1. Elmore Delay [20 pts]

Consider the small layout shown below. As in the class notes, assume the resistance $R$ and capacitance $C$ of a wire segment can be calculated as:

- $R = r \cdot L/W$, where $r = 1/2$
- $C = c \cdot L \cdot W$, where $c = 2$

Do the following:

- Compute the Elmore delay for each of the 3 driven gates in this layout. Draw your resulting RC tree and show the calculations. Note that the length of the path from the driver gate to each driven gate is the same, 16 units. Are all the delays the same? If not, why not?

- Suppose we want to equalize the delay to each of the 3 driven gates. If you are allowed to change the width $W$ of any single wire segment in this layout, which would you change, and what would the new value of $W$ be to equalize the delays? (Note, we don’t specify what the delay is, just that all 3 delays are the same). Show the algebra here, tell the final $W$, and tell the new equalized delay. (Hint: write that segment’s pi model in terms of the variable $W$, then write the delay for the left side and right side of the wire and solve for $W$ to make them equal. It’s just one equation in one unknown.)
2. Elmore Delay for a Parameterized, Constrained Clock Tree [30 pts]

Consider the symmetric clock tree layout shown below. It has 7 segments with fixed length, with 3 of these having variable width. It has 4 leaf nodes. Now assume that the resistance and capacitance can be calculated as:

- \( R = r \cdot \frac{L}{W} \), where \( r = 50 \times 10^{-3} \)
- \( C = c \cdot L \cdot W \), where \( c = 0.2 \times 10^{-15} \)
- Assume that \( L, W \) must be in units of microns for the above formulas to work right. Note that some lengths in the picture are in millimeters. Remember to convert!

Suppose we want to make the delay as small as possible here. One simple way is a monotone widening scheme: the bottom-most wires are all width=1um, and each level up the clock tree has wires \( K \) times wider than the level below. There are 7 segments in this clock, but only one designable variable now: \( K \). Do this:

- How fast can we make this clock run, i.e., what is the minimum Elmore delay \( \tau \) that we can achieve if we can set \( K \) to any positive number? (\( K \) does not have to be an integer).
- How fast can we make this clock if we limit the maximize width of the widest wire to 9 um?

- Suppose we also have a power limit. To first order, the power in the clock is \( CV^2f \), where \( C \) is the total capacitance of the entire clock (segments + leaf nodes), \( V \) is the voltage of the clock signal (let’s call it 1V to make life easy), and \( f \) is the frequency of operation (Hz) of the clock. Let’s assume we can approximate \( f \) as \( 1/\tau \). How big can we make \( f \) if we have a total limit of 5 milliWatt dissipated in the clock, i.e., \( CV^2f \leq 0.005 \)?

(Hint: Turn each segment into a pi model as a function of \( K \), write delay as a function of \( K \), and total capacitance as a function of \( K \). You can solve nonlinear equations graphically by plotting for various values of the variable)
3. Quadratic Placement [30 pts]

Consider this simple netlist with fixed pins, which has 3 placeable objects. All the 2-point wires have $C_{ij}=1$ except the two wires labeled “W” in the figure:

Do this:

- Assume $W=1$ so all wires have unit weight, and show how to formulate and solve the quadratic placement problem as in the class notes. Show the $[C_{ij}]$ matrix, the $[A]$ matrix, the two $b$ vectors (one for solving the $x$ problem, one for the $y$ problem). Solve the two resulting $3x3$ matrix problems (however you want to do it) to get a placement. Plot the placement as in class (you can do it by hand, or you can be fancy and use a program; either is OK).

- Now, assume $W=64$, and repeat the above placement exercise, again showing all the matrices, vectors, and final placement.
4. Quadratic Placement Again: Linear Reweighting [20 pts]

We mentioned in the class the fact that the quadratic formulation for point placement requires us to “relax” a lot of physical assumptions about our cells (ie, they are points) and our nets (all 2-point, quadratic length metric) to be able to solve it. It turns out one can “add back” a lot of these lost features with some clever mathematical formulation. In this problem we look at a trick that tries to make the wire lengths we compute behave a little more like linear half-perimeter length calculations.

Consider this trivial one-dimensional placement problem at left below: one movable object at “x”, two fixed pins at left and right, four nets with weight=\(wij=1\). We would model this as the simple quadratic placement problem shown below at the right: two 2-point connections with \(Cij = 3\) and \(1\), respectively.

Do this:

• **Solve** this simple placement problem using the quadratic wirelength metric from class.
  You don’t neet any matrix stuff: just write the quadratic wirelength as a function of the one variable “x”, then minimize it with standard calculus.

You will observe that the answer puts the cell closer to \(x=0\). Each of the three wires at the left ends up longer in order to try to minimize the length of the wire on the right. The reason is that when the wirelength is quadratic, a longer wire--say, one that is 2X longer--counts as \(4\) times more in this cost function than the other wires. It has been observed that this can sometimes cause problems in placement: the long wires get minimized very aggressively, but this happens at the expense of many short wires becoming longer. Overall, this can lead to a lot more total wirelength on big placements.

It turns out there is actually a mathematical trick that almost fixes this, called *linear reweighting*. The idea is that instead of solving the quadratic placement problem once, we solve it several times, but we change (update) the \(Cij\) weights on the individual wires using information from the most recent placement. The algorithm is very simple:

\[
\begin{align*}
&\text{set } k=0 \\
&\text{solve the placement problem for the first time to get } x[0], \text{ initial solution} \\
&\text{done } = \text{false} \\
&\text{while (!done)} \{ \\
&\quad \text{update each } Cij: Cij(\text{new}) = \sum_{\text{wires}} [wij(\text{old}) / (\text{length of associated wire}) ] \\
&\quad k = k + 1 \\
&\quad \text{re-solve quadratic placement problem using updated } Cij(\text{new}) \text{ to get } x[k] \\
&\quad \text{if (the value for } x[k] \text{ is not changing much over last few solutions)} \\
&\quad\quad \text{done } = \text{true} \\
&\} 
\end{align*}
\]
The idea in the general case is that if we optimize the wirelength $C_{ij}(x_i - x_j)^2$ we get a quadratically “weighted” length. But if we iteratively reweight, roughly like this:

$$C_{ij} \left| x_i[k] - x_j[k] \right| (x_i[k + 1] - x_j[k + 1])^2$$

(EQ 1)

and re-solve, then the effect of the linear distance from the k-th iteration divided into the quadratic distance being solved in the next (k+1) iteration tends to make the overall wire-length behave like a linear function instead of a quadratic function.

To make this work out right, you actually have to go back to the “original” wires at the left of our figure, and divide each wire’s original $wij$ weight by its most recent placed length, and then add these up to get the new $Cij$. For us, this means adding three reweighted terms to the get the $Cij$ for the left connection, and just one reweighted term to get the final $Cij$ for the right connect.

It actually does work; do this:

- You already have $x[0]$ from the previous bullet. Divide each $wij$ by the length of the associated wire from this solution, sum them up as in the algorithm, and reformulate the quadratic placement problem with these two new $Cij$, and solve again to get $x[1]$. Repeat this **once** more to get $x[2]$.

- Comment on the final solution: what is it doing to the long wire at the right originally found in the $x[0]$ solution? What is it doing to the three short wires in this original $x[0]$ solution?
5. Geometric Data Structures [20 pts]

Consider this simple layout, which has 11 rectangles (labelled a-k) on 2 layers: the dark grey layer, and the lighter (it’s actually striped) layer:

Do this:

- Draw the **quadtree** that represents this layout. You don’t need to try to draw the actual rectangle in each node of the quadtree, just show the quadtree’s overall structure (quadrants, cut lines, hierarchy) and what rectangles (by letter) live in each part of the tree.

- Draw the **maximally horizontal corner stitched tileplane** that represents this layout. Assume we want a single tileplane, so that each tile will be a unique combination of layers. So, we will have space tiles, dark-grey tiles, light-grey-striped tiles, and dark-grey-AND-light-grey-striped tiles. Label things clearly so we can tell what each tile is. Draw the grid coordinates as above so we can see where the edges of each tile fall. Remember the canonical form for the tileplane: solid tiles are maximally wide and then maximally high; solid tiles cannot have space tiles (or the edge of the whole layout) at their extreme left/right ends, OR, in this case, they have another solid tile with a different “color”, ie, a different set of solid layers inside the neighboring tile.