Program #4: Simulated Annealing Placement

The next step in implementing the Aphyds system is the creation of a simulated annealing based placer. In later assignments we will then route the resulting layouts.

Algorithm Description

Simulated annealing is composed of a set of major elements: cost function computation, move function, cooling schedule, and the iterative simulated annealing engine. The Aphyds system provides you with the cooling schedule and parts of the cost and move functions. Also, a greedy placement algorithm is provided as an example for the development of your simulated annealing engine.

Your efforts will focus on the functions Semiperimeter, moveFunction, and doAnnealPlacement:

Semiperimeter: The cost function for your annealer is the sum of the semiperimeters of the nets in the circuit. The core of this is the semiperimeter function, which computes the semiperimeter (1/2 the perimeter of the bounding box) of a single net. Basically, you must figure out the size of the smallest rectangle containing all nodes directly connected to this net.

moveFunction: Simulated annealing is based upon making many random swaps of elements. By carefully determining which of these moves are accepted, a good final placement can be achieved. The moveFunction generates the random swaps that are evaluated. There are two constraints on these swaps: I/O placement and the bounding box. First, I/Os must be placed at the periphery, and non-I/Os cannot be put at the periphery. Specifically, the placement will assign positions (W,H), where 0<=W<W_max, 0<=H<H_max. W_max and H_max are provided by getPlacementDim(). All I/Os must be at an edge of the region, either (0,X), (W_max-1,X), (X,0), or (X, H_max-1). All non-I/Os cannot be placed in these locations, but can only be in the middle. Specifically, non-I/Os are confined to positions (W,H) where 0<W<W_max-1, 0<H<H_max-1. Second, the bounding-box radius limits swaps to relatively local moves, which tends to improve the annealing quality. Specifically, if we are going to swap two locations (W_1,H_1) and (W_2,H_2), we require -BBradius <= W_1-W_2 <= BBradius and -BBradius <= H_1-H_2 <= BBradius.

doAnnealPlacement: This is the main engine of your placement algorithm. It calls moveFunction to create a swap, and test it with moveAndComputeCost. This is done for multiple steps, controlled by the temperature schedule. The routine doGreedyPlacement is a complete greedy placement algorithm, which will have a reasonable amount of similarity to this routine – understanding doGreedyPlacement will help you write much of doAnnealPlacement.

Getting Started

You will build upon the code you produced in the previous programming assignments. Again, to run the program you compile and run “CircuitViewer.java”. Then, open a circuit (File->open) and start placement (Edit->Place & Route Circuit). An initial placement of the circuit is shown in the dialog box, along with a graph of the net lengths (which is incorrect until you implement the semiperimeter routine). The nets will be color-coded accordingly on the placement. Click one of the nodes in the placement – you can use the node and net dialogs on this view as well. Notice that the node dialogs now have a highlight placement field, and the net has a semiperimeter. You can move a node to a new location (swapping with whatever is there) by editing the node’s placement field.

Your first task will be to write the Semiperimeter routine. You can test the results by looking at the Semiperimeter field on the net dialog box. You can also move nodes attached to a net around for testing.

The next step will be to get the moveFunction routing working. The results of this can be tested by running the greedy placement algorithm in doGreedyPlacement. Click Greedy Placement on the action menu of the Place and Route Dialog. Both Greedy and Simulated Annealing placement are controlled by the Iterations, Moves/Iterations/Node, and Bounding Box Radius fields on this dialog. When you start the greedy placer, it
displays its progress via the placement graph and the results table (the table is at right on the dialog). The table has the fields Iter (Iteration number), Curr (Current cost, the cost at the end of this iteration), Best (the best cost in the iteration), Wrst (the worst cost accepted in the iteration), wTry (the worst cost tried in the iteration), Acpt (% of attempted moves accepted), and Bacp (% of bad moves accepted). The graph shows much of this as well. Although some of these numbers are irrelevant for greedy placement (like Bacp, which is always 0) they will be very useful during annealing.

Running greedy placement will test moveFunction. A defective moveFunction will typically show up in two ways: first, error messages will show up from testMoves if the moves are incorrect. Next, check to see that nodes can move into previously empty locations in the placement – typically you can watch the I/O pins and see if they move to empty locations. The corners may also be worth watching – does your algorithm ever add or remove elements from here? Spend some time using the Move Function dialog that pops up after the placement is done – does your move function move nodes to every location within the bound box? Although random variation may miss some moves for short runs, for longer runs all possible moves will likely be attempted at least once. Note that it is difficult, perhaps impossible, to have all moves occur with equal probability. However, if there are some moves that just never happen this may be a sign of a bad moveFunction.

Finally, you will need to write doAnnealPlacement. This will be based largely on the ideas from doGreedyPlacement, with help from other routines listed in the Javadoc. You can run your program from the Action menu. Note that the cooling schedule performs one pass initially that accepts all bad moves, scrambling up the placement. Thus, you can run multiple annealings without reinitializing the layout. DoGreedyPlacement iteratively improves the current layout, so you can try using doGreedyPlacement on an already annealed circuit (essentially doing further quenching).

When either algorithm is done, you will be able to see the improvement from the graph in the lower right, as well as the Ttl Semiperimeter field, which is the total cost function of the placement. The net color-coding will be the same as that of the graph, so that you can easily see the results.

**Programming Assignment**

In this programming assignment you will need to implement the functions Semiperimeter, moveFunction, and doAnnealPlacement in Placer.java. You should look at the Placer.html page produced by Javadoc to understand what needs to be done, and what routines will be useful in your efforts. In general, each of the routines will be relatively easy to write, especially with the helper routines described for each method.

This programming assignment will require understanding the public and protected functions (those that appear in the .html files from Javadoc) for Placer.java. You should only need to actually modify Semiperimeter(), moveFunction(), and doAnnealPlacement().

**Experiments**

Once your algorithm is running, you should experiment with greedy vs. annealing based placement. Also, try playing with the controls of the annealing algorithm – number of iterations, number of moves per iteration, and bounding box – as well as multiple annealing runs for the same circuit. This is a random algorithm, and different runs can produce different results.

A couple good test cases are Mesh.circ, Serpent.circ, and ShortSerpent.circ. These are a 2-D Mesh, and two linear circuits, for which an optimal placement is easy to determine. However, the greedy placer should have significant problems, and the annealer may take a longer running time to succeed.

Finally, you should examine the placement graph for a simulated annealing run, especially for a longer run for a larger circuit. You can see the random nature of the algorithm, and the overall improvement of the results. A similar run for greedy placement will demonstrate the limits of such a greedy algorithm.