or moving cells.

2.2 Implementation Details:

The Simulated annealing algorithm discussed above was implemented in C++ and tested on the IBM benchmarks. The details of implementation are discussed below.

2.2.1 Data Structures:

The data structures used in the implementation of simulated annealing were very similar to FM method discussed earlier. Two classes were used, net class and node class, to store the information of nets and nodes.


1. Node class:

Table 1: Node class

X	Y	Netlist
Stores the current X co-ordinate of the cell	Stores the current Y co-ordinate of the cell	Stores a list of the nets connected to the cell.

Table 1 shows the members of the class 'node'. All cell related data from the node class can be accessed using an object of that cell. The objects are stored in another construct i.e. map named as 'cell'. The map stores the cell no. as a key and value is the object of the node class corresponding to that cell. The structure of map 'cell' is shown in table 2 with an example to illustrate how this map will be accessed for node 'cell2'

Table 2: Structure of 'cell' map



Cell no (key)	Object of class node (Value)		
0	Object of cell class corresponding to cell 0		
1	Object of cell class corresponding to cell 1		
2	X	Y	Netlist
3	Object of cell class corresponding to cell 3		
...	.		
n	Object of cell class corresponding to cell n		

2. Net Class:

Table 3: Net class

X coord	Y coord	Cell list	Cell count	Length
X co-ordinate map.	Y co-ordinate map	Stores a list of the cells connected to the net.	Stores the number of cells connected to the net.	Stores the length of the net.

The table 3 shows the members of class net. All net related data from the net class can be accessed using an object of that net. The objects are stored in another construct i.e. map named as 'net_map'.

The map stores net no. as a key and value is the object of the net class corresponding to that net. The structure of net_map is shown in table 4 with an example to illustrate how this map will be accessed for net 2.

Table 4: Structure of map net

Net no (key)	Object of class net (Value)				
0	Object of cell class corresponding to net 0				
1	Object of cell class corresponding to net 1				
2	X coord	Y coord	Cell list	Cell count	Length
3	Object of cell class corresponding to net 3				
.	.				
.	.				
n	Object of cell class corresponding to net n				

3. Details of class net:

Cell list: Cell list stores a list of cells which are connected to the net. The length of the net is recalculated whenever a cell which is moved or swapped is present in this list.

Cell count: This attribute stores an integer number which represents the number of cells connected to the net. This attribute is helpful to determine and remove nets with only one cell. (A net is said to be connected to only one cell, when it's a net between one cell and a pad)

Length: The length attribute stores the length of the net. To calculate this length, the cells which are placed on X and Y extremes of the net are first determined as shown in figure 2. A rectangle of width $W = (X_{max} - X_{min})$ and Height $H = (Y_{max} - Y_{min})$ is drawn through the extremes of the net. The half perimeter of this rectangle is considered to be the length of the net.

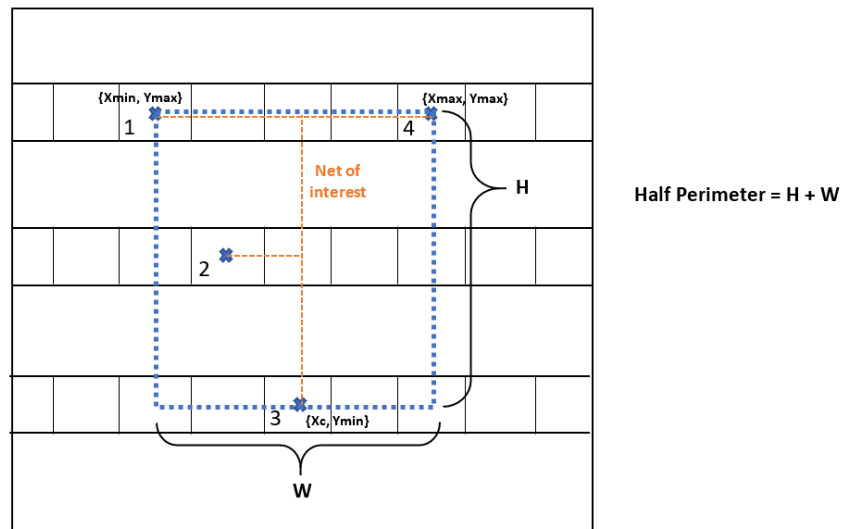


Figure 2: Wirelength calculation

X coord and Y coord: The X coord and Y coord shown in table 3 are the attributes constructed using maps. The keys for xcoord and ycoord maps are all the X co-ordinates and Y co-ordinates of the cells present in the cell list of the net respectively. The number of cells present on the respective co-ordinates of the cells are the values of respective keys. e.g. cell1, cell2, cell3, and cell4 are placed at (2,5), (3,3), (4,1), (6,5)

respectively then the key value (key,value) pairs for Xcoord are (2,1), (3,1),(4,1), and (6,1). Similarly, for Ycoord pairs are (1,1), (3,1), and (5,2). The purpose of using map data structure to store this information is that, the keys for maps are always stored in sorted order which makes determining Xmin, Xmax, Ymin, and Ymax even faster.

2.2.2 Implementation:

The implementation starts with reading a netlist file which stores the complete information of the circuit. First 5 lines of the netlist file provide numeric details of the circuit out of which 5th line represents the total number of cells present in the circuit. This number is very important to select the aspect ratio of the circuit. Having this number beforehand saves a great amount of time required to read the complete netlist and then determine the total number of cells.

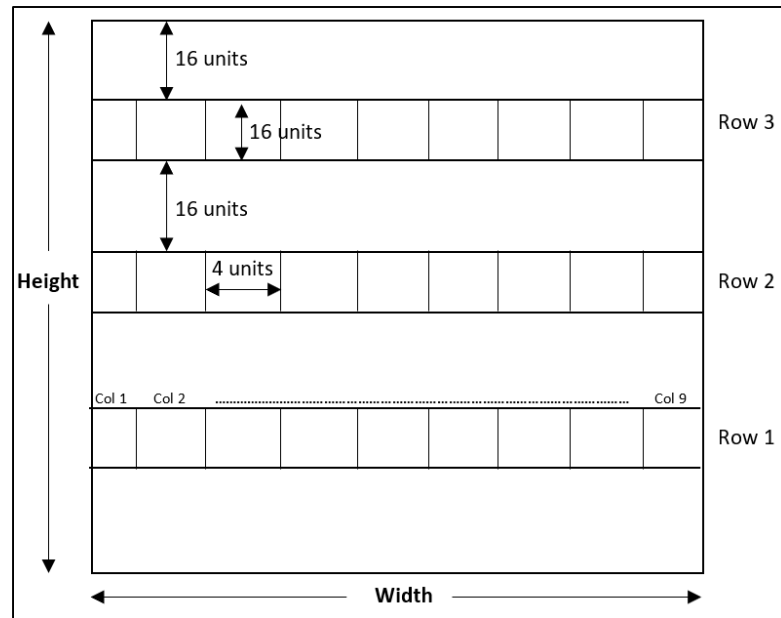


Figure 3: Dimension details of the die

As per the requirements for this assignment the aspect ratio was supposed to be closer to 1. The first task was to read the 5th line in the netlist file and select an aspect ratio which is close to 1. We developed a small formula, shown below, to determine the number of rows and columns required to get an aspect ratio roughly equal to 1.

Let's assume,
 tot = total number of cells
 col = total number of columns for the die
 row = total number of legal rows for the die.

In figure 3, legal rows are the rows in which cells are actually placed and other rows are routing channels. Here there are 3 legal rows, and 9 columns. Each cell is of height 16 units and width 4 units. From the geometry in the figure 3 we can also say,
 Total number of rows = $(2 * row) + 1$
 Height = total number of rows * 16
 Width = total number of columns * 4

$$\frac{Height}{Width} = \frac{((2 * row) + 1) * 16}{(col) * 4} = 1 \quad //desired aspect ratio$$

Also,

$$tot = row * col$$

Using these two equations, a quadratic equation is generated,

$$8(row)^2 + 4(row) - tot = 0$$

The roots of this equation give the number of rows and the determination of aspect ratio follows these steps:

```
Begin
  tot = The number on the 5th line in the netlist file;
  row = round((-4 + sqrt(16 + 32 * tot))/16);           //calculation of roots
  col = ceil(tot/row);
  AR  = ((8*row)+8)/col;
End
```

After determining the total number of rows and columns, the netlist file can be further read, and data structure can be updated simultaneously as shown,

```

/* Reading netlist file */

cost: total wirelength
length(n): length of net n

FOR each net 'n' = 1 ... N
    FOR each cell 'c' on the net n
        update cell list of net n;
        update net list of cell c;

        randomly place the cell c;
        update x_coord and y_coord maps for net n;
    END for c

    length(n) = netlength(n)*;    // (Xmax - Xmin)+(Ymax - Ymin)
    update cost;
END for n
* The function uses the Half Perimeter Wire Length formula discussed earlier.
The values for Xmin, Xmax, Ymin, and Ymax are determined using maps X_coord
and Y_coord.

```

At the end of this routine, an initial random placement is obtained, and a complete data structure is formed to start the Simulated Annealing process. This process goes as follows,

```

/* Simulated Annealing */

T = initial temperature
cost = C0

WHILE ( T>0.1 )
begin
    WHILE ( inner loop criteria satisfied )
    begin
        generate(); //Perturbation function
         $\Delta c$  = check_lengths(); //calculates new cost and  $\Delta c$ 
        if ( accept( $\Delta c, T$ ) )
            cost = new cost;
        else
            undo the move;
        end
        update T;
    end
end

```

This main loop calls three functions, **generate** (used to randomly pick two co-ordinates to swap/move cell/s), **check_lengths** (used to check the effect of move as well as to actually make the move), and **accept($\Delta c, T$)** (makes decision whether the move should be accepted or not).

The flow of generate() function is as follows,

```
/* Pick two locations */
First location: (x1,y1)
Second location: (x2,y2)

Begin
  DO
  {
    randomly select value for (x1,y1)
  }
  WHILE (condition for location_1)

  DO
  {
    randomly select value for (x2,y2)
  }
  WHILE (condition for location_2)
End
```

The condition for location_1 is that the first location must be occupied by a cell. It should not be an empty location. The condition for location_2 is that it can be an empty location, but it should not be same as location_1, if not empty. Also, it has to be within the range limited by a range limiter. Now, a swap/move is performed between cells from location_1 and location_2 depending upon location_2 occupied by a cell or not. The **check_lengths** function calculates the new cost in terms of wire length and change in cost (Δc). The routine for change length function:

```
/* Check_lengths function */

C1: cell from location_1
C2: cell from location_2

Begin
  new_cost = cost

  IF (location_1 is not empty)
  BEGIN
    FOR each net n in netlist of c1
      Determine x and y extremes after moving c1;
      calculate new_length for net n;
      new_cost = new_cost - old_length + new_length;
    END for netlist of c1
  END
  IF (location_2 is not empty)
  BEGIN
    FOR each net n in netlist of c2
      Determine x and y extremes after moving c2;
      calculate new length for net n;
      new_cost = new_cost - old_length + new_length;
    END for netlist of c2
  END
   $\Delta c$  = new_cost - cost;

  RETURN  $\Delta c$ ;
End
```

The **accept** (Δc , T) function decides if this change (Δc) is acceptable at current temperature T . This decision is based on Boltzmann's probability function discussed in section 3.1. The accept function returns TRUE if the change is acceptable and FALSE otherwise. The routine for **accept** (Δc , T) is as follows:

```
/* Accept function */
```

```
P: Probability of acceptance,  $P = e^{(-\Delta E/T)}$ 
```

```
r: random number between 0-1
```

```
Begin
```

```
  IF ( $\Delta c < 0$ )
```

```
    RETURN TRUE;
```

```
  ELSE IF ( $r < \min(1, P)$ )
```

```
    RETURN TRUE;*
```

```
  ELSE
```

```
    RETURN FALSE;
```

```
End
```

*Accepting $\Delta c > 0$ enables the ability of hill climbing to get out from local minima.

2.2.3 Internal loop criteria:

The generate, check_length, and accept functions performed until internal loop criteria is met. In our implementation this criterion is 3000 iterations or 2.5 times the total number of cells depending on whether the circuit size is small or large. A circuit with less than 500 cells is considered as a small circuit and a circuit with more than 500 cells is considered as big. These numbers are obtained experimentally and can be tuned to get different results.

2.2.4 Exit Criteria:

The program ends when the temperature becomes less than 0.1. The initial value for temperature is taken as 40000 which is obtained experimentally and can be tuned. The temperature is decreased gradually according to a cooling schedule. The program explores a predefined number of designs at each temperature step and ends with a solution close to the best acceptable solution.

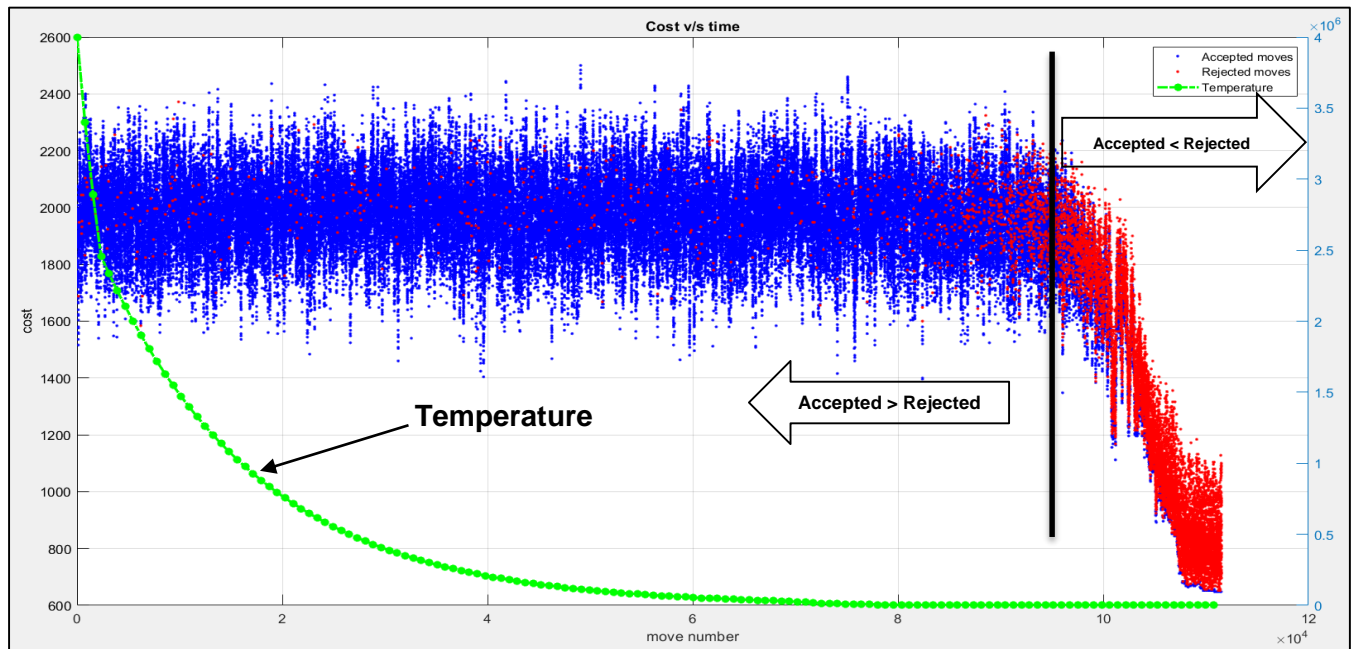


Figure 4: Cost v/s Number of moves

Figure 4 shows the graph of moves over the time and the costs associated with those moves. It also distinguishes the accepted and rejected moves with color coding. Accepted moves are shown in blue color and rejected moves are shown in red color. As shown in the figure 4, initially (at high temperature) all the moves are accepted and very less are rejected. As the temperature decreases, the rejection rate starts increasing and at one point the number of rejected moves becomes more than number of accepted moves. Keeping this behaviour in mind, the cooling schedule is defined for the temperature. This cooling routine is shown in table 5 below.

Table 5: cooling schedule

Condition	Cooling rate
Accepted > Rejected	-20%
Accepted < Rejected	-5%
$T < 4$	-2%

Initially the temperature parameter (T) is more dominant in the Boltzmann's probability function as compared to the change in cost (Δc). A lot of non-useful solutions get accepted which may or may not help for hill climbing. To avoid the acceptance of redundant solutions, initially the cooling rate is comparatively high (-20%). At a point where number of accepted solutions are less than the rejected ones, the temperature is low enough to make the change in cost significant. After this point the probability of accepting a non-useful solution is less, hence temperature steps are increased reducing the cooling rate down to -5%. When the temperature is really low (≈ 4), only useful solutions are accepted and almost all the non-useful solutions are rejected. The system is cooled very slowly by reducing the cooling rate to -2%. This increases the number of temperatures steps from this point. The cooling schedule shown in table 5 is obtained experimentally and can be tuned to get different results.

2.3 Testing and results

The above discussed algorithm was implemented in C++. To test the functionality and correctness of the code we were provided with 12 benchmark circuits of different sizes. Out of these, 10 were IBM benchmarks with relatively large sizes while the other two were very small. The table 6 below describes the nature of these benchmarks.

Table 6: Brief description of benchmarks

Benchmark name	Number of nodes	Number of nets
IBM01	12506	14111
IBM03	22853	27401
IBM04	27220	31970
IBM06	32332	34826
IBM08	51023	50513
IBM10	68685	75196
IBM12	70439	77240
IBM14	147088	152772
IBM16	182980	190048
IBM18	210341	201920
EACG21	14	17
EACG20	39	46

As can be seen from the table 6 above, the last two benchmarks are very small in size. These two benchmarks were very useful to do a quick test on the source code functionality whenever the changes were added to it. To test the functionality, we plotted the Cost v/s Execution number plot as shown in the figure 4.

The given benchmark circuits were simulated with the algorithm discussed above. The initial temperature was kept 40000. During simulation, we kept the record of initial cost and the cost at the end of each temperature step in a log file. The results and analysis for IBM01 benchmark are discussed in detail in the following section. The rest of the benchmarks follow exactly same procedure of analysis.

Table 7: Execution result for benchmark IBM01

Benchmark Name	Initial Wirelength Estimate	Final Wirelength Estimate	Percentage Improvement	Execution Time (Sec)
IBM 01	16182756	4061804	74.90041869	387
		3541312	78.11675589	666
		3329004	79.42869558	888
		3203036	80.2071044	1111

During the initial stages of the execution, the rate of improvement in result was fast. For example, we can see in table 7 that in IBM01 benchmark, ~75% improvement was achieved in first 387 seconds. As the program continued its execution, the improvement tends to slow down. But in the next 279 seconds, the improvement was only about ~3%. And the execution ends after 724 seconds with only ~5% of additional improvement. This is believed to be happening because initially there is a lot of scope for the improvement. But as the improvement starts increasing, the number of swaps/moves that reduces the total cost are reduced drastically. In essence, the improvement starts to saturate after some time. This behaviour can be seen graphically in the figure 5 below.

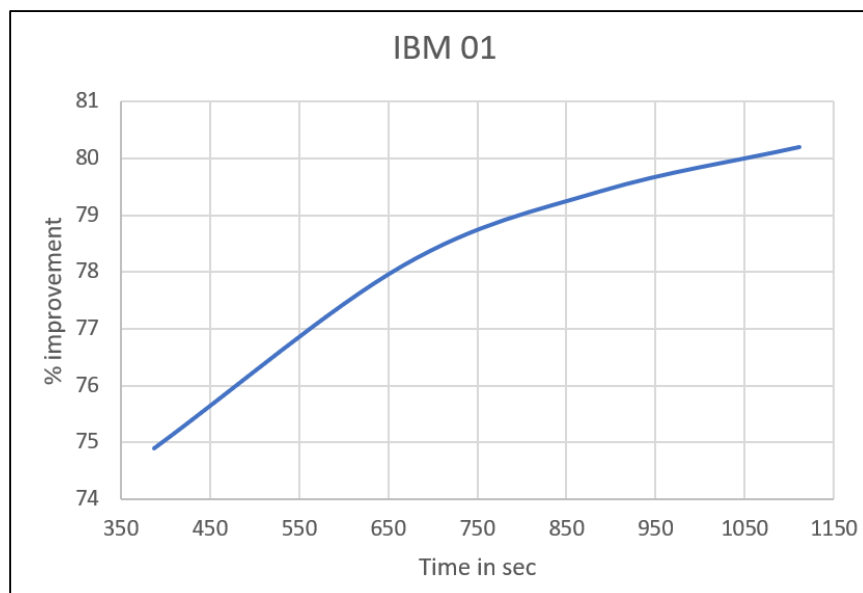


Figure 5: Improvement v/s Time for benchmark IBM01

The analytical results for rest of the benchmarks are shown in table 8:

Table 8: Analytical result for rest of the Benchmarks

Benchmark Name	Initial Wirelength Estimate	Final Wirelength Estimate	Percentage Improvement	Execution Time (Sec)
IBM 03	41533360	11410368	72.52722149	537
		10087176	75.71307498	1194
		9503548	77.11827793	1592
		9160496	77.9442453	1990
IBM 04	53015440	13430632	74.66656506	934
		11793392	77.75479747	1401
		11078168	79.1038837	1868
		10644228	79.92239996	2335
IBM 06	65634764	19506580	70.28010949	1146
		16855228	74.31966389	1719
		15913948	75.75378194	2292
		10644228	83.7826369	2867
IBM 08	120044976	33560984	72.04299162	1952
		29602004	75.34090556	2928
		28005292	76.67100038	3904
		26966244	77.5365493	4882
IBM 10	212195092	52779140	75.12706844	2104
		46138280	78.25666957	4209
		43042140	79.71577024	5612
		41164880	80.60045611	7016
IBM 12	222777376	54008088	75.75692426	1789
		47394548	78.72560093	3579
		44250292	80.13699021	4772
		42387444	80.97318284	5965
IBM 14	617612188	240887636	60.99694263	2222
		153876120	75.08531681	6666
		126941400	79.44642245	8888
		121120112	80.38896991	11113
IBM 16	896906428	220049428	75.46573186	5076
		192046448	78.58790594	10152
		178196372	80.13211117	13536
		169419632	81.1106681	16924
IBM 18	1020381488	267752340	73.75958471	5250
		236513700	76.82105146	12117
		221901632	78.25307156	16156
		212706544	79.15421374	20199
EACG 50	2212	544	75.40687161	0.42
EACG 21	336	112	66.66666667	0.55

2.4 Conclusion:

Simulated Annealing provides a probabilistic approach for optimizing the solution towards global minima in a large solution space. Therefore, the Simulated Annealing approach proves to be very effective algorithm for the problem like cell placement which has very huge solution space. The result of this approach depends on various parameters such as amount of exploration done initially as well as at every temperature step, starting temperature, and cooling schedule. Changes in these parameters may change the result and the execution time. The size of the network also impacts the results significantly. Therefore, it has been observed that having a relation between these parameters and size of the network leads to a better result.