Programming Assignment Two: Placement – 3 Quadratic Placement

Out: June 21, 2016; Due: July 6, 2016.

I. Motivation

To give you experience in implementing a quadratic placer.

II. Programming Assignment

In class, we described how to build a quadratic placer that uses matrix solvers to optimize the placement. This placer will use a recursive partitioning strategy to deal with the fact that the gates will be clustered in overlapping, nonphysical ways, after each quadratic placement (QP) solution. In this assignment, you are going to implement a core "3QP" placer.

The 3QP placer requires you to execute exactly 3 Quadratic Placement (QP) matrix solves. The first one solves on the full-size netlist, the second one solves on the gates assigned to the left half of the chip surface, and the third one solves on the gates assigned to the right half of the chip surface. This will require you to demonstrate that you understand and can implement most of the technical ideas behind a real, quadratic placer.

Below is the precise set of steps we want you to implement:

1. **Reading Input**. You read from **standard input** a netlist describing the circuit for you to place. The circuit contains a number of gates, nets, and input/output (I/O) pads, which have fixed locations around the edges of the chip. The chip is a square, with dimension in both X and Y going from 0 to 100. The details about the input format will be discussed in Section III.

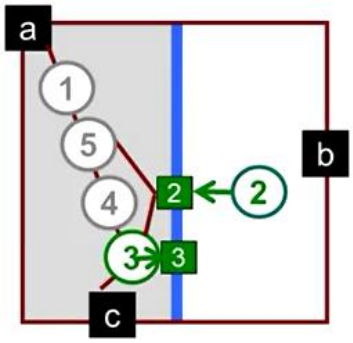
2. **QP1—First Quadratic Placement**. You set up the A matrix and the b_x and b_y vectors to solve the first quadratic placement. (This is, of course, 2 matrix solves: one for X and one for Y). The result is a placement of the gates inside the 100x100 square that defines the chip.

3. **Assignment**. Sort the gates placed after QP1 to decide which go to the left and which go to the right. The first partitioning cut is vertical. So, you will first sort on X coordinate of the point. If there are any ties, further sort on Y coordinate. If there are still any ties, further sort on node ID. If the number of gates is odd, you cannot have exactly the same number of gates on each side. In

this case, put smaller number of gates on the left. For example, if you have 9 gates to assign, put 4 gates on the left. **Note**: The QP1 solve may be extremely unbalanced. You might find all the gates want to be on the left, or none of the gates want to be on the left. This is why the assignment step is critical.

4. **Left-side Containment**. Propagate the gates assigned to the right side and pads in the right side that are connected to these left-side gates, and place them on the **centerline** of the partition. This just means: find the right-side gates/pads that have nets connected to the left-side gates, and represent them with their same Y coordinate, but with their new X coordinate as 50.0.

Note: in this step, although some gates assigned to the right side are now in the left part due to the first QP, they should also be propagated to the centerline, since they finally will be placed into the right region. An example is gate 3 shown in the figure below. (This is from the partitioning example discussed in our lecture.) After first QP, we decide to assign gate 3 to the right side. Although gate 3 is in the left side after first QP, it should also be propagated to the **centerline**.



5. **QP2—Second Quadratic Placement**. Solve the new QP2 placement problem. This means: set up the new *A* matrix (note: it has ~ ½ of the gates, the ones assigned to the left) and the 2 new *b* vectors. Solve it, for the new *X* and *Y*.

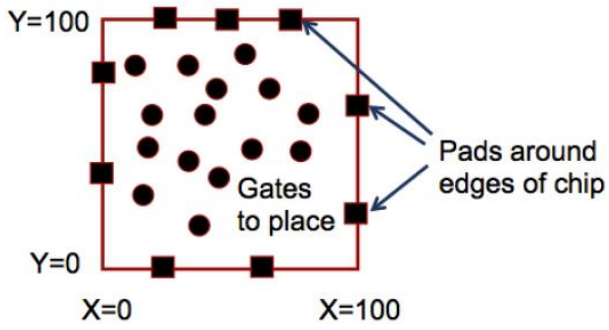
6. **Right-side Containment**. Now, you need to place the right side gates. So, first, propagate the gates and pads on the left side that are connected to the right-side gates, using their new locations from QP2 (!) to define the new right-side placement problem. This just means: find the left-side gates/pads that have nets connected to the right-side gates, and represent them with their same Y coordinate, but with their new X coordinate as 50.0.

7. **QP3—Third Quadratic Placement**. Solve the new QP3 placement problem. This means: set up the new *A* matrix (note: it has ~ ½ of the gates, the ones assigned to the right) and the 2 new *b* vectors. Solve it, for new *X* and *Y*.

8. **Output**. Print out the (*X*, *Y*) coordinates of all the gates in a simple textfile. See Section IV for details about the output format.

III. Input Format

You will read from the **standard input** a description of the netlist, which specifies the gates in the circuit and the nets that connect these gates. The nets can have any number of gates to which they connect, not just two! The file also specifies a set of input output (I/O) pads, which have fixed locations around the edges of the chip. The chip is a square, with dimension in both X and Y going from 0 to 100. The X coordinate increases from left to right and the Y coordinate increases from bottom to top. The figure blow illustrates the setup of the placement problem.



The input file format is illustrated using the following example:

```
4 6 // NumGates G and NumNets N
1 3 1 2 5 // Gate# NumNetsConnected NetNumber...
2 2 5 6 // Example: Gate#2 is connected to two nets, #5 & #6
3 3 3 4 5
4 2 2 3
4 // Number of pads P
1 1 0 0 // PadID NetNumberConnectedTo PadX PadY
2 2 100 0 // Ex: Pad #2 connected to Net #2 at (X,Y)=(100, 0)
3 6 0 75
4 4 75 100
```

The first line has two integers, which specify the number of gates *G* and the number of nets *N*, respectively.

The next *G* lines specify the gates, with one line per gate. Each line begins with a Gate ID number (starting from 1 and continuing in order without any gaps to gate #*G*). The second number *M* in each line is the number of nets connected to this gate. After that, there are *M* numbers, which are the Net IDs of the *M* nets this gate is connected to.

The next line shows the number of pads *P* on this chip.

The next *P* lines specify the pads, with one line per pad. Each line begins with a Pad ID (starting at 1, and continuing in order without any gaps to pad #*P*), followed by the Net ID this pad is connected to, followed by the X coordinate and the Y coordinate of the pad.

Note:

1. The comments in the above example are for illustration purpose. In the real input, there are no comments.

2. The weights of the nets are not specified. Therefore, just assume all of them are 1. However, in setting up the matrix, you need to replace a *k*-point net with $k(k-1)/2$ 2-point net. The weight of each 2-point net should be $1/(k-1)$. With this transformation, there may be multiple 2-point nets connecting two gates. Then, the connection weight between these two gates should be the sum of the weights of all these nets that connect these two gates.

3. Typically, we will describe the netlist in a file. However, since your program takes input from the standard input, you need to use the Linux input redirection "<" on the command line to read the netlist from the file.

IV. Output Format

The output is printed through `cout`. It contains *G* lines, with each line specifying the final location of a gate. Each line looks like:

```
GateID floatXcoordinate floatYcoordinate
```

where `GateID` is the gate ID, and `floatXcoordinate` and `floatYcoordinate` denote the *X* and *Y* coordinates of the gate in floating numbers, respectively. **The output order of these lines follows the gate ID order.** For each floating number, please print out the value with 4 digits after the decimal point. An example is shown below.

```
1 69.0724 51.5016
2 29.5798 7.4603
3 40.6463 23.1241
4 27.9712 63.7866
```

V. Matrix Solver

You need to be able to solve a large matrix to complete this assignment. So, we are providing you with such a solver. The source code of the solver can be found in Programming-Assignment-Two-Related-Files.zip. Besides the source code, we also show you a small example of using the subroutines. The solver is a simple iterative matrix solver based on the Conjugate Gradient (CG) method.

One thing to note however: you will get a solution very close to the perfect answer, but always a little bit "off". This is usually never a problem. But in this context, one strange thing can happen. Suppose you are working on the *X* solver for the left-side partition. You expect all the *X* coordinate in the $Ax=b_x$ solve to come out as floating point numbers between 0.000000000000000 and 50.000000000000. But, it is entirely possible for the *X* coordinate to be 50.0000000000000002, or something similar. Don't be alarmed – this is just how things work in the real world.

VI. Program Arguments and Error Checking

Your program takes no arguments. You do not need to do any error checking. You can assume that all the inputs are syntactically correct.

VII. Implementation Requirements and Restrictions

- You must make sure that your code compiles successfully on a Linux operating system. You are required to write your own `Makefile` and submit it together with your source code files. **Your compiled program must be named as `p2` exactly.**
- You may `#include <iostream>`, `<fstream>`, `<sstream>`, `<iomanip>`, `<string>`, `<cstdlib>`, `<cassert>`, `<ctime>`, `<climits>`, `<vector>`, `<deque>`,

`<list>`, `<map>`, `<set>`. No other system header files may be included, and you may not make any call to any function in any other library.

- Output should only be done where it is specified.

VIII. A Simple Test Case

We have provided you with two simple test cases in the Programming-Assignment-Two-Related-Files.zip. One test input is called `4nodes.in`. The correct output is written in `4nodes.out`. To do the test, type the following command into the Linux terminal once your program has been compiled:

```
./p2 < 4nodes.in > test.out
```

If the values in the `test.out` are quite different from those in the `4nodes.out`, you have a bug. The second test input is called `9nodes.in`. The correct output is written in `9nodes.out`. You can do the test in a similar way.

This is the minimal amount of tests you should run to check your program. You should also write other different test cases yourself to test your program extensively.

IX. Submitting and Due Date

You should submit all the related `.h` files, `.cpp` files, and the `Makefile`. The `Makefile` compiles a program named `p2`. The submission deadline is 11:59 pm on July 6th, 2016. You should put all the files into a single `.tar` file and submit the `.tar` file through the assignment link on Sakai. You should name the `.tar` file with your last name and first name as `LastName_FirstName.tar`.

X. Grading

Your program will be graded through a number of test cases of different sizes. Each test case is scored with 2 criteria:

1. **[20%] Structural correctness.** Did you lose any gates? Are they all inside the 100x100 chip area? (Yes, this sounds a bit simple – but if you set up the matrix solve incorrectly,

your gates can go in very strange places. Be careful to check this!) Having ALL the gates present earns you 10%, and having them ALL placed inside the chip area earns you another 10%.

2. **[80%] Similarity to our solution**. If you follow the procedure and the rules specified in this document, there is only one correct answer. However, due to numerical difference, your answer may be slightly different. We will measure the similarity between your answer and our answer using the Sum of the Absolute Difference (SAD). We will set a threshold: if the SAD is smaller than the threshold, you answer is treated as correct.