VLSI Physical System Design
Implementation of Simulated Annealing-based Placement Algorithm (TimberWolf Algorithm)

Ritu Agarwal, Shashank Rao and Kewal Raul
Stony Brook University

Abstract— **Cell placement is an essential step in VLSI physical design automation - it is the portion of design flow that assigns exact locations for various circuit components within the chip's core area. This paper presents the implementation of a Simulated Annealing-based placement algorithm (TimberWolf algorithm) in Java, discussing the performance of code and its resulting output.**

I. INTRODUCTION

The increasing complexity of VLSI circuits led to the breaking of design processes into several steps, as well as the introduction of several semi-custom application-specific integrated circuits design methodologies such as standard cell and gate array design styles. Partitioning is used to divide the original circuit into sub-circuits, so that they can be dealt with separately [1]. Once the original circuit has been partitioned, cell placement techniques can be applied to the sub-circuits in order to construct the next step in VLSI physical design automation.

A. Overview Of Placement

Placement is the process of arranging the circuit components on a layout surface. During the placement steps, the VLSI circuit is seen as a set of rectangular blocks interconnected by signal nets. Placement consists of placing these blocks on a two dimensional surface such that no two blocks overlap, while optimizing the area of surface, the interconnection length between the blocks and the VLSI circuit performance [1].

A placement which involves a large amount of wiring space must necessarily include long wires and hence the total wirelength will be large. Hence, the total wirelength (ω) is a good parameter for measuring the layout area. Given a placement of cells or modules with ports(inputs, outputs, power and ground pins) on the boundaries, the dimensions of these cells(height, width, etc.) and a collection of nets(which are a set of ports wired together), the process of placement consists of finding suitable physical locations for each cell on the entire layout [1]. These locations selected should be subject to certain constraints such as avoidance of overlap of layout cells and that the cells must fit in a certain rectangular surface.

The main concepts underlying all placement techniques are: estimation of total wirelength using one of the many available estimation techniques such as - semi-perimeter method, minimum spanning tree, etc; minimization of the above calculated total wirelength; minimization of the net-cut size of placement cells obtained; minimization of the density of placement cells, thus reducing congestion; maximizing performance by imposing timing constraints on interconnects and paths of design; minimization of power dissipation of individual chips

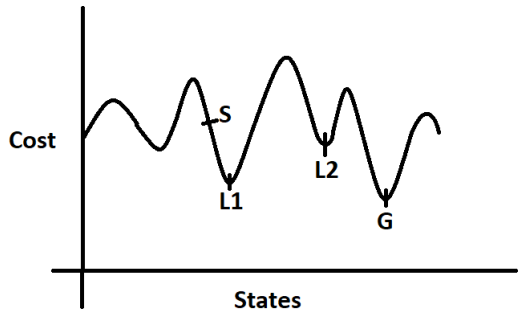

Fig. 1. Local and Global Minimas: Here S is the initial state; L1 and L2 are local minimas; G is the global minima

B. Overview Of Simulated-Annealing Algorithm

Simulated annealing is the most well developed method available for cell placement. It is an adaptive heuristic (in which some, or all parameters of the algorithm are changed during the execution) and belongs to the class of non-deterministic algorithms. The algorithm starts with a given state, and examines the local neighbourhood of the state for better solutions. A local neighbourhood of a state S is the set of all states which can be reached from S by making a small change to S [1]. The algorithm moves from the current state to a state in the local neighbourhood, if the latter has better cost. If all the local neighbours have inferior costs, then the algorithm is said to have converged to a local optimum. Further, the algorithm is assumed to be a minimization problem. The cost curve is convex, i.e., it has multiple minima [1]. Thus, the algorithm should not accept an inferior solution at the local minima, but should apply 'hill-climbing' methods so that it can climb out of the local minima to search for the global minima. Fig.1 shows such local and global minimas. Simulated annealing is such a hill-climbing algorithm.

1) The Simulated Annealing Algorithm: The core of the algorithm is the *Metropolis* procedure, which simulates the annealing process at a given temperature T. Metropolis also receives as input the current solution S which it improves through local search [1]. Metropolis procedure should also be provided with value M, which is the amount of time for which annealing must be applied at temperature T. The main simulated annealing procedure invokes the Metropolis procedure at various decreasing temperatures [1]. Temperature is initialized to a value T_0 at the beginning of the main procedure, and is slowly reduced in a geometric progression; and a parameter α is used to achieve this cooling. The amount of time spent in annealing at a temperature is gradually increased as temperature is lowered. This is done using the parameter $\beta > 1$ [1]. Another variable is used to keep track of the time being expended in each call to the Metropolis procedure. The main annealing procedure halts when this time variable exceeds a pre-defined allowed time.

The Metropolis procedure takes a solution set, the temperature T, and Metropolis time M as its parameters. It will then calculate the Cost function of the passed solution set as well as the cost function of a new set (which will be obtained by modifying the passed set on the basis of certain pre-defined parameters). If the difference in costs of new solution set and existing solution set is less than zero, then the new solution set will be accepted since it has a lower cost, meaning that it is an improvement over the existing passed solution set. If the difference is greater than zero, then a random number between 0 and 1 is selected. If this random number generated is less than the parameter ($e^{-\Delta h/T}$), then the new solution set is accepted, else the new solution set is rejected and a different new solution is once again calculated. In the parameter ($e^{-\Delta h/T}$), Δh is the difference in costs and T is the passed temperature. The random number is compared with the parameter ($e^{-\Delta h/T}$) so as to decide whether or not the hill-climbing solution of the algorithm should be applied and this criterion for accepting the new solution is known as Metropolis criterion. This is continued till Metropolis time M is reduced to 0.

2) The TimberWolf Algorithm for Placement: The Simulating Annealing algorithm modified for placement constitute the TimberWolf algorithm. The input data and parameters along with the total width of standard cells to be placed, enable the TimberWolf algorithm to compute the initial position and target lengths of the rows. Macro blocks will be placed next followed by the placement of pads. These macro blocks as well as pads will retain their initial positions and only the placement of standard-cells is optimized.

The described simulated annealing algorithm can be modified for cell placement by the following steps: The solution set will be that of a particular placement set; an initial placement solution set will be calculated and sent to the Metropolis procedure; within this procedure, a suitable *perturb* function will be used to generate a new placement configuration (cell assignments to slots); and then a suitable accept function will be defined.

The perturb function will perform one of the three functions [1]-

1) Move a single cell to a new location, say, to a different row.
2) Swap two cells
3) Mirror or rotate the cell about the x-axis

This selection between the three functions will occur randomly for each iteration within the Metropolis procedure. However, we will attempt mirroring only if the first two moves are not successful in reducing the cost.

The cost function that will be used by the TimberWolf algorithm is a sum of three components

$$Cost = Cost_1 + Cost_2 + Cost_3 \tag{1}$$

Here, $Cost_1$ is a measure of the total estimated wirelength. $Cost_1$ can be defined as:

$$Cost_1 = \alpha_1 * (\text{total wire length}) \tag{2}$$

where α_1 is a constant weight included in the equation for the total wirelength. Since the total wire length will be calculated as sum of the netlist length or as the semi-perimeter of the bounding box, this equation can be re-written as:

$$Cost_1 = \alpha_1 * \sum_{i \in nets} (\text{length of each net}) \tag{3}$$

$$Cost_1 = \alpha_1 * \sum (\text{Semi-perimeter of the bounding box}) \tag{4}$$

$Cost_2$ is a measure of the overlap between the swapped cells. $Cost_2$ can be defined as:

$$Cost_2 = \alpha_2 * (\text{penalty for cell overlap}) \tag{5}$$

where α_2 represents a constant weight included in the equation for cell overlap. If O_{ij} represents the area of overlap between two cells i and j, then

$$Cost_2 = \alpha_2 * \sum_{i \neq j} |O_{ij}|^2 \tag{6}$$

$Cost_3$ represents a penalty for the length of a row R exceeding or falling short of the expected length $\overline{L_R}$. $Cost_3$ can be defined as:

$$Cost_3 = \alpha_3 * (\text{penalty for uneven rows}) \tag{7}$$

$$Cost_3 = \alpha_3 * \sum_k |L_R - \overline{L_R}| \tag{8}$$

where α_3 is the weight for row unevenness; L_R is the actual length of row; $\overline{L_R}$ is the actual length of k; and k is the total number of rows.

The cooling schedule is represented by

$$T_{i+1} = \alpha(T_i) * T_i \tag{9}$$

where $\alpha(T)$ is the cooling rate parameter which is determined experimentally [1]. The annealing process is started at a very high initial temperature of $T_i = 4,000,000$. Initially, the temperature is reduced rapidly ($\alpha(T) \approx 0.8$). In the medium range, the temperature is reduced slowly ($\alpha(T) \approx 0.95$). Most processing is done at this range. In

the low temperature range, the temperature is again reduced rapidly ($\alpha(T) \approx 0.8$). The algorithm will be terminated when $T < 0.1$. [1]

At each temperature, a fixed number of moves will be attempted, i.e., the Metropolis time M will be set at a constant value. In our experiments, we will be setting this value to $\approx$ 120 moves at a given temperature.

II. RELATED WORK

In 1985, Carl Sechen and Alberto Sangiovanni-Vincentelli introduced the TimberWolf placement algorithm in a paper, titled "The TimberWolf Placement and Routing Package" [2]. This algorithm was an integrated set of placement and routing optimization programs. Their experimental results showed area savings over the then existing layout programs of industrial circuits from 15% to 62%.

This paper develops four basic optimization of the TimberWolf package - a standard-cell placement program, a standard cell global router program, a macro/custom cell placement program, and a generalized gate-array placement program. The initial section of the paper provides with the algorithm structure for probabilistic hill-climbing algorithm like simulated annealing. The algorithm structure provides an outer "stopping criterion" and an "inner loop criterion". This inner loop criterion contains a generate function and an accept function. We have implemented this inner loop criterion as the Metropolis procedure in our project.

The next section in the paper describes the standard cell placement optimization program for TimberWolf algorithm. The program was interfaced to the CIPAR standard cell placement package developed by American Microsystems, Inc. [2]. For the circuits tested, TimberWolf achieved total estimated wirelength reductions ranging from 45 to 66 percent and chip area reductions ranged from 15 to 57 percent. It was also found that the TimberWolf algorithm ran 12.2 times faster on IBM/UTS system in comparison to VMS and UNIX systems. The layouts of seven different circuits optimized using TimberWolf was compared to the manual layout of the same circuits. The projected manual layout was 10 percent larger than the layout produced with TimberWolf [2]. TimberWolf cell placement program was also interfaced with the Zymos placement and routing package(ZYPAR). TimberWolf reduced the total estimated wirelength by 44 percent but the chip area reduction was limited at 8 percent(this was mainly due to ZYPAR post-placement row-compaction routine). Intel Corp. and Hughes Aircraft Company also developed an interface to TimberWolf resulting in a chip area reduction of nearly 25 percent in both cases.

The following section in the paper describes the standard cell global router program as well as the results of the algorithm. In this case too, pad-limited area reduction achieved was limited to 11 percent while the core size(area inside the pad ring) area reduction was 22 percent. TimberWolf standard cell global router program was applied to four layout circuits. The next sections provide the TimberWolf macro and custom cell placement optimization program description. The results discussed were preliminary since these programs were being interfaced at the time of publishing of the paper as well as due to constraints deriving from the automatic insertion of components on the printed circuit board which were neglected by the TimberWolf algorithm. Gate array placement optimization programs are also discussed along with their results.

One of the other related works referred to during this project was "Optimization by simulated annealing", published by S.Kirkpatrick, C.Gelatt and M.Vecchi in 1983. This paper presented the central constructs in combinatorial optimization and in statistical mechanics and then develop the similarities between the two fields. The paper shows how the Metropolis algorithm for approximate numerical simulation of the behavior of a system at a finite temperature provides a natural tool for bringing the techniques of statistical mechanics to bear on optimization [3]. These concepts were then applied to partitioning, component placement and wiring of electrical systems as well as to optimizing the travelling salesman problem.

III. PROPOSED SOLUTION

The project describes the implementation of TimberWolf algorithm using Java. The overview of the files is that the written Java files will read the 5 input files, store the data in the appropriate data structure and then perform the TimberWolf algorithm on the data in order to optimize the placement structure. The core algorithm has been implemented on two main Java classes - NodeRowInfo.java and TimberWolf.java.

A. The NodeRowInfo Class

NodeRowInfo.java file contains the data structure that is used to hold the entire placement details of the cells. NodeRowInfo class has three subclasses - Node subclass, Row subclass & Boundaries subclass - along with other variables and methods. The following section describes each of the class variable and class method in detail.

1) The Node subclass: Node subclass holds the details of each node(or cell) which is present in the input file. It has the following class variables - "nodeName" (to hold the name of node like a1, p1, etc.) of String type, "width" (to hold the width of the node) of int type, "height" (to hold the height of the node) of int type, "area" (to hold the area occupied by the nodes/weights of nodes) of int type, "terminal" (which is a variable that holds whether the node is a pad or a cell) of int type, "xCordinate" (to hold the x-coordinate of the node being stored) of int type, "yCordinate" (to hold the y-coordinate of the node being stored) of int type, "xCenter" (to hold the x-coordinate of the center of the node) of int type, "yCenter" (to hold the y-coordinate of the center of the node) of int type, "orientation" (to hold the orientation of the node) of String type, "cellRowId" (to hold row number where the node will be placed) of int type, and "netList" (to hold the set of nets of which the node is a part of) of ArrayList of Integer type. The Node subclass also has the following class methods - a default constructor which initializes all the class variables (with String type variables initialized as

null, integer variables initialized as zero & ArrayList is allocated memory with the new operator); copyNode() to copy the node object created into anther similar Node object; setParametersFromNodes() method is used to set the class variables nodeName, width, height and terminal with the values read from the file; setParametersFromWts() method which is used to set the class variable area; setParametersFromPl() which is used to set the class variables xCordinate, yCordinate and orientation; setRowId(), which will set the cellRowId class variable; setNetList(), which will set the netList class variable; setCenter(), which will set the xCenter and yCenter class variables; and printNodeParameters() which will print the node details on the standard output.

2) The Row subclass: Row subclass holds the details of each of the row in entire placement area. It has the following class variables - "Id" (which holds the row number) of int type, "cordinate" (which holds the y-coordinate of each row on the entire placement area) of int type, "height" (which holds the height of the row; it has been observed that this row height is fixed at 16 for all 18 ibm files, but we have still considered it as a variable) of int type; variables "siteWidth", "siteSpacing", "siteOrient", "siteSymmetry", "siteRowOrigin", "numSites", overlap of int & string types which are row parameters read from the files but not used in our algorithm; and variable "cellList" (which holds the list of all the cells present in that particular row) of type ArrayList of String. The Row subclass also has the following class methods - a default constructor which initializes all the class variables (with String type variables initialized as null, integer variables initialized as zero & ArrayList is allocated memory with the new operator); copyNode() to copy the node object created into anther similar Node object; setId(), setRowParameter(), setCellList() methods are used to set the corresponding class variables; setCellListElement() and removeCellListElement() is used to set and delete the cells present in the row, i.e., add and delete elements from the ArrayList cellList variable defined; sortByX() method, which will sort the elements in the cellList variable according to the x-coordinate of the nodes; printRowParameters() which will print the row details on the standard output.

3) The Boundary subclass: Boundary subclass contains the boundaries of the entire placement area. It has four class variables - minXBound, maxXBound, minYBound & maxYBound all of int type. Each of these four variables taken separately will hold the edges of the entire placement area, i.e., (minXBound, minYBound) gives the coordinates of the bottom left corner of the placement area, (maxXBound, minYBound) gives the coordinates of the bottom right corner of the placement area, and so on. This subclass only contains a default constructor method to set initial value of four class variables to zero; copyBoundaries() method which will copy the values into an object copy; printBoundaries() method which will print the four boundaries.

4) The other variables and methods in NodeRowInfo class: NodeRowInfo also contains the following other variables and methods which are used to perform the algorithm.

We have used the following main data structures for storing the details - "nodeId" (which is a HashMap containing key-value pairs of nodeName-Node type, and which holds all nodes details present in the entire placement area); "rowId" (which is a HashMap containing key-value pairs of Id-Row, and which holds all rows details present in the entire placement area) and "netToCell" (which is a HashMap containing key-value pairs of net id and all the cells present in that particular net, and which holds all such nets over the entire placement area). These three hashmaps combined hold all the required values of the entire placement. Other variables used include - boundaries of type Boundaries(subclass defined above); numberOfCells of type int used to hold the count of cells which will be used in the placement algorithm, i.e., it does not include the count of macro cells and pads; totalWidthOfCells of type int which once again contains the width of all cells excluding the pads and macro blocks; xLimit of type int which is the x-coordinate from which standard cell placement will start, i.e., x-coordinate after the placement of the last macro cell; numNodes and numTerminals of type int, which contains the total number of nodes and number of terminals respectively; as well as xCordinateAfterLastPlacedCell & yCordinateAfterLastPlacedCell which will contain the x and y coordinates of the last standard cell placed during the algorithm. The class methods used are described next.

1) *printNodeRowInfo() function*: This function has calls to other print functions printNodeIdHashMap(), printRowIdHashMap(), printNetToCellHashMap() which will print the details present in the three hashmaps
2) *updateCenterOfEachCell() function*: This function is used to update xCenter and yCenter values in the Node object of the entire nodeId hashmap present; this is required after moving a cell since only xCordinate and yCordinate values of Node object are updated at that time
3) *createRowToCellMap() function*: This function is used during initial placement in order to create mappings between row and node object; updates cellRowId of all Node objects and cellList of all Row objects within the two hashmaps nodeId and rowId
4) *calculateBoundariesOfEntireRegion() function*: This function is used to update the boundaries of the entire placement area; updates the minimum and maximum x and y coordinates with the least and most values present in the pad coordinates
5) *placeMacroBlocks() function*: This function does the initial placement of the macro blocks; this function sets the xCordinate and yCordinate of all macro blocks (we have considered those blocks as macro blocks whose height is greater than 16) present in the nodeId hashmap
6) *initialPlacement() function*: This function is used for the initial placement of all the standard cells(height os cell is equal to 16); sets the x and y coordinates of node objects in nodeId hashmap; considers the placement to start at xLimit(x-coordinate+width of last placed

macro cell) and performs placement till the end of the row(maxXBound)

7) *forGraphicalRepresentation() function*: This function creates the text files which are used for graphical representation
8) *checkIfPadOrMacroblock(String, String) & checkIfPadOrMacroblock(String) functions*: These functions check if the given cells are pads or macros by comparing the terminal and height details stored in Node object
9) *swapCells(String, String function*: This function is used to swap the two cells passed to it; sets the xCordinate, yCordinate, xCenter, yCenter and cellRowId of the two cells in the nodeId hashmap as well as cellList of two cells in rowId hashmap
10) *moveCell(String) function*: This function moves the passed cell from its current location to another randomly selected location as well as displaces the cells present in the target row so as to avoid overlapping; sets xCordinate, yCordinate, cellRowId, and cellList variables in nodeId hashmap and updates cellList variables in rowId hashmap
11) *moveCell2(String) function*: This function moves the passed cell from its current location to another randomly selected location but does not displace the affected cells in the target row; does the same function as movecell() function without modifying the displaced cells
12) *sortCellListAccordingToX() function*: This function sorts the ArrayList cellList present in all the rows; internally calls sortByX() method of Row subclass for this function
13) *searchForEmptySpace(String, int) function*: This function is called from the moveCell() function internally; it searches for empty space in each row and inserts the cell at a free space if the width of the cell is smaller than the free space
14) *getRowIdFromY(int) function*: This function provides the row ID when provided with the y coordinate of the cell
15) *updateCellListAndRowCellIdEntry(String, int) function*: This function updates the cellList entry for the passed cell - removing it from the current row list and adding it to the new row list which is passed; also updates rowCellId of nodeId hashmap
16) *wireLengthCalc() function*: This function calculates the wirelength in the semi-perimeter of bounding box method; iterates through the cells present in the netList and finds out minimum and maximum values of x and y coordinates in order to calculate the semi-perimeter; is used during cost calculation
17) *overlapAreaCalc() function*: This function calculates the cell overlap area in each row; is used during cost calculation
18) *unevenRowsCalc() function*: This function is used to calculate the row unevenness among all rows; used during cost calculation
19) *copyMaps() function*: This function is used to make copies of the hashmaps; is used during copy of objects

B. The TimberWolf Class

This contains the main class which is run for executing the TimberWolf placement algorithm. The main class contains an object of NodeRowInfo class which hold all the details required for the algorithm to run. The flow of the algorithm from the main function is as follows:

1) Read all the five ibm files - the functions readNodesFile(), readWtsFile(), readPlFile(), readNetsFile(), readSclFile() are used for this functionality; the NodeRowInfo object N is populated with the values read from the file; pads are also placed into their final location on the placement area
2) Calculate the boundaries of the entire placement region - the function NodeRowInfo.calculateBoundariesOfEntireRegion() is used for this functionality; the minimum and maximum values of the x and y coordinates are calculated here using the pad values
3) Place the Macro blocks - the function NodeRowInfo.placeMacroBlocks() is used for this functionality; once the macro blocks are placed, the x and y coordinates for these nodes is set and this will be considered as the final placement for the macro cells
4) Perform the initial placement, update the center of each cell, create row-cell mappings and create the initial placement file for graphical representation - These functionalities are performed by NodeRowInfo.initialPlacement(), NodeRowInfo.updateCenterOfEachCell(), NodeRowInfo.createRowToCellMap() and NodeRowInfo.forGraphicalRepresentation() respectively; additionally, the initial placement is printed for clarity using NodeRowInfo.printNodeRowInfo() function
5) Run the TimberWolf Algorithm - the function runTimberWolfAlgo(NodeRowInfo) method is called to execute this functionality
6) In the main TimberWolf procedure, set the initial values and call the Metropolis procedure - in runTimberWolfAlgo() method, the initial values of temperature is set to 4000000, alpha is initially set to 0.8, and M is set at 120; a while loop is then run until cooling, i.e, as long as temperature is greater than 0.1; within this loop, the Metropolis procedure is called; temperature is reduced as a factor of alpha - alpha is initially set at 0.8, after temperature drops to one third of its initial value, it is set at 0.95, and again after temperature drops to one third of its value, alpha is set at 0.8
7) In the Metropolis procedure, perturb the initial placement, calculate the old and new cost and accept the new placement provided the cost has reduced - the perturb() and costFunction() is used for these functionalities within while loop(for M times)
8) The perturb() function randomly selects whether to move or swap randomly selected cells using the

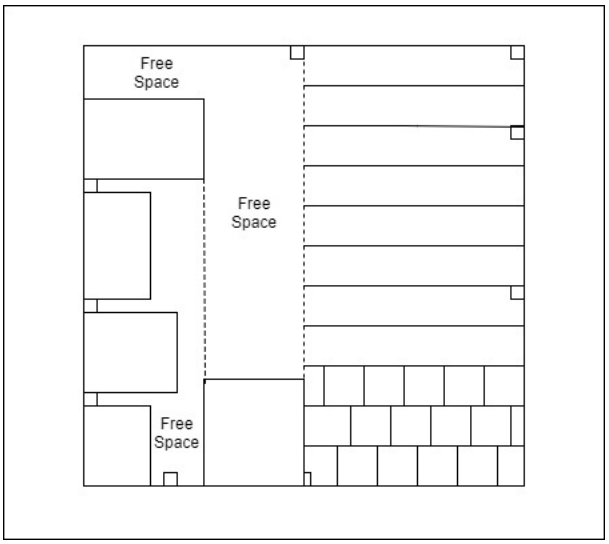

Fig. 2. Free unused space in our implementation

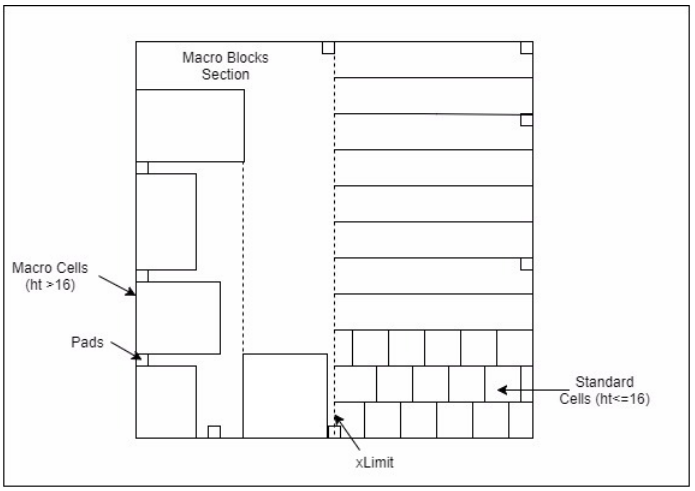

Fig. 3. Rough Estimate of Result through our placement algorithm

cellMove() [this function internally calls NodeRowInfo.moveCell2()] and cellSwap() [this function internally calls NodeRowInfo.swapCells()] respectively

9) The costFunction() calculates the wirelength, overlap area, and uneven row area using the functions calculateWireLength() [this function internally calls NodeRowInfo.wireLengthCalc()], calculateOverlapArea() [this function internally calls NodeRowInfo.overlapAreaCalc()] and calculateUnevenRowsPenalty() [this function internally calls NodeRowInfo.unevenRowsCalc()]; the weights for alpha1, alpha2 and alpha3 have been set at 1.5, 3 and 1 respectively for wirelength, overlapping area and uneven rows cost
10) In the Metropolis function, once the perturbation is completed and cost is calculated, depending on the cost difference, the new placement is accepted or rejected and once the M value (120 set here), is completed, the final accepted value is sent back to runTimberWolfAlgo(); this process is continued till temperature becomes less than 0.1

C. The Placement Class

This class contains the methods required for graphical representation of the initial and final placement. It uses JavaFX which can be directly included in the library in Java 8. We will be passing the x-coordinate, y-coordinate, width and height of each cell to the inbuilt rectangle function, which will generate the graphical representation.

IV. IMPLEMENTATION ISSUES

One of the main implementation issue faced was the creation of the data structure to hold the placement area details. The structure that we have created is pretty big and may cause memory inefficiency. One of the other issue that we faced during the project implementation was the placement of macro blocks. Our current method places the biggest macro blocks, followed by the next biggest macro blocks, and so on. However, this does not take into account the fact that some standard cells may be closer if random placement of macro blocks were made. Implementing random placement of macro blocks was a very big challenge, owing to the fact that the corresponding area had to be blocked off. Our resulting placement looks like in Fig.2. We also did not implement rotate cell functionality; we have only implemented cell move and swap functionalities during perturb of placement data structure. It also runs with different execution times on different machines. Also implementing graphical representation using JavaFX proved challenging due to scaling constraints and screen resolution constraints.

V. EXPERIMENTAL RESULTS AND DISCUSSIONS

Our initial placement algorithm first places the macro cells and then the standard cells are placed depending on the maximum x-coordinate of the macro cells. A rough estimate of the output is shown in Figure 3.

ISPD02 IBM-MS Mixed-size Placement Benchmarks consists of 18 ibm files, accommodating around 12,000 to 210,000 cells. The Figure 4 shows the execution times of the placement benchmarks run by us.

The Figures 5 and 6 show the initial and final graphical representations of the placement cells of ibm01 placement benchmark. Our graphical representation scaling is skewed when we go up the ibm benchmarks. While the correct values are being stored in the structure as well as created in the files, representing this was a challenge using JavaFX since it is dependent on the screen resolution as well as subject to other different constraints. Figure 7 shows the execution trace when completed for ibm01 benchmark. The reason for the relatively small execution time might probably be due to the way in which we have reduced the temperature.

We have observed that the initial temperature value that we have selected does effect the final placement result. On taking a smaller temperature, the execution time decreases, but the

Benchmark Files	Execution Time (s)
ibm01	379.87
ibm02	541.00
ibm03	702.56
ibm04	863.21
ibm05	1021.78
ibm06	1189.78
ibm07	1346.09
ibm08	1509.67
ibm09	1674.00
ibm10	1832.40
ibm11	1995.79
ibm12	2151.89
ibm13	2315.65
ibm14	2473.05
ibm15	2645.10
ibm16	2795.99
ibm17	2970.84
ibm18	3298.67

Fig. 4. Current execution times for placement benchmarks

number of iterations run will be fewer, resulting in a less optimized final cell placement. On selecting a higher initial temperature, execution time increases exponentially and the number of iterations run will increase causing better optimized final cell placement result. We have selected the same initial temperature (4000000) as in the TimberWolf algorithm paper presented by Carl Sechen and Alberto Sangiovanni-Vincentelli [2]. We have also changed alpha at one thirds of total temperature, i.e., at temperatures 2666666 and 1333333. Between temperature ranges 4000000 and 2666666 as well as 1333333 and 0.1, alpha is set at 0.8 resulting in faster reduction of temperature and fewer iterations. Between range 2666666 and 1333333, alpha is set at 0.95, resulting in slower reduction of temperature and greater iterations.

The value of cost weights alpha1, alpha2 and alpha3 also plays a role in optimizing the final placement. alpha1, which is the weight for wirelength cost has been selected as 1.5 in our program. Increasing this weight will increase the importance of wirelength cost, i.e., for same increase/decrease in wirelength, total cost of the function will increase with higher alpha1. Hence, we need to increase this cost in order to optimize the placement algorithm for greater emphasis on shorter wirelength. Cell overlap weight, alpha2, needs to be higher than the other two weights since cell overlap should not occur during placement. Keeping a high weight on alpha2 will ensure higher total cost of the placement option, resulting in greater chances on the placement option to be rejected if it has a high cell overlap as compared to previous placement option. Similarly, we have selected alpha3 to be 1, in order to signify lower optimization for uneven rows. Thus, the control of values of these three parameters results in achieving a multi-optimization algorithm. The value of alpha1, alpha2 and alpha3 decides whether the optimization is geared towards wirelength minimization, overlap area minimization or equal row length maximization. The values that we have selected hopefully optimizes the algorithm for all three requirements.

The perturbation function that implements cell swapping and cell move have also been optimized under certain constraints in our algorithm. We have also not implemented the cell rotate functionality during the perturbation. We hoped that since this functionality will be called with only a probablity of 10%, it will not be used a lot. Cell move functionality that we implemented also does not displace the surrounding cells in the target row. However, since we are calculating the overlap area, we hoped that the overlap of the cells in the target row will be accounted for.

VI. CONCLUSIONS

The implementation of TimberWolf algorithm presented in this paper has been proved to work correctly. We have run this algorithm on all the ibm files and shared the algorithm executions times. TimberWolf algorithm is a excellent option for cell placement since it has high optimizations as well as can achieve multi-optimization objectives.

REFERENCES

[1] ”VLSI Physical Design Automation - Theory And Practice” by Sadiq M. Sait and Habib Youssef

[2] ”The TimberWolf Placement and Routing Package” by Carl Sechen and Alberto Sangiovanni-Vincentelli, IEEE-JOURNAL OF SOLID-STATECIRCUITS, VOL. SC-20, NO. 2, APRIL 1985

[3] ”Optimization by simulated annealing” by S. Kirkpatrick, C. Gelatt and M. Vecchi, IBM Computer Science/Engineering Technology Watson Res. Center, Yofktown Heights, NY, Tech. Rep,, 1982

[4] ISPD02 IBM-MS Mixed-size Placement Benchmarks - Saurabh Adya , Igor Markov

[5] N.A Sherwani ”Placement,” in Algorithms for VLSI Physical Design Automation, 3rd ed. Norwell, K.A.P., 1999

VII. APPENDIX

A. *NodeRowInfo.java*

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;
import java.util.TreeMap;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class NodeRowInfo {

  public class Node{
    String nodeName;
    int width;
    int height;
    int area;
    int terminal;
    int xCordinate;
    int yCordinate;
    int xCenter;
    int yCenter;
    String orientation;
    int cellRowId;
    ArrayList<Integer> netList;

    public Node() {
      this.nodeName = "";
      this.width = 0;
      this.height = 0;
      this.area = 0;
      this.terminal = 0;
      this.xCordinate = 0;
      this.yCordinate = 0;
      this.xCenter = 0;
      this.yCenter = 0;
      this.orientation = "";
      this.cellRowId = 0;
      this.netList = new ArrayList<Integer
          >();
    }
    void copyNode(Node copyToNode)
    {
      copyToNode.nodeName=this.nodeName;
      copyToNode.width=this.width;
      copyToNode.height=this.height;
      copyToNode.area=this.area;
      copyToNode.terminal=this.terminal;
      copyToNode.xCordinate=this.xCordinate;
      copyToNode.yCordinate=this.yCordinate;
      copyToNode.xCenter=this.xCenter;
      copyToNode.yCenter=this.yCenter;
      copyToNode.orientation=this.
          orientation;
      copyToNode.cellRowId=this.cellRowId;
      for (Integer net : this.netList)
        copyToNode.netList.add(net);

    }

    void setParametersFromNodes(String N, int
        W, int H, int T) {
      this.nodeName = N;
      this.width = W;
      this.height = H;
      this.terminal = T;
    }

    void setParametersFromWts(int A) {
      this.area = A;
    }

    void setParametersFromPl(int X, int Y,
        String S) {
      this.xCordinate = X;
      this.yCordinate = Y;
      this.orientation = S;
    }

    Node setRowId(int R_id) {
      this.cellRowId = R_id;
      return this;
    }

    void setNetList(int N_id) {
      this.netList.add(N_id);
    }

    Node setCenter(int X, int Y) {
      this.xCenter = X;
      this.yCenter = Y;
      return this;
    }

    void printNodeParameters() {
      System.out.println("
          ----------------------------------------
          ");
      System.out.println("NodeName: "+this.
          nodeName);
      System.out.println("Width: "+this.
          width);
      System.out.println("Height: "+this.
          height);
      System.out.println("Area: "+this.area)
          ;
      System.out.println("Terminal: "+this.
          terminal);
      System.out.println("xCo-ordinate: "+
          this.xCordinate);
      System.out.println("yCo-rdinate: "+
          this.yCordinate);
      System.out.println("x/2: "+this.
          xCenter);
      System.out.println("y/2: "+this.
          yCenter);
      System.out.println("Orientation: "+
          this.orientation);
      System.out.println("CellRowId: "+this.
          cellRowId);

      Iterator<Integer> it = this.netList.
          iterator();
      System.out.print("Netlist: ");
      while(it.hasNext()) {
        System.out.print(it.next()+" ");
      }
```

```
      System.out.println("");
    }
  }

  public class Row {
    int Id;
    int cordinate;
    int height;
    int siteWidth;
    int siteSpacing;
    String siteOrient;
    String siteSymmetry;
    int siteRowOrigin;
    int numSites;
    int overlap;
    ArrayList<String> cellList;

    public Row() {
      this.Id = 0;
      this.cordinate = 0;
      this.height = 0;
      this.siteWidth = 0;
      this.siteSpacing = 0;
      this.siteOrient = "";
      this.siteSymmetry = "";
      this.siteRowOrigin = 0;
      this.numSites = 0;
      this.overlap = 0;
      this.cellList = new ArrayList<String
          >();
    }

    void copyRow(Row copyToRow)
    {
      copyToRow.Id=this.Id;
      copyToRow.cordinate=this.cordinate;
      copyToRow.height=this.height;
      copyToRow.siteWidth=this.siteWidth;
      copyToRow.siteSpacing=this.siteSpacing
          ;
      copyToRow.siteOrient=this.siteOrient;
      copyToRow.siteSymmetry=this.
          siteSymmetry;
      copyToRow.siteRowOrigin=this.
          siteRowOrigin;
      copyToRow.numSites=this.numSites;
      copyToRow.overlap=this.overlap;
      for(String e:this.cellList)
       copyToRow.cellList.add(e);
    }
    void setId(int I) {
      this.Id = I;
    }

    void setRowParameter(int C, int H, int SW
        , int SS, String SO, String SSym, int
         SRO, int NS) {
      this.cordinate = C;
      this.height = H;
      this.siteWidth = SW;
      this.siteSpacing = SS;
      this.siteOrient = SO;
      this.siteSymmetry = SSym;
      this.siteRowOrigin = SRO;
      this.numSites = NS;
    }

    void setCellList(ArrayList<String> cellId
        ) {
      this.cellList = cellId;
    }

    void setCellListElement (String cellId) {
      this.cellList.add(cellId);
    }

    String removeCellListElement(String
        cellId) {
      this.cellList.remove(cellId);
      return cellId;
    }

    ArrayList<String> sortByX() {
      int i=0;
      int x=0;
      String t;

      Map<Integer, String> sortX = new
          HashMap<Integer, String>();
      for(i=0; i<(this.cellList.size()); i
          ++) {
        t = this.cellList.get(i);
        x = nodeId.get(t).xCordinate;
        sortX.put(x, this.cellList.get(i));
      }

      ArrayList<String> list = new ArrayList
          <String>();
       Map<Integer, String> map = new
           TreeMap<Integer, String>(sortX);
       Iterator<Map.Entry<Integer, String>>
           it = map.entrySet().iterator();
       while(it.hasNext()) {
        Entry<Integer, String> next = it.
            next();
        list.add(next.getValue());
       }

      this.cellList = list;

      return this.cellList;
    }

//   void calculateRowOverlap() {
//     int xLast = 0, widthLast = 0;
//     String t = this.cellList.get(this.
    cellList.size()-1);
//     xLast = nodeId.get(t).xCordinate;
//     widthLast = nodeId.get(t).width;
//     int overlap = xLast + widthLast - (
    rowWidth+xLimit);
//
//     this.overlap = overlap;
//   }

    void printRowParameters() {
      System.out.println("
          ----------------------------------------
          ");
      System.out.println("RowId: "+this.Id);
      System.out.println("RowCordinate: "+
          this.cordinate);
      System.out.println("Height: "+this.
          height);
```

```
      System.out.println("SiteWidth: "+this.
          siteWidth);
      System.out.println("SiteSpacing: "+
          this.siteSpacing);
      System.out.println("SiteOrient: "+this
          .siteOrient);
      System.out.println("SiteSymmetry: "+
          this.siteSymmetry);
      System.out.println("SiteRowOrigin: "+
          this.siteRowOrigin);
      System.out.println("NumSites: "+this.
          numSites);
      System.out.println("Overlap: "+this.
          overlap);
      System.out.print("Cells in Row: ");

      Iterator<String> it = this.cellList.
          iterator();
      while(it.hasNext()) {
        System.out.print(it.next()+" ");
      }
      System.out.println("");
    }
  }

  public class Boundaries {
    int minXBound, maxXBound, minYBound,
        maxYBound;

    public Boundaries() {
      this.minXBound = 0;
      this.maxXBound = 0;
      this.minYBound = 0;
      this.maxYBound = 0;
    }

    void copyBoundaries(Boundaries
        copyToBoundary)
    {
      copyToBoundary.minXBound=this.
          minXBound;
      copyToBoundary.maxXBound=this.
          maxXBound;
      copyToBoundary.minYBound=this.
          minYBound;
      copyToBoundary.maxYBound=this.
          maxYBound;
    }
    void printBoundaries() {
      System.out.println("
          ------------------------------------
          ");
      System.out.println("Boundaries Of
          Region: ");
      System.out.println("Bottom-Left
          Boundary: ( "+this.minXBound+", "+
          this.minYBound+" )");
      System.out.println("Bottom-Right
          Boundary: ( "+this.maxXBound+", "+
          this.minYBound+" )");
      System.out.println("Top-Right Boundary
          : ( "+this.maxXBound+", "+this.
          maxYBound+" )");
      System.out.println("Top-Left Boundary:
           ( "+this.minXBound+", "+this.
          maxYBound+" )");
    }
  }


  public NodeRowInfo() {
    this.node = new Node();
    this.row = new Row();
    this.boundaries = new Boundaries();
//  this.maxNumberOfCellsPerRow = 0;
    this.numberOfCells = 0;
    this.totalWidthOfCells = 0;
    this.nodeId = new HashMap<String, Node>()
        ;
//  this.rowWidth = 0;
    this.xLimit = 0;
//  this.xLimit2 = 0;
//  this.yLimit = 0;
    this.rowId = new HashMap<Integer, Row>();
    this.netToCell = new HashMap<Integer,
        ArrayList<String>>();
    this.numNodes = 0;
    this.numTerminals = 0;
    this.xCordinateAfterLastPlacedCell = 0;
    this.yCordinateAfterLastPlacedCell = 0;
  }

  public void printNodeRowInfo() {
    printNodeIdHashMap();
    printRowIdHashMap();
    printNetToCellHashMap();
    this.boundaries.printBoundaries();
  }

  public void printNodeIdHashMap() {
    System.out.println("
        ****************************************
        ");
    System.out.println("For HashMap nodeId:")
        ;
    Iterator<Map.Entry<String, Node>> it1 =
        this.nodeId.entrySet().iterator();
    while(it1.hasNext()) {
      Entry<String, Node> next = it1.next();
      System.out.println("
          ----------------------------------------
          ");
      System.out.println("NodeId HashMap Key
          Value: "+next.getKey());
      next.getValue().printNodeParameters();
    }
  }

  public void printRowIdHashMap() {
    System.out.println("
        ****************************************
        ");
    System.out.println("For HashMap rowId:");
    Iterator<Map.Entry<Integer, Row>> it2 =
        this.rowId.entrySet().iterator();
    while(it2.hasNext()) {
      Entry<Integer, Row> next = it2.next();
      System.out.println("
          ----------------------------------------
          ");
      System.out.println("RowId HashMap Key
          Value: "+next.getKey());
      next.getValue().printRowParameters();
    }
  }

  public void printNetToCellHashMap() {
```

```
    System.out.println("
        ****************************************
        ");
    System.out.println("For HashMap netToCell
        :");
    Iterator<Map.Entry<Integer, ArrayList<
        String>>> it3 = this.netToCell.
        entrySet().iterator();
    while(it3.hasNext()) {
      Entry<Integer, ArrayList<String>> next
          = it3.next();
      System.out.println("NetToCell HashMap
          Key Value: "+next.getKey());
      System.out.println("
          ----------------------------------
          ");
      Iterator<String> it_3 = next.getValue
          ().iterator();
      while(it_3.hasNext()) {
        String next_3 = it_3.next();
        System.out.print(" "+next_3);
      }
      System.out.println("");
      System.out.println("
          ----------------------------------
          ");
    }
  }

  void updateCenterOfEachCell() {
    Iterator<Map.Entry<String, Node>> it =
        this.nodeId.entrySet().iterator();
    int xCenter = 0, yCenter = 0;
    while(it.hasNext()) {
      Entry<String, Node> next = it.next();
      xCenter = (next.getValue().xCordinate)
          + ((next.getValue().width)/2);
      yCenter = (next.getValue().yCordinate)
          + ((next.getValue().height)/2);
      Node newNode = new Node();
      newNode = next.getValue();
      next.setValue((newNode.setCenter(
          xCenter, yCenter)));
    }
  }

  void createRowToCellMap() {
    Iterator<Map.Entry<String, Node>>
        NodeIter = this.nodeId.entrySet().
        iterator();
    Iterator<Map.Entry<Integer, Row>> RowIter
        = this.rowId.entrySet().iterator();

    while(RowIter.hasNext()) {
      Entry<Integer, Row> RowNext = RowIter.
          next();
      while(NodeIter.hasNext()) {
        Entry<String, Node> NodeNext =
            NodeIter.next();
        if(NodeNext.getValue().height == 16)
            {
//          if( ((RowNext.getValue().cordinate
    ) <= (NodeNext.getValue().yCordinate)) &&
//              (((RowNext.getValue().
    cordinate)+(RowNext.getValue().height))
    >= (NodeNext.getValue().yCordinate)) ) {
          if((RowNext.getValue().cordinate)
              == (NodeNext.getValue().
                  yCordinate)) {

            Node newNode = new Node();
            newNode = NodeNext.getValue();
            newNode.setRowId(RowNext.getKey
                ().intValue());
            NodeNext.setValue(newNode);

            Row newRow = new Row();
            newRow = RowNext.getValue();
            ArrayList<String> newCellList =
                new ArrayList<String>();
            newCellList = newRow.cellList;
            newCellList.add(NodeNext.getKey
                ().toString());
            newRow.setCellList(newCellList);
//            Iterator<String> it = newRow.
    cellList.iterator();
//            while(it.hasNext()) {
//              System.out.println("id = "+
    newRow.Id+" "+it.next()+" ");
//            }
            RowNext.setValue(newRow);
          }
        }
      }
      NodeIter = this.nodeId.entrySet().
          iterator();
    }
  }

  void calculateBoundariesOfEntireRegion() {
    int xValue = 0, yValue = 0;
    Iterator<Map.Entry<String, Node>> it =
        this.nodeId.entrySet().iterator();
    while(it.hasNext()) {
      Entry<String, Node> next = it.next();
      if(next.getValue().terminal == 1) {
        xValue = next.getValue().xCordinate;
        yValue = next.getValue().yCordinate;
        if(xValue < this.boundaries.
            minXBound) {
          this.boundaries.minXBound = xValue
              ;
        }
        if(xValue > this.boundaries.
            maxXBound) {
          this.boundaries.maxXBound = xValue
              ;
        }
        if(yValue < this.boundaries.
            minYBound) {
          this.boundaries.minYBound = yValue
              ;
        }
        if(yValue > this.boundaries.
            maxYBound) {
          this.boundaries.maxYBound = yValue
              ;
        }
      }
    }
  }

  int placeMacroBlocks() {
    int xVal = this.boundaries.minXBound,
        yVal = 0;
    this.xLimit = this.boundaries.minXBound;
```

```
418
419     int rowHeight = 16; // This value is
            taken as constant since all 18 ibm
            files do not have any row height
            greater than this value
420
421     Set<Entry<String, Node>> entries = nodeId
            .entrySet();
422
423     Comparator<Entry<String, Node>>
            valueComparator = new Comparator<
            Entry<String, Node>>() {
424       @Override public int compare(Entry<
              String, Node> e1, Entry<String,
              Node> e2) {
425         String v1 = String.valueOf(e1.
                getValue().height);
426         String v2 = String.valueOf(e2.
                getValue().height);
427         return v1.compareTo(v2);
428         }
429       };
430
431     List<Entry<String, Node>>listOfEntries =
            new ArrayList<Entry<String, Node>>(
            entries);
432
433     Collections.sort(listOfEntries,
            valueComparator);
434
435     LinkedHashMap<String, Node> sortedByValue
             = new LinkedHashMap<String, Node>(
            listOfEntries.size());
436     for(Entry<String, Node> entry :
            listOfEntries){
437       sortedByValue.put(entry.getKey(),
              entry.getValue());
438       }
439
440     Iterator<Map.Entry<String, Node>> iter =
            sortedByValue.entrySet().iterator();
441     while(iter.hasNext()) {
442       Entry<String, Node> next = iter.next()
              ;
443       if((next.getValue().terminal == 0) &&
              (next.getValue().height > rowHeight
              )) {
444
445         Node newNode = new Node();
446         newNode = next.getValue();
447
448         if( (xVal + next.getValue().width) >
                this.xLimit) {
449           this.xLimit = xVal + next.getValue
                  ().width +1;
450         }
451
452         if((yVal + next.getValue().height) <
                this.boundaries.maxYBound) {
453           newNode.xCordinate = xVal;
454           newNode.yCordinate = yVal;
455           next.setValue(newNode);
456         }
457         else {
458           xVal = this.xLimit;
459           yVal = 0;
460           newNode.xCordinate = xVal;
461           newNode.yCordinate = yVal;
462           next.setValue(newNode);
463         }
464         yVal = yVal + next.getValue().height
                ;
465       }
466     }
467 //  this.xLimit2 = xVal;
468 //  this.yLimit = yVal;
469     return this.xLimit;
470   }
471
472   void initialPlacement() {
473     Iterator<Map.Entry<String, Node>>
          iterNode1 = this.nodeId.entrySet().
          iterator();
474     Iterator<Map.Entry<Integer, Row>> iterRow
          = this.rowId.entrySet().iterator();
475
476 //  int totalWidthOfCells = 0, rowWidth1 = 0,
      rowWidth2 = 0, count = 0, xCord = this.
     xLimit, yCord = 0;
477     int xCord = this.xLimit;
478     while(iterNode1.hasNext()) {
479       Entry<String, Node> next = iterNode1.
            next();
480
481       if((next.getValue().terminal == 0) &&
            (next.getValue().height == 16)) {
482         this.totalWidthOfCells += next.
              getValue().width + 1;
483         this.numberOfCells++;
484       }
485     }
486
487 //  int numOfRows = this.rowId.size();
488     int rowWidth = (this.boundaries.maxXBound
          -1) - (this.xLimit);
489 //  this.maxNumberOfCellsPerRow = ((this.
     numberOfCells)/(numOfRows)) ;
490     int occupiedRowWidth = 0;
491
492     Iterator<Map.Entry<String, Node>>
          iterNode2 = this.nodeId.entrySet().
          iterator();
493     Entry<Integer, Row> nextRow = iterRow.
          next();
494     while(iterNode2.hasNext()) {
495       Entry<String, Node> nextNode =
            iterNode2.next();
496       if( iterRow.hasNext() && (nextNode.
            getValue().terminal == 0)
497          && (nextNode.getValue().height <=
                nextRow.getValue().height)) {
498
499         occupiedRowWidth += nextNode.
              getValue().width;
500
501         if(occupiedRowWidth >= rowWidth) {
502           occupiedRowWidth = 0;
503           xCord = xLimit;
504           nextRow = iterRow.next();
505         }
506
507         Node newNode = new Node();
508         newNode = nextNode.getValue();
509         newNode.yCordinate = nextRow.
              getValue().cordinate;
510         newNode.xCordinate = xCord;
```

```
//        System.out.println(nextRow.getValue
    ().Id);
          newNode.setRowId(nextRow.getValue().
              Id);
          nextNode.setValue(newNode);
          xCord += (newNode.width)+1;
//        count++;
        }
      }
      this.xCordinateAfterLastPlacedCell =
          xCord;
      this.yCordinateAfterLastPlacedCell =
          nextRow.getValue().cordinate;
    }

    public void forGraphicalRepresentation(
        String filename) throws IOException {

      BufferedWriter bw = null;
      FileWriter fw = null;
      fw = new FileWriter(filename);
      bw = new BufferedWriter(fw);
      Iterator<Map.Entry<String, Node>>
          iterNode2 = this.nodeId.entrySet().
          iterator();
      String content1 = this.boundaries.
          minXBound+" "+this.boundaries.
          maxXBound+" "+this.boundaries.
          minYBound+" "+this.boundaries.
          maxYBound+"\n";
      bw.write(content1);
      while(iterNode2.hasNext()) {
        Entry<String, Node> nextNode =
            iterNode2.next();
        String content = nextNode.getValue().
            nodeName+" "+nextNode.getValue().
            xCordinate+" "+nextNode.getValue().
            yCordinate+
             " "+nextNode.getValue().width+" "+
                 nextNode.getValue().height+"\n
                 ";
        bw.write(content);
      }

      bw.close();
      fw.close();
    }

    public boolean checkIfPadOrMacroblock(
        String cell1, String cell2) {
      Node n1 = nodeId.get(cell1);
      Node n2 = nodeId.get(cell2);
      if(n1.terminal == 0 && n2.terminal == 0
          && n1.height == 16 && n2.height ==
          16) {
        return true;
      }
      return false;
    }

    public boolean checkIfPadOrMacroblock(
        String cell) {
      Node n = nodeId.get(cell);
      if(n.terminal == 0 && n.height == 16) {
        return true;
      }
      return false;
    }

    public void swapCells(String cell1, String
        cell2) {
      Node n1 = new Node();
      Node n2 = new Node();
      n1 = this.nodeId.get(cell1);
      n2 = this.nodeId.get(cell2);
      Row r1 = new Row();
      Row r2 = new Row();
      r1 = this.rowId.get(n1.cellRowId);
      r2 = this.rowId.get(n2.cellRowId);
      int r1key = r1.Id;
      int r2key = r2.Id;

      System.out.println("Swapping cell "+cell1
          +" of row "+n1.cellRowId+" with cell
          "+cell2+" of row "+n2.cellRowId);

      Node nTemp = new Node();

      nTemp.xCordinate = n1.xCordinate;
      nTemp.yCordinate = n1.yCordinate;
      nTemp.xCenter = n1.xCenter;
      nTemp.yCenter = n1.yCenter;
      nTemp.cellRowId = n1.cellRowId;

      n1.xCordinate = n2.xCordinate;
      n1.yCordinate = n2.yCordinate;
      n1.xCenter = n2.xCenter;
      n1.yCenter = n2.yCenter;
      n1.cellRowId = n2.cellRowId;

      n2.xCordinate = nTemp.xCordinate;
      n2.yCordinate = nTemp.yCordinate;
      n2.xCenter = nTemp.xCenter;
      n2.yCenter = nTemp.yCenter;
      n2.cellRowId = nTemp.cellRowId;

      this.nodeId.put(cell1, n1);
      this.nodeId.put(cell2, n2);

      String s1 = r1.removeCellListElement(
          cell1);
      String s2 = r2.removeCellListElement(
          cell2);
      r1.setCellListElement(s2);
      r2.setCellListElement(s1);

      this.rowId.put(r1key, r1);
      this.rowId.put(r2key, r2);

      sortCellListAccordingToX();
    }

    public void moveCell(String cell) {

      int min = 1, max = rowId.size();
      int randomRowSelected = (int) (Math.
          random() * (max-min) + min);
      min = this.xLimit;
      max = this.boundaries.maxXBound - 1;
      int randomXcordinateSelected = (int)(Math
          .random() * (max - min) + min);

      Node moveNode = new Node();
      moveNode = this.nodeId.get(cell);
      int prevRow = moveNode.cellRowId;
      int prevX = moveNode.xCordinate;
```

```
//   int cellWidth = moveNode.width;

     while(randomXcordinateSelected+moveNode.
         width > this.boundaries.maxXBound-1)
         {
       randomXcordinateSelected = (int)(Math.
           random() * (max - min) + min);
     }

     Row destinationRow = new Row();
     destinationRow = rowId.get(
         randomRowSelected);

     ArrayList<String> cellsInRow =
         destinationRow.cellList;
     Iterator<String> cellIt = cellsInRow.
         iterator();

     int i = randomXcordinateSelected;
     int d = moveNode.width;

     Map<String, Integer>
         forUpdatingCellListAfterLoop = new
         HashMap<String, Integer>();
     while(cellIt.hasNext()) {
       String c = cellIt.next();
       if(c.equals(cell)) {
         continue;
       }
       Node n = new Node();
       n = this.nodeId.get(c);
       int x = 0;

       if((n.xCordinate < i) && (n.xCordinate
           +n.width < i)) {
         continue;
       }
       else if((n.xCordinate < i) && (n.
           xCordinate+n.width > i)) {
         x = i + d + 1;
         i = x;
         d = n.width;
         System.out.println("Displacing cell
             "+c+ " of row "+n.cellRowId+"
             and x-cordinate "+n.xCordinate+"
              to x-cordinate "+x);
       }
       else if((n.xCordinate > i) && (n.
           xCordinate < i+d)) {
         int diff = (i+d)-n.xCordinate;
         x = n.xCordinate + diff + 1;
         i = x;
         d = n.width;
         System.out.println("Displacing cell
             "+c+ " of row "+n.cellRowId+"
             and x-cordinate "+n.xCordinate+"
              to x-cordinate "+x);
       }
       else if((n.xCordinate >= i) && (n.
           xCordinate+n.width < i+d)){
         x = i + d + 1;
         i = x;
         d = n.width;
         System.out.println("Displacing cell
             "+c+ " of row "+n.cellRowId+"
             and x-cordinate "+n.xCordinate+"
              to x-cordinate "+x);
       }
       else if((n.xCordinate > i) && (n.
           xCordinate >= i+d)) {
         break;
       }

       if((x+d > this.boundaries.maxXBound-1)
           || (x > this.boundaries.maxXBound
          -1)) {
         x = this.
             xCordinateAfterLastPlacedCell;
         int y = this.
             yCordinateAfterLastPlacedCell;

         if(this.
             xCordinateAfterLastPlacedCell+d
             <= this.boundaries.maxXBound-1)
             {
           this.xCordinateAfterLastPlacedCell
               += d;
           n.xCordinate = x;
           n.yCordinate = y;
           int newRowId = getRowIdFromY(n.
               yCordinate);
           String updNode = n.nodeName;
           forUpdatingCellListAfterLoop.put(
               updNode, newRowId);
//           updateCellListAndRowCellIdEntry(
    updNode, newRowId);
           this.nodeId.put(c, n);
         }
         else {
           this.xCordinateAfterLastPlacedCell
               = this.xLimit;
           if(this.
               yCordinateAfterLastPlacedCell
               < this.boundaries.maxYBound-1)
               {
             this.
                 yCordinateAfterLastPlacedCell
                  += 16;
             n.xCordinate = this.
                 xCordinateAfterLastPlacedCell
                 ;
             n.yCordinate = this.
                 yCordinateAfterLastPlacedCell
                 ;
             int newRowId = getRowIdFromY(n.
                 yCordinate);
             String updNode = n.nodeName;
             forUpdatingCellListAfterLoop.put
                 (updNode, newRowId);
//             updateCellListAndRowCellIdEntry(
    updNode, newRowId);
             this.nodeId.put(c, n);
           }
           else {
             searchForEmptySpace(c, d);
           }
         }
       }
       else if((x+d <= this.boundaries.
           maxXBound-1)) {
         n.xCordinate = x;
         this.nodeId.put(c, n);
       }
     }
```

```
    for(Map.Entry<String, Integer> it:
        forUpdatingCellListAfterLoop.entrySet
        ()) {
      updateCellListAndRowCellIdEntry(it.
          getKey().toString(), it.getValue().
          intValue());
    }

//   int destRowId = destinationRow.Id;

    moveNode.yCordinate = destinationRow.
        cordinate;
    moveNode.xCordinate =
        randomXcordinateSelected;
//   moveNode.cellRowId = destinationRow.Id;
    this.nodeId.put(cell, moveNode);

    updateCellListAndRowCellIdEntry(cell,
        destinationRow.Id);

//   ArrayList<String> c1 = destinationRow.
    cellList;
//
//   Row r = this.rowId.get(prevRow);
//   r.removeCellListElement(cell);
//   this.rowId.put(prevRow, r);
//
//   destinationRow.setCellListElement(cell);
//   this.rowId.put(destinationRow.Id,
    destinationRow);

    System.out.println("Moving "+cell+" from
        row "+prevRow+" at position "+prevX+"
         to row "+destinationRow.Id+" at
        position "+randomXcordinateSelected);
    sortCellListAccordingToX();
//   this.printNodeRowInfo();
  }

  void moveCell2(String cell) {
    int min = 1, max = rowId.size();
    int randomRowSelected = (int) (Math.
        random() * (max-min) + min);
    min = this.xLimit;
    max = this.boundaries.maxXBound - 1;
    int randomXcordinateSelected = (int)(Math
        .random() * (max - min) + min);

    Node moveNode = new Node();
    moveNode = this.nodeId.get(cell);
    int prevRow = moveNode.cellRowId;
    int prevX = moveNode.xCordinate;
//   int cellWidth = moveNode.width;

    while(randomXcordinateSelected+moveNode.
        width > this.boundaries.maxXBound-1)
        {
      randomXcordinateSelected = (int)(Math.
          random() * (max - min) + min);
    }

    Row destinationRow = new Row();
    destinationRow = rowId.get(
        randomRowSelected);

    moveNode.yCordinate = destinationRow.
        cordinate;
    moveNode.xCordinate =
        randomXcordinateSelected;
//   moveNode.cellRowId = destinationRow.Id;
    this.nodeId.put(cell, moveNode);

    updateCellListAndRowCellIdEntry(cell,
        destinationRow.Id);

//   ArrayList<String> c1 = destinationRow.
    cellList;
//
//   Row r = this.rowId.get(prevRow);
//   r.removeCellListElement(cell);
//   this.rowId.put(prevRow, r);
//
//   destinationRow.setCellListElement(cell);
//   this.rowId.put(destinationRow.Id,
    destinationRow);

    System.out.println("Moving "+cell+" from
        row "+prevRow+" at x-coordinate "+
        prevX+" to row "+destinationRow.Id+"
        at x-coordinate "+
        randomXcordinateSelected);
    sortCellListAccordingToX();
  }

  void sortCellListAccordingToX() {
    for(Map.Entry<Integer, Row> it : this.
        rowId.entrySet()) {
      ArrayList<String> sortedList = it.
          getValue().sortByX();
      int rowid = it.getValue().Id;
      Row r = it.getValue();
      r.setCellList(sortedList);
      this.rowId.put(rowid, r);
    }
  }

  void searchForEmptySpace(String cell, int
      width) {
    Iterator<Map.Entry<Integer, Row>> rowIter
        = this.rowId.entrySet().iterator();
    String c1 = null, c2 = null;
    sortCellListAccordingToX();
    Map<String, Integer>
        forUpdatingCellListAfterLoop = new
        HashMap<String, Integer>();
    while(rowIter.hasNext()) {
      int flag = 0;
      Entry<Integer, Row> nextRow = rowIter.
          next();
      ArrayList<String> cL =nextRow.getValue
          ().cellList;
      Iterator<String> it = cL.iterator();
      while(it.hasNext()) {
        c1 = it.next();
        if(flag == 0) {
          c2 = it.next();
          flag = 1;
        }
        Node n1 = this.nodeId.get(c1);
        Node n2 = this.nodeId.get(c2);
        int lower = n1.xCordinate + n1.width
            ;
        int upper = n2.xCordinate;
        if(upper-lower >= width) {
          Node n = this.nodeId.get(cell);
```

```
            n.xCordinate = lower+1;
            n.yCordinate = nextRow.getValue().
                cordinate;
            forUpdatingCellListAfterLoop.put(
                cell, n.yCordinate);
            this.nodeId.put(cell, n);
          }
          c2 = c1;
        }
      }
      for(Map.Entry<String, Integer> it:
          forUpdatingCellListAfterLoop.entrySet
          ()) {
        updateCellListAndRowCellIdEntry(it.
            getKey().toString(), it.getValue().
            intValue());
      }
    }

    int getRowIdFromY(int y) {
      int row = 0;
      for(Map.Entry<Integer, Row> it: this.
          rowId.entrySet()) {
        if(it.getValue().cordinate == y) {
          row = it.getKey().intValue();
        }
      }
      return row;
    }

    void updateCellListAndRowCellIdEntry(String
        addRemElem, int addRowTo) {
      Node n = this.nodeId.get(addRemElem);
      int oldRow = n.cellRowId;
      n.cellRowId = addRowTo;
      this.nodeId.put(addRemElem, n);

      Row rRem = this.rowId.get(oldRow);
      rRem.removeCellListElement(addRemElem);
      this.rowId.put(oldRow, rRem);

      Row rAdd = this.rowId.get(addRowTo);
      rAdd.setCellListElement(addRemElem);
      this.rowId.put(addRowTo, rAdd);

//    sortCellListAccordingToX();
    }

// void test() {
//   Row r1 = this.rowId.get(1);
//   r1.removeCellListElement("a2");
//   r1.removeCellListElement("a5");
//   r1.setCellListElement("a5");
//   r1.setCellListElement("a2");
//   this.rowId.put(1, r1);
// }

    int wireLengthCalc() {
      int wireLength = 0;
      int xCord = 0, yCord = 0;

      for(Map.Entry<Integer, ArrayList<String>>
          netIter : this.netToCell.entrySet())
          {
        int minXBound = 500000, minYBound =
            500000, maxXBound = -500000,
            maxYBound = -500000;
        ArrayList<String> cellList = netIter.
            getValue();
        Iterator<String> cellListIter =
            cellList.iterator();
        while(cellListIter.hasNext()) {
          String cell = cellListIter.next();
          xCord = this.nodeId.get(cell).
              xCenter;
          yCord = this.nodeId.get(cell).
              yCenter;

          if(xCord < minXBound) {
            minXBound = xCord;
          }
          if(xCord > maxXBound) {
            maxXBound = xCord;
          }
          if(yCord < minYBound) {
            minYBound = yCord;
          }
          if(yCord > maxYBound) {
            maxYBound = yCord;
          }
        }

        wireLength += (Math.abs(maxXBound -
            minXBound)) + (Math.abs(maxYBound -
            minYBound));
      }

      return wireLength;
    }

    int overlapAreaCalc() {
      int totalOverlapArea = 0;
      sortCellListAccordingToX();
      String c1 = null, c2 = null;
      int xCord1 = 0, xCord2 = 0, xWidth1 = 0,
          xWidth2 = 0;
      for(Map.Entry<Integer, Row> rowIter :
          this.rowId.entrySet()) {
        int rowOverlapArea = 0, flag = 0;
        ArrayList<String> rowCellList =
            rowIter.getValue().cellList;
        Iterator<String> rowCellListIter =
            rowCellList.iterator();
        while(rowCellListIter.hasNext()) {
          if(flag == 0) {
            c1 = rowCellListIter.next();
            if(rowCellListIter.hasNext()) {
              c2 = rowCellListIter.next();
            }
            else {
              break;
            }
            flag = 1;
          }
          else if (flag == 1 || flag == 2) {
            c1 = c2;
            if(rowCellListIter.hasNext()) {
              c2 = rowCellListIter.next();
            }
            else {
              break;
            }
          }
          else if (flag == 3) {
            if(rowCellListIter.hasNext()) {
```

```
            c2 = rowCellListIter.next();
          }
          else {
            break;
          }
          flag = 1;
        }
        xCord1 = this.nodeId.get(c1).
            xCordinate;
        xWidth1 = this.nodeId.get(c1).width;
        xCord2 = this.nodeId.get(c2).
            xCordinate;
        xWidth2 = this.nodeId.get(c2).width;

        if((xCord1+xWidth1 >= xCord2) && (
            xCord2+xWidth2 >= xCord1+xWidth1
            )) {
          rowOverlapArea += (((xCord1+
              xWidth1) - xCord2)*16);
          flag = 2;
        }
        else if((xCord2 >= xCord1) && (
            xCord2+xWidth2 < xCord1+xWidth1)
            ) {
          rowOverlapArea += ((xWidth2)*16);
          flag = 3;
        }
      }
      totalOverlapArea += rowOverlapArea;
    }
    return totalOverlapArea;
  }

  float unevenRowsCalc() {
    float desiredLengthOfEachRow = (this.
        totalWidthOfCells)/(this.rowId.size()
        );
    float totalDeviationFromDesiredLength =
        0;
    for(Map.Entry<Integer, Row> rowIter :
        this.rowId.entrySet()) {
      float perRowDeviationFromDesiredLength
          = 0, perRowUsedLength = 0;
      ArrayList<String> cellList = rowIter.
          getValue().cellList;
      Iterator<String> cellListIter =
          cellList.iterator();
      while(cellListIter.hasNext()) {
        String cell = cellListIter.next();
        perRowUsedLength += this.nodeId.get(
            cell).width + 1;
      }
      perRowDeviationFromDesiredLength =
          Math.abs(desiredLengthOfEachRow -
          perRowUsedLength);
      totalDeviationFromDesiredLength +=
          perRowDeviationFromDesiredLength;
    }
    return totalDeviationFromDesiredLength;
  }

  void copyMaps(NodeRowInfo newN) {
    Map<Integer, Row> newRowId = new HashMap<
        Integer, Row>();
    Map<Integer, ArrayList<String>>
        newNetToCell = new HashMap<Integer,
        ArrayList<String>>();
    Map<String, Node> netNodeId = new HashMap
        <String, Node>();

    for(Map.Entry<Integer, Row> it : this.
        rowId.entrySet()) {
      newRowId.put(it.getKey().intValue(),
          it.getValue());
    }
    for(Map.Entry<Integer, Row> it : this.
        rowId.entrySet()) {
      newRowId.put(it.getKey().intValue(),
          it.getValue());
    }

    newNetToCell.putAll(this.netToCell);
    netNodeId.putAll(this.nodeId);
  }

  Node node;
  Row row;
  Boundaries boundaries;
  int numberOfCells;
  int totalWidthOfCells;
  Map<String, Node> nodeId;
  int xLimit;
  Map<Integer, Row> rowId;
  Map<Integer, ArrayList<String>> netToCell;
  int numNodes;
  int numTerminals;
  int xCordinateAfterLastPlacedCell;
  int yCordinateAfterLastPlacedCell;
}
```

B. *TimberWolf.java*

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.nio.charset.Charset;
import java.util.ArrayList;
import java.util.Map;

public class TimberWolf {

// NodeRowInfo N;
//
// public TimberWolf() {
//   N = new NodeRowInfo();
//
// }

  public void readNodesFile(String filename,
      NodeRowInfo N) {

    String line = null;
    int i=0, val=2;
    try {
        filename = filename + ".nodes";
        InputStream fis = new
            FileInputStream(filename);
         InputStreamReader isr = new
             InputStreamReader(fis, Charset.
             forName("UTF-8"));
```

```
        BufferedReader br = new
            BufferedReader(isr);
        while ((line = br.readLine()) !=
            null) {
          i++;
          if(i==6) {
            String[] words = line.split("\\s
                +");
            N.numNodes = Integer.parseInt(
                words[2]);
          }
          else if(i==7) {
            String[] words = line.split("\\s
                +");
            N.numTerminals = Integer.
                parseInt(words[2]);
          }
          else if(i>7) {
            String[] words = line.split("\\s
                +");
            int len = words.length;
            if(len>4) {
              if(words[4].equals("terminal"
                  )) {
                val = 1;
              }
            }
            else {
              val = 0;
            }

            N.node = N.new Node();
            N.node.setParametersFromNodes(
                words[1], Integer.parseInt(
                words[2]), Integer.parseInt(
                words[3]), val);
            N.nodeId.put(words[1], N.node);
          }
        }

        fis.close();
        isr.close();
        br.close();

    }catch(FileNotFoundException ex) {
        System.out.println(
              "Unable to open file "+
                  filename);
        }
        catch(IOException ex) {
           System.out.println(
              "Error reading file "+filename
                  );
        }
  }

  public void readWtsFile(String filename,
      NodeRowInfo N) {
    String line = null;
    int i=0;
    try {
        filename = filename + ".wts";
        InputStream fis = new
            FileInputStream(filename);
        InputStreamReader isr = new
            InputStreamReader(fis, Charset.
            forName("UTF-8"));
        BufferedReader br = new
            BufferedReader(isr);
        while ((line = br.readLine()) !=
            null) {
          i++;
          if(i>5) {
            String[] words = line.split("\\s
                +");
//            System.out.println(words[1]+"hi
    "+words[2]);
            (N.nodeId.get(words[1])).
                setParametersFromWts(Integer
                .parseInt(words[2]));
          }
        }

        fis.close();
        isr.close();
        br.close();

    }catch(FileNotFoundException ex) {
        System.out.println(
              "Unable to open file "+
                  filename);
        }
        catch(IOException ex) {
           System.out.println(
              "Error reading file "+filename
                  );
        }
  }

  public void readPlFile(String filename,
      NodeRowInfo N) {
    String line = null;
    int i=0;
    try {
        filename = filename + ".pl";
        InputStream fis = new
            FileInputStream(filename);
        InputStreamReader isr = new
            InputStreamReader(fis, Charset.
            forName("UTF-8"));
        BufferedReader br = new
            BufferedReader(isr);
        while ((line = br.readLine()) !=
            null) {
          i++;
          if(i>6) {
            String[] words = line.split("\\s
                +");
            (N.nodeId.get(words[0])).
                setParametersFromPl(Integer.
                parseInt(words[1]), Integer.
                parseInt(words[2]), words
                [4]);
          }
        }

        fis.close();
        isr.close();
        br.close();

    }catch(FileNotFoundException ex) {
        System.out.println(
              "Unable to open file "+
                  filename);
        }
```

```
        catch(IOException ex) {
           System.out.println(
              "Error reading file "+filename
                 );
        }
  }

  public void readNetsFile(String filename,
      NodeRowInfo N) {
    String line = null;
    int i=0, val=0;
    int NetId = 1;
    try {
        filename = filename + ".nets";
        InputStream fis = new
            FileInputStream(filename);
         InputStreamReader isr = new
             InputStreamReader(fis, Charset.
             forName("UTF-8"));
         BufferedReader br = new
             BufferedReader(isr);
         while ((line = br.readLine()) !=
             null) {
           i++;
           if(i>7) {
             String[] words = line.split("\\s
                 +");
             if(words[0].equals("NetDegree"))
                  {
               val = Integer.parseInt(words
                   [2]);
               ArrayList<String> wordsTemp =
                    new ArrayList<String>();
               for(int j=0; j<val; j++) {
                 line = br.readLine();
                 words = line.split("\\s+");
                 wordsTemp.add(words[1]);
                 (N.nodeId.get(words[1])).
                     setNetList(NetId);
               }
               N.netToCell.put(NetId,
                   wordsTemp);
             }
             NetId++;
           }
         }

         fis.close();
         isr.close();
         br.close();

    }catch(FileNotFoundException ex) {
        System.out.println(
              "Unable to open file "+
                  filename);
        }
        catch(IOException ex) {
           System.out.println(
              "Error reading file "+filename
                 );
        }
  }

  public void readSclFile(String filename,
      NodeRowInfo N) {
    String line = null;
    try {
        int i=0, j=0;
        int Id = 1;
        int cordinate = 0;
        int height = 0;
        int siteWidth = 0;
        int siteSpacing = 0;
        String siteOrient = "";
        String siteSymmetry = "";
        int siteRowOrigin = 0;
        int numSites = 0;
        filename = filename + ".scl";
        InputStream fis = new
            FileInputStream(filename);
         InputStreamReader isr = new
             InputStreamReader(fis, Charset.
             forName("UTF-8"));
         BufferedReader br = new
             BufferedReader(isr);
         while ((line = br.readLine()) !=
             null) {
           i++;
           if(i>8) {
             String[] words = line.split("\\s
                 +");
             j = i%9;
             if(j == 1)
             {
               cordinate = Integer.parseInt(
                   words[3]);
             }
             else if(j == 2)
             {
               height = Integer.parseInt(
                   words[3]);
             }
             else if(j == 3)
             {
               siteWidth = Integer.parseInt(
                   words[3]);
             }
             else if(j == 4)
             {
               siteSpacing = Integer.
                   parseInt(words[3]);
             }
             else if(j == 5)
             {
               siteOrient = words[3];
             }
             else if(j == 6)
             {
               siteSymmetry = words[3];
             }
             else if(j == 7)
             {
               siteRowOrigin = Integer.
                   parseInt(words[3]);
               numSites = Integer.parseInt(
                   words[6]);
             }
             else if(j == 8)
             {
               N.row = N.new Row();
               N.row.setId(Id);
               N.rowId.put(Id, N.row);
               (N.rowId.get(Id)).
                   setRowParameter(cordinate,
                    height, siteWidth,
                   siteSpacing, siteOrient,
```

```
                    siteSymmetry,
                    siteRowOrigin, numSites);
                Id++;
              }
            }
          }

          fis.close();
          isr.close();
          br.close();

      }catch(FileNotFoundException ex) {
          System.out.println(
                "Unable to open file "+
                    filename);
          }
          catch(IOException ex) {
             System.out.println(
                "Error reading file "+filename
                    );
          }
  }



  public NodeRowInfo runTimberWolfAlgo(
      NodeRowInfo T_N) {
    float temperature = 4000000;
    float time = 0;
    float alpha = (float) 0.8;
    int M = 120;

    System.out.println("Starting TimberWolf
        Algorithm...");
    while(temperature > 0.1) {

       System.out.println("Current
           Temperature = "+temperature+" and
           alpha = "+alpha);
       T_N = Metropolis(temperature, M, T_N);
       temperature = alpha*temperature;

       if(temperature < 4000000 && (
           temperature >= 2666667)) {
         alpha = (float) 0.8;
       }
       else if((temperature < 2666667) && (
           temperature >= 1333333)) {
         alpha = (float) 0.95;
       }
       else if((temperature < 1333333) && (
           temperature > 0.1)) {
         alpha = (float) 0.8;
       }

//     if(temperature < 4000000 && (
    temperature >= ((temperature/3)*2))) {
//       alpha = 0.8;
//     }
//     else if((temperature < ((temperature
    /3)*2)) && (temperature > (temperature/2)
    )) {
//       alpha = 0.95;
//     }
//     else if((temperature < (temperature/2)
    ) && (temperature > 0.1)) {
//       alpha = 0.8;
//     }

//     if(alpha <= 0.95) {
//       alpha = alpha + 0.01;
//     }
//     else if(alpha >= 0.80) {
//       alpha = alpha - 0.01;
//     }
    }
    return T_N;
  }

  public NodeRowInfo Metropolis(double T, int
      M, NodeRowInfo T_N) {

    System.out.println("Entering Metropolis
        Function...");
    while(M != 0) {
      NodeRowInfo newN = new NodeRowInfo();
      copyObjects(T_N, newN);
//     newN = T_N;
      newN = perturb(newN);
      int oldCost = CostFunction(T_N);
      int newCost = CostFunction(newN);
      System.out.println("Cost of Placement
          before perturbation = "+oldCost);
      System.out.println("Cost of Placement
          after perturbation = "+newCost);
      int deltaH = newCost - oldCost;

      double randomNum = (double) (java.lang
          .Math.random());
      double x = Math.exp(-deltaH/T);

      if((deltaH < 0) || (randomNum < x)) {
//       T_N = newN;
        copyObjects(newN, T_N);
        System.out.println("New Placement
            accepted due to lower cost");
      }
      else {
//       perturbCount++;
        System.out.println("New Placement
            rejected due to higher cost");
      }
      M--;
    }
    return T_N;
  }

  public void copyObjects(NodeRowInfo oldN,
      NodeRowInfo newN) {

    oldN.boundaries.copyBoundaries(newN.
        boundaries);

    newN.numberOfCells = oldN.numberOfCells;
    newN.totalWidthOfCells = oldN.
        totalWidthOfCells;
//  newN.nodeId = oldN.nodeId;
    for (Map.Entry<String, NodeRowInfo.Node>
        entry : oldN.nodeId.entrySet()) {
       NodeRowInfo.Node tmpNode=oldN.new Node
           ();
       entry.getValue().copyNode(tmpNode);
       newN.nodeId.put(entry.getKey(),tmpNode
           );
    }
    newN.xLimit = oldN.xLimit;
```

```
//   newN.rowId = oldN.rowId;
    for (Map.Entry<Integer, NodeRowInfo.Row> entry : oldN.rowId.entrySet()) {
       NodeRowInfo.Row tmpRow=oldN.new Row();
       entry.getValue().copyRow(tmpRow);
       newN.rowId.put(entry.getKey(),tmpRow);
    }
//   newN.netToCell = oldN.netToCell;
    for (Map.Entry<Integer, ArrayList<String>> entry : oldN.netToCell.entrySet()) {
      ArrayList<String> tmpNetToCell=new ArrayList<String>();
      for(String e : entry.getValue())
        tmpNetToCell.add(e);
       newN.netToCell.put(entry.getKey(), tmpNetToCell);
    }
    newN.numNodes = oldN.numNodes;
    newN.numTerminals = oldN.numTerminals;
    newN.xCordinateAfterLastPlacedCell = oldN.xCordinateAfterLastPlacedCell;
    newN.yCordinateAfterLastPlacedCell = oldN.yCordinateAfterLastPlacedCell;
  }

  int perturbCount = 1;
  public NodeRowInfo perturb(NodeRowInfo newN) {

//   NodeRowInfo newN = new NodeRowInfo();
    System.out.println("Entering perturb function...");
    int min = 1;
    int max = 3;
    int x = (int)(Math.random() * (max - min) + min);
    if(x == 1) {
      System.out.println("Cell swap randomly selected!");
      newN = cellSwap(newN);
    }
    else if(x == 2) {
      System.out.println("Cell move randomly selected!");
      newN = cellMove(newN);
    }

//   if(perturbCount == 3) {
//     newN = cellMirror(N);
//     perturbCount = 1;
//   }
    return newN;
  }

  public NodeRowInfo cellSwap(NodeRowInfo newN) {
//   NodeRowInfo newN = N;
    String randomCell1 = null, randomCell2 = null;
    int randomNo1 = 0, randomNo2 = 0;
    int randomMin = 1;
    int randomMax = newN.numNodes - newN.numTerminals;

    while((randomNo1 == 0 && randomNo2 == 0)) {
      randomNo1 = (int)(Math.random() * (randomMax - randomMin) + randomMin);
      randomNo2 = (int)(Math.random() * (randomMax - randomMin) + randomMin);
      if(randomNo1 == randomNo2) {
        randomNo1 = randomNo2 = 0;
        continue;
      }
      randomCell1 = "a" + randomNo1;
      randomCell2 = "a" + randomNo2;
      if(!newN.checkIfPadOrMacroblock(randomCell1, randomCell2)) {
        randomNo1 = randomNo2 = 0;
        continue;
      }
    }

    if( (!(randomCell1.equals(null))) && (!(randomCell2.equals(null))) ){
      newN.swapCells(randomCell1, randomCell2);
    }
    newN.updateCenterOfEachCell();
    return newN;
  }

  public NodeRowInfo cellMove(NodeRowInfo newN) {
//   NodeRowInfo newN = N;
    String randomCell = null;
    int randomNo = 0;
    int randomMin = 1;
    int randomMax = newN.numNodes - newN.numTerminals;

    while(randomNo == 0) {
      randomNo = (int)(Math.random() * (randomMax - randomMin) + randomMin);
      randomCell = "a" + randomNo;
      if(!newN.checkIfPadOrMacroblock(randomCell)) {
        randomNo = 0;
        continue;
      }
    }

    if(!(randomCell.equals(null))) {
      newN.moveCell2(randomCell);
    }
    newN.updateCenterOfEachCell();
//   newN.printNodeRowInfo();
    return newN;
  }
//
// public NodeRowInfo cellMirror(NodeRowInfo N) {
//
// }

  public int CostFunction(NodeRowInfo S) {
    int C = 0;

    float c1 = calculateWireLength(S);
    float c2 = calculateOverlapArea(S);
    float c3 = calculateUnevenRowsPenalty(S);
```

```
445
446     double alpha1 = 1.5;
447     double alpha2 = 3;
448     double alpha3 = 1;
449
450     C = (int)((alpha1*c1) + (alpha2*c2) + (
            alpha3*c3));
451
452     return C;
453   }
454
455   public int calculateWireLength(NodeRowInfo
          S) {
456     int wireLength = S.wireLengthCalc();
457     return wireLength;
458   }
459
460   public int calculateOverlapArea(NodeRowInfo
          S) {
461     int overlapArea = S.overlapAreaCalc();
462     return overlapArea;
463   }
464
465   public float calculateUnevenRowsPenalty(
          NodeRowInfo S) {
466     float unevenRowsLength = S.unevenRowsCalc
            ();
467     return unevenRowsLength;
468   }
469
470   public static void main(String[] args)
          throws IOException {
471     long startTime = System.currentTimeMillis
            ();
472
473     String filename = "src\\ibm09\\ibm09";
474
475     TimberWolf T = new TimberWolf();
476
477     NodeRowInfo T_N = new NodeRowInfo();
478 //  NodeRowInfo T_Temp = new NodeRowInfo();
479
480     T.readNodesFile(filename, T_N);
481     T.readWtsFile(filename, T_N);
482     T.readPlFile(filename, T_N);
483     T.readNetsFile(filename, T_N);
484     T.readSclFile(filename, T_N);
485
486     T_N.calculateBoundariesOfEntireRegion();
487     T_N.placeMacroBlocks();
488     T_N.initialPlacement();
489     T_N.updateCenterOfEachCell();
490     T_N.createRowToCellMap();
491     T_N.forGraphicalRepresentation("src\\
            InitialPlacement.txt");
492
493     System.out.println("INITIAL PLACEMENT");
494     T_N.printNodeRowInfo();
495
496     T_N = T.runTimberWolfAlgo(T_N);
497     T_N.forGraphicalRepresentation("src\\
            FinalPlacement.txt");
498     System.out.println("FINAL PLACEMENT");
499     T_N.printNodeRowInfo();
500
501     long endTime = System.currentTimeMillis()
            ;
502     long totalTime = endTime - startTime;
503     System.out.println("Execution Time is: "+
            totalTime);
504   }
505 }
```

C. *Placement.java*

```
1  import java.io.*;
2  import javafx.application.Application;
3  import javafx.scene.Group;
4  import javafx.scene.Scene;
5  import javafx.scene.chart.LineChart;
6  import javafx.scene.chart.NumberAxis;
7  import javafx.scene.paint.Color;
8  import javafx.stage.Stage;
9  import javafx.scene.shape.Rectangle;
10
11 public class Placement extends Application {
12     @Override
13     public void start(Stage stage) {
14       try {
15           Group root = new Group();
16           Scene scene = new Scene(root, 1360,
                 720, Color.WHITE);
17
18           FileReader fr=new FileReader("src\\
                 InitialPlacement.txt");
19           BufferedReader br=new
                 BufferedReader(fr);
20           String line;
21           int i=0,xMin=0,yMin=0;
22           int xMax,yMax,scaleX=1, scaleY=1;
23           while((line = br.readLine())!=null)
                 {
24            i++;
25            String [] words=line.split("\\s+")
                 ;
26            if(i==1) {
27
28            xMax=Integer.parseInt(words[1]);
29            yMax=Integer.parseInt(words[3]);
30            while(xMax >1360 || yMax>720) {
31              xMax=xMax/scaleX;
32              yMax=yMax/scaleY;
33              scaleX++;
34              scaleY++;
35            }
36
37            }
38            else if(i>1) {
39            System.out.println(line);
40            int x= (Integer.parseInt(words[1])
                 /scaleX)*2+50;
41            int y= (Integer.parseInt(words[2])
                 /scaleY)+50;
42            double w= (Integer.parseInt(words
                 [3])/scaleX)*2;
43            double h= (Integer.parseInt(words
                 [4])/scaleX);
44            Rectangle rectangle = new
                 Rectangle(x,y,w,h);
45            rectangle.setFill(Color.WHITE);
46            rectangle.setStroke(Color.
                 CADETBLUE);
47
48             root.getChildren().add(rectangle)
                  ;
```

```
49
50              }
51             }
52
53             br.close();
54             fr.close();
55
56             stage.setScene(scene);
57             stage.show();
58         }catch (Exception e) {
59         e.printStackTrace();
60       }
61
62       }
63
64   public static void main(String[] args) {
65       launch(args);
66
67       }
68
69   }
```