18760 Project 2: Analytical Placer

Xiang Lin (xianglin)
Clement Loh (changshl)

Goals

- Build analytical placer (based on optimization of cost function)
- Reasonably fast
- Produces balanced result between wire length, discrepancy and area utilization (75%)
- To keep discrepancy strictly under 3.00 for all benchmarks and seek to achieve a utilization of around 0.5.

Features

- Gradient descent optimization of cost function
- Weighted combination of wire length, gate density and boundary penalty
- “Outer Loop” that tweaks parameters
- Final legalizer

The analytical placer we have built is written in MATLAB. It uses a provided optimizer (gradient descent) to find the minimum of a cost function that takes a given placement and maps it to a numerical score. While ideally an analytical form of the gradient of the cost function would be used in the gradient descent algorithm, since the cost function does not have a closed-form expression, this gradient function is computed numerically instead. This optimization is performed multiple times in an “outer loop”: each iteration runs the gradient descent optimizer, but changes some parameters each run and uses the placement generated by the previous run as a starting point. Finally, a simple legalizer is used to ensure that all gates are within legal boundaries.

The cost function we use can be written as:

cost = W_{WL}* wireLength(x) + W_{DP}* densityPenalty(x) + W_{BP}* boundaryPenalty(x)

where x is a vector representing the coordinates of each gate. Wire length is approximated using half-perimeter wire length of each net, density penalty is computed using a potential contribution of each gate to an imaginary grid (that gets progressively fine-grained), and boundary penalty is exponentially proportional to how out-of-bounds a gate is.

wireLength

To compute wirelength, we take the current positions of all the gates and pins on the chip and compute the Half-Perimeter Wire Length (HPWL) of the extremes in the x and y dimensions.

densityPenalty

We compute densityPenalty as a function of gridLength. We assume a normal potential function of radius/2 sigma in both x and y dimensions centered at the position of every gate. For each cell in the grid, we compute a density penalty for each gate as dpOfGate = (gateSize*fundamentalUnit*normpdf(x)*normpdf(y)/radius^2 - gridpointCapacity)^2

boundaryPenalty

We compute a boundaryPenalty for every gate that is placed outside out the 100x100 chip area. The total penalty is the sum of (exceeded_length / alpha)^2 for each dimension of every gate.

The gradient function computes the gradient vector numerically; each dimension individually modified, and then the difference in cost is computed, thus resulting in the gradient for that dimension. This is performed for all dimensions. The value of the increment is small enough that it results in a reasonably accurate estimate of the actual gradient.

The outer loop varies some parameters. In particular, it makes each progressive run more fine-grained by tuning the gridlength (affecting number of imaginary grid points) and radius of the potential function. These values (and formula for next iteration) have been tuned to give a reasonable result without sacrificing too much in computational runtime. The weights in the cost function are also tweaked with the the goal of balancing wire length and utilization. Based on our tweaking, the initialized values of each parameter are as follows:

W_{BP}	100
W_{DP}	1
W_{WL}	5 * densityPenalty / wireLength;
radius	2
gridLength	100 / (numGates - 1);

alpha	gridLength*radius

The outer loop runs twice; the following parameters are updated on the second run:

radius’	radius + 1
gridLength’	gridLength / 10

Finally, a legalizer ensures that the resultant placement does not have gates outside of legal boundaries. As the W_{BP} term is fairly large, a simple rounding is performed.

Results

The following results were obtained by running the provided checker on 4 of the sample benchmarks:
Running on an i5-2520M with 4GB RAM,

	Gates	Wirelength	Discrepancy	Utilization	Elapsed Time (s)
toy1	18	1704.22	2.00	0.39	15.02
toy2	32	3805.06	2.83	0.49	28.39
fract	125	14304.18	2.95	0.51	140.72
primary1	752	80824.76	2.43	0.50	3912.11

Note that the results were obtained by fixing the seed for random (initial) placement for consistency and testing. (line 9, parse.m)

As we discussed earlier, we tweaked our parameters to try to keep utilization and discrepancy as close to 0.75 as possible, while reducing wirelength. While it was possible to achieve a shorter wirelength by incrementing the penalty term W_{WL} , we decided to trade off better utilization and discrepancy values for a longer wirelength to maintain our goal of keeping discrepancy under 3.00 and utilization of about 0.5. With regards to speed of our code, we observe that there is a non linear increase in elapsed time with respect to problem size, and perhaps a future improvement would be to implement the placer in a more efficient language such as C++.

Another possible improvement to our method would be to find a method to recalculate the cost and gradient functions incrementally. A numerical estimate (assuming a

constant second order gradient) might work fairly well, as the cost function is limited to quadratic and exponential terms.