HOMEWORK 2 REPORT

Anay Rakibe: M15158222
Sruthi Padinhattayil: M15214298

Placement algorithm: Force Directed Placement Algorithm

Data Structures used:

1. A net information vector which stores net number, source terminal, target terminal and source and target numbers.
2. A connectivity matrix to store each cell's connections with other cells through nets.
3. A cell weight vector to store the connectivity size of each cell.
4. A cell placement vector to store the location of each cell in the placement matrix.
5. A placement matrix to manipulate the cells and place them at the best possible locations using the force directed algorithm.

Force directed placement algorithm:

Initially, all the cells are placed randomly (all the cells are placed at consecutive positions in the placement matrix). Cells are sorted based on the connectivity of the cells. A cell with highest connectivity is chosen as the seed cell for the algorithm. Its target location is calculated based on the zero force equation (net force on the cell is equated to zero and corresponding new x and y coordinates are found).

The seed cell location is updated as empty. Now we have four cases based on the target location:

1. If the target location is empty, the seed cell is placed at the target location and the target location is marked as locked in the cell weight vector. The algorithm then proceeds by choosing the next highest weighted cell as the new seed.
2. If the target location is occupied, the seed cell is moved to the target location and it is marked as locked. The cell present at the target location is chosen as the next seed for the algorithm to continue.
3. If the target location is blocked, the algorithm calculates the nearest empty cell to the target location by traversing in all four directions. If any of the locations in the vicinity of the target location are also blocked, the abort count is increased by one.
   a. If a new empty cell is found as the new target location, the algorithm proceeds as in Case 1. The cell is moved to the new target location, marked as locked, and the next highest weighted cell is chosen as the new seed.
   b. If a new occupied cell is found as the new target location, the algorithm proceeds as in Case 2. The cell is moved to the new target location, marked as locked, and the cell present at the new target location is chosen as the new seed.
4. If the abort count reaches the abort limit, all the locked cells are unlocked and the same process is repeated. The algorithm continues to run until the iteration limit is reached.

Pseudocode:

```
FORCEDIRECTEDPLACEMENTALGO(noOfCells)
    iterationLimit = 5
    iterationCount = 0
    abortCount = 0
    abortLimit = 0.35 * noOfCells
    counter = 0
    cellCount = cellWeightVector.size()
    emptyCell = coOrdinates(-1, -1)

    // Main loop for the algorithm
    WHILE iterationCount < iterationLimit
        currentCellWeightInfo = cellWeightVector.at(counter)
        endRipple = true

        // If cell is blocked, move to the next cell
        WHILE currentCellWeightInfo.blocked AND counter < cellCount
            counter++
            IF counter < cellCount
                currentCellWeightInfo = cellWeightVector.at(counter)
            ELSE
                BREAK
        // If all cells are blocked, reset the placement and try again
```

```
IF counter >= cellCount
    FOR i = 0 TO cellCount-1
        tempCellWeightInfo = cellWeightVector.at(i)
        tempCellWeightInfo.blocked = false
        cellWeightVector.at(i) = tempCellWeightInfo
        tempCellInfo = cellPositionVector.at(tempCellWeightInfo.cellNumber)
        tempCellInfo.blocked = false
        cellPlacementMatrix[tempCellInfo.x][tempCellInfo.y] = tempCellInfo
        cellPositionVector.at(tempCellInfo.cellNumber) = tempCellInfo
    END FOR
    counter = 0
    iterationCount++

// If cell is not blocked, perform force-directed placement
IF NOT currentCellWeightInfo.blocked
    sourceCellInfo = cellPositionVector.at(currentCellWeightInfo.cellNumber)
    endRipple = false

    // Ripple outwards from the cell until placement is successful or aborted
    WHILE endRipple == false
        cellConnectivitySize = cellConnectivityMatrix[currentCellWeightInfo.cellNumber].size()
        targetX = 0
        targetY = 0

        // Calculate average position of connected cells
        FOR i = 0 TO cellConnectivitySize-1
            connectedCellNumber = cellConnectivityMatrix[currentCellWeightInfo.cellNumber][i]
```

```
    connectedCellX = cellPositionVector[connectedCellNumber].x
    connectedCellY = cellPositionVector[connectedCellNumber].y
    targetX = targetX + connectedCellX
    targetY = targetY + connectedCellY
END FOR

// If there are connected cells, set the target position to their average
IF cellConnectivitySize != 0
    targetX = targetX / cellConnectivitySize
    targetY = targetY / cellConnectivitySize
    targetCellInfo = cellPlacementMatrix[targetX][targetY]
ELSE
    targetCellInfo = cellPlacementMatrix[targetX][targetY]

// Attempt to place cell at target position
IF targetCellInfo.cellNumber == -1    //If new target position is vacant
    targetCellInfo.cellNumber = currentCellWeightInfo.cellNumber
    targetCellInfo.blocked = true
    targetCellInfo.cellWeightVectorIndex = sourceCellInfo.cellWeightVectorIndex
    currentCellWeightInfo.blocked = true
    cellPlacementMatrix[targetCellInfo.x][targetCellInfo.y] = targetCellInfo
    cellPositionVector.at(targetCellInfo.cellNumber) = targetCellInfo
    cellWeightVector.at(targetCellInfo.cellWeightVectorIndex) = currentCellWeightInfo
    IF NOT sourceCellInfo.blocked
 sourceCellInfo.blocked = false
```

Routing Algorithm: Lee Algorithm

Data Structures used:

1. A struct to store lee numbers, blocks, via, net numbers, net surrounding information in both the layers.
2. A cell coordinate vector to store cell terminal locations.
3. A queue to run breadth first search.

Lee Algorithm:

Wave Propagation: Lee algorithm tries to propagate a wave from source to target by assigning a number equivalent to Manhattan distance at each block. This wave propagation ends when the target point is reached. Lee propagation happens in two layers. Whenever there is a blockage in one layer, it checks in the other layer and assigns lee number if there is no blockage.
Complexity: O(L^2) L: Manhattan distance between source and target.

Back trace: Back trace starts at target and tries to go in the same direction until it can and then take turns to reach destination whenever it cannot go in the same direction. At every stage it tries to go to the block which has a lee number less than its previous value. Whenever there is a layer change (m1 to m2 or m2 to m1), a via is inserted at that location.
Complexity: O(L)

The choice of data structure used is a chip grid matrix to represent the chip's layout, with each block in the matrix representing a cell in the design. The algorithm updates the Lee number of each block as it visits them and uses a queue to keep track of the unvisited blocks.

Pseudocode:

```
function RunLeesAlgo(noOfNets, failedMethod, metalType):
    for each net in the design:
        if net.routingMetal is not set:
            sourceCellNumber = net.sourceCellNumber
            targetCellNumber = net.targetCellNumber
```

```
        sourceCellCoOrdinates = cellCoOrdinateVector[sourceCellNumber-1]
        sourceTerminalCoOrdinates = FindTerminalCoord(sourceCellCoOrdinates,
    net.sourceCellTerminalNumber)
        source = chipGridMatrix[sourceTerminalCoOrdinates.x][sourceTerminalCoOrdinates.y]
        targetTerminalCoOrdinates = FindTerminalCoord(cellCoOrdinateVector[targetCellNumber-1],
    net.targetCellTerminalNumber)
        target = chipGridMatrix[targetTerminalCoOrdinates.x][targetTerminalCoOrdinates.y]
        failedMethod = GetLeesNofromSourcetoTerminal(source, target, metalType)
        if failedMethod is 0:
            failedMethod = Backtraceroute(net.netNumber, chipGridMatrix[source.x][source.y],
    chipGridMatrix[target.x][target.y])
        ClearLeesNofromSourcetoTerminal(chipGridMatrix[source.x][source.y],
    chipGridMatrix[target.x][target.y])
        if failedMethod is not -2 or failedMethod is not -1:
            net.routingMetal = metalType
    print "Number of nets routed:", count

function GetLeesNofromSourcetoTerminal(source, target, metalType):
    source.m1LeeNumber = 0
    source.m2LeeNumber = 0
    unVisitedQueue.push(source)
    targetReached = -1
    while unVisitedQueue is not empty:
        currentCell = unVisitedQueue.front()
        unVisitedQueue.pop()
        if currentCell.x == target.x and currentCell.y == target.y:
            targetReached = 0
            break
        for each neighbor of currentCell:
            if neighbor is not blocked and not visited:
                if neighbor is not via:
                    if neighbor.m1Blocked is 0 or (not neighbor.m1NetSurrounding and
    ifBlockinsurrounding(neighbor, source, target)):
                        neighbor.m1LeeNumber = currentCell.m2LeeNumber + 1
                    if neighbor.m2Blocked is 0 or (not neighbor.m2NetSurrounding and
    ifBlockinsurrounding(neighbor, source, target)):
                        neighbor.m2LeeNumber = currentCell.m1LeeNumber + 1
                else:
                    if neighbor.m1Blocked is 0 and neighbor.m2Blocked is 0:
                        neighbor.m1LeeNumber = currentCell.m2LeeNumber + 1
                        neighbor.m2LeeNumber = currentCell.m1LeeNumber + 1
        mark currentCell as visited
    return targetReached

function Backtraceroute(netNumber, source, target):
    currentCell = target
```

```
while currentCell is not source:
   if currentCell.m1LeeNumber is not -1 and currentCell.m1LeeNumber is less than or equal to
  currentCell.m2LeeNumber:
      currentCell.m1NetNumber = netNumber
      currentCell.m1Blocked = 1
      currentCell = cell with m1LeeNumber equal to currentCell.m1LeeNumber - 1
   else:
   currentCell.m2NetNumber = netNumber
      currentCell.m2Blocked = 1
      currentCell = cell with m2LeeNumber equal to currentCell.m2LeeNumber - 1
return 0
```

HOW THE PLACEMENT AND ROUTING ALGORITHM INTERACT:

- Force-directed placement algorithms determine the location of the cells on the chip based on the interconnections between them, while Lee's algorithm determines the optimal routing of the interconnections.
- The output of force-directed placement algorithms is typically a placement database that contains the coordinates of each cell on the chip. This placement database is then used as input to Lee's algorithm, which generates a routing database that specifies the paths of the interconnections between the cells.
- The routing database can then be used to generate a physical design layout that includes detailed information about the interconnections between the cells, such as wire widths and spacing. This layout is then used to create the masks needed for the manufacturing of the chip.
- In summary, force-directed placement algorithms and Lee's routing algorithm work together to determine the location and routing of the cells on the chip, respectively. The placement database generated by force-directed placement algorithms is used as input to Lee's algorithm, which generates a routing database that specifies the paths of the interconnections between the cells. This routing database is then used to create the physical design layout of the chip.

BM	BOUNDING BOX DIMENSION (1x1)	BOUNDING BOX AREA (Square unit)	NO OF NETS ROUTED	TOTAL WIRELENGTH	NO OF VIAS	EXECUTION TIME(S)	MEMORY USED
B1	135x150	20250	42	3403	177	3	191KB
B2							
B3							
B4	899 x 902	810898	365	220252	10735	60	8.5MB
B5							
B6	1157 x 1164	1346748	524	335944	15677	100	13.4MB

B7	1172 x 1201	1407572	502	368306	16063	150	14.4MB
B8	1311 x 1282	1680702	434	385476	15014	240	15.3MB
B9	1411 x 1409	1988099	447	406801	15957	450	16.7MB
B10							

NOTE: Execution of B2, B3, and B5 ended abruptly before generating mag file. B10 didn't finish execution even after 5 hours.

6.. Include a retrospective describing what you think is the cause of the good/poor performance of your tool and how you might be able to improve the performance (both execution speed and quality of result) of your system if you had more time?

- Optimization of data structures and algorithms: One potential way to improve performance is to optimize the data structures and algorithms used in the algorithm. For example, using efficient data structures like hash maps or binary trees can reduce search times and increase performance. Additionally, using parallel computing techniques can distribute the workload across multiple processors or threads, further improving performance.
- Reducing the number of iterations: Another way to improve performance is to reduce the number of iterations required to reach a good solution. This can be done by improving the convergence criteria or by using more advanced optimization techniques.
- Improving the quality of the force model: The performance of the algorithm is also dependent on the quality of the force model used. By improving the accuracy of the force model or using a more advanced model, the algorithm can achieve better results with fewer iterations.

- Since the lee algorithm is based on a pathfinding algorithm and backtracking, it may be slower for larger input sizes due to the complexity of the algorithm.
- To improve the performance of the system, some potential areas of improvement include optimizing the data structures and algorithms used, parallelizing the computation, for example, using a more efficient data structure for the chip grid, such as a quadtree may improve the speed of the pathfinding algorithm.

- Additionally, using a heuristic approach to guide the search may also improve the speed of the algorithm.
- Use more efficient data structures: For example, you might consider using a priority queue instead of a regular queue to store the unvisited nodes in Lee's algorithm.