VLSI CAD: LOGIC TO LAYOUT:
Programming Assignment 3: A Quadratic Placer

Rob A. Rutenbar
University of Illinois at Champaign-Urbana

In lecture, we described how the quadratic wirelength model allows us to build an analytical-style placer that uses large matrix solves to optimize each placement. This placer will also 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 programming assignment, you get to build a version of a *real* quadratic placer, and run on it some *real* industrial benchmarks. The assignment has two parts:

- **Required**: Build a core **"3QP"** placer: execute one vertical cut, and then one containment step on each side of that cut. This will require exactly 3 "QP" solves, and requires you to demonstrate most of the technical steps that comprise a real placer.
- **Optional Extra Credit:** Build a deep **"8x8"** placer that runs recursive partitioning down to an 8x8 grid of 64 partitions, placing rough 1/64th of the gates in each partition.

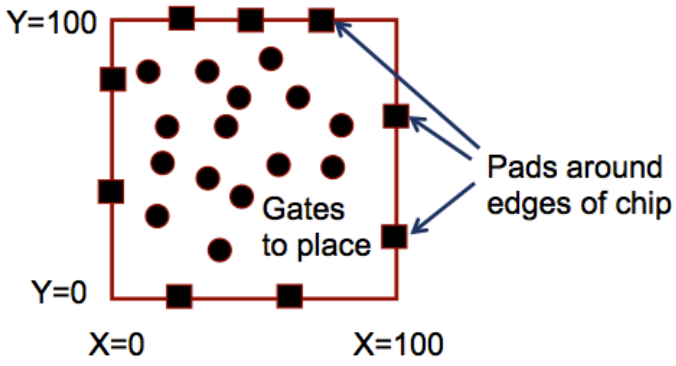
You can write this program in any language you choose. We will provide netlist inputs as simple textfiles. You will output your placement result as a simple textfile in a fixed format. You upload your results to the Coursera site and our auto-grader will score your results. We will also provide simple matrix solvers for four languages: C++, Java, Python and Matlab/Octave.

[100 points] Core 3QP Placer

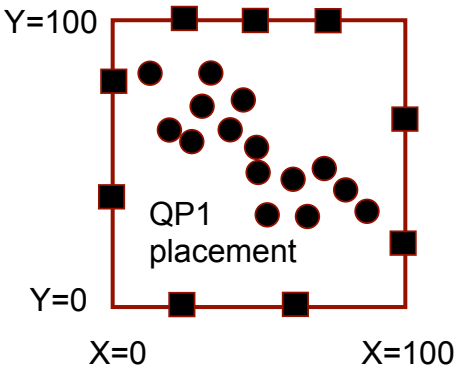
The 3QP placer requires you to execute exactly 3 Quadratic Placement (QP) matrix solves. One solve on the full-size netlist, and then one solve on the gates assigned to the left half of the chip surface, and then one solve to the gates assigned to the right half of the chip surface. This will require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

Here is the precise set of steps we want you to implement, with some diagrams to clarify what we need you to:

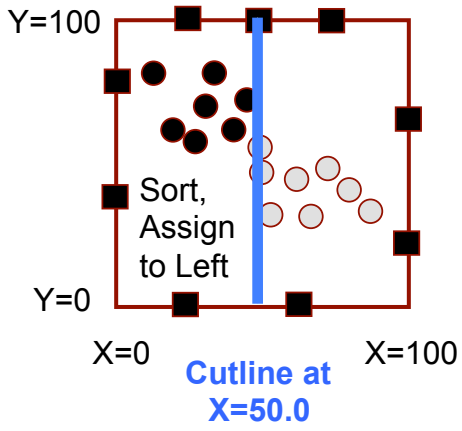
1. **Input**: you read in a netlist in a simple textfile format. The netlist specifies how many gates and defines the nets that connect these gates. The nets can have any number of gates to which they connect. The file also specifies a set of input output (I/O) pads, which have fixed locations around the edges of the chip. The chip itself is always a square, with dimension in both X and Y going from 0 to 100.



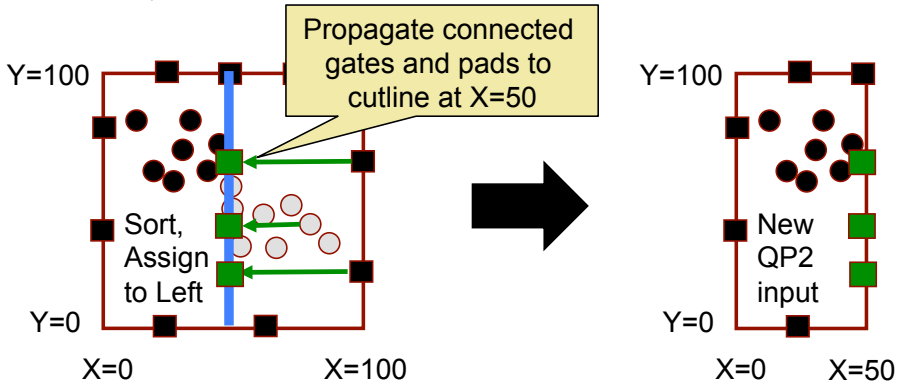
2. **QP1—First Quadratic Placement:** You set up the **A** matrix and the $\mathbf{b}_x$ and $\mathbf{b}_y$ vectors to solve the first quadratic placement. (This is, of course, 2 matrix solves: one for X, and one for Y). The result is a placement of the gates inside the 100x100 square that defines the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



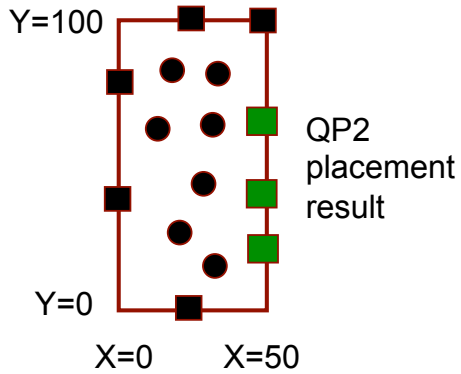
3. **Assignment**: Sort the QP1 placed gates to decide which go on the left, and which go on the right, for first vertical partitioning cut. This means: you have 2 sort keys, first is X coordinate for the point, and then if there are any ties, next is Y coordinate for the point. (Hint: If you just sort on some single numerical key like (**100,000*X + Y**), this will all work easily. As long as there are fewer than 100K gates.) **Remember**: 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 sorting/assignment step is critical. Also, if the number of gates is **odd**, you cannot have exactly the same number of gates on each side. Put smaller number of gates on the **left**. Example: if you have 101 gates to assign, put 50 gates on the left. If you have 103 gates, put 51 gates on the left.



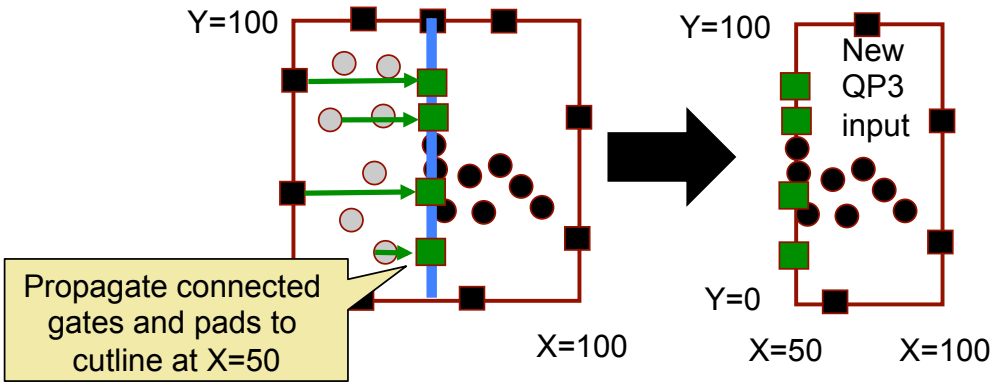
4. **Left-side containment step:** Propagate the gates and pads to which these left-side gates are connected, and place them on the centerline of the partition. This just means: find the right-side gates/pads that have nets connected to the left-side gates, and represent them now as having their same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



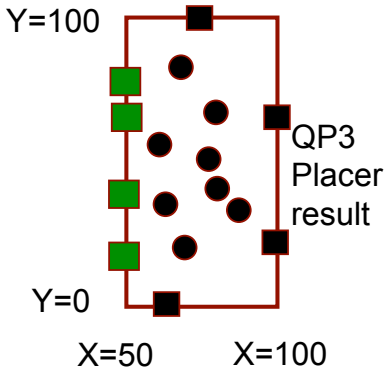
5. **QP2—Second Quadratic Placement:** solve the new QP2 placement problem. This means: set up the new **A** matrix (note: it has ~ ½ of the gates, the ones assigned to the left) and the 2 new **b** vectors. Solve it, for new X and Y. This locates these gates to minimize their quadratic wirelength *inside*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



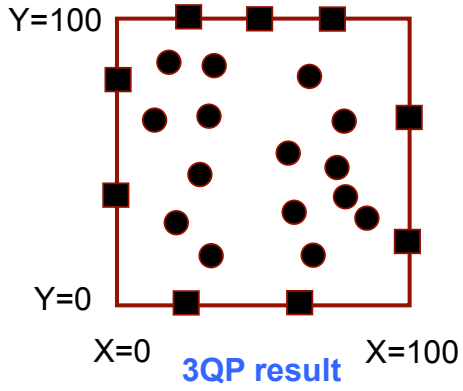
6. **Right-side containment step:** Now, we need to place the right side gates. So, first, we propagate the connected gates and pads on the left side, **using their new placement locations** (!), to define the new right side placement problem.



7. **QP3—Third Quadratic Placement:** solve the new QP3 placement problem. This means: set up the new **A** matrix (note: it again has about ½ of the gates, the ones assigned to the right) and 2 new **b** vectors. Solve it. This locates these gates to minimize their quadratic wirelength *inside* this right partition.



8. **Output 3QP result:** tell us the (X, Y) coordinates of all the gates in a simple textfile. This is the full result for the 3QP placer: a complete vertical partitioning cut, assignment of gates to each side, a new containment step QP solve to locate the gates properly on each side.



..and, that is what we want you to do for this Programming Assignment.

Core 3QP Placer: Discussion

Why are we asking for this specific set of QP steps? Because this is basically the "core" of any real quadratic, partitioning-based placer. You *read* a real netlist; you *solve* the QP problem with a matrix solver; you *recursively partition* and solve the "smaller problems" that result; you deal with the problems of *assignment* (what gates go where?) and *containment* (how do I reformulate the QP to keep these gates inside this region?).

It's small, but it's really *most* of the essential core of a realistic QP engine.

Note: Our reference 3QP solver has about **700** lines of C++ code, and it has lots of comments.

Core 3QP Placer Logistics

Input file format: We use a simple text format to specify each placement problem. Here is the complete listing of our smallest example benchmark, with some embedded comments:

```
18 20           // NumberofGates G and NumberofNets N
1 2 20 19       // Gate# NumNetsConnected NetNumber...
2 2 2 1         // Example: Gate#2 is connected to two nets: #2 & #1
3 2 1 18
4 3 3 4 5
5 6 6 5 4 3 7 8
6 2 9 4
7 2 10 3
8 2 19 9
9 2 9 10
10 4 11 10 12 13
11 2 12 16
12 2 7 6
13 2 12 11
14 2 14 17
15 2 8 15
16 5 15 16 17 18 7
17 4 16 17 18 14
18 5 13 16 17 18 12
6              // Number of pads P connected to nets
1 12 25 0      // PinID NetNumberConnectedTo PinX PinY
2 17 50 100    // Ex: Pin #2 connected to Net #17 at (X,Y)=(50,100)
3 18 100 50
4 2 0 50        // NOTE: The above comments will NOT
5 19 0 25       // be in the REAL input files!
6 20 75 100
```

Here is how the textfile input format works:

- **First line:** two integers. The number of gates G and number of nets N
- **Next G lines:** one line per gate. Each line has a Gate ID number (starting from 1 and continuing in order without any gaps to gate #G), then the number M of nets connected to this gate, then the Net IDs of the M nets this gate is connected to.
- **Next line:** Number of pads P on this chip.
- **Next P lines:** one line per pad. Each line has a Pad ID (starting at 1, and continuing in order without any gaps to pad #P), then the Net ID this pad is connected to, then an X coordinate and a Y coordinate.

Note: the weights of nets are not specified. So just assume all of them are 1.

Output file format: We use a simple text format to specify each placement output. Here is how the textfile output format works:

- If you have **n** gates, you write out **n** lines. Each line looks like:

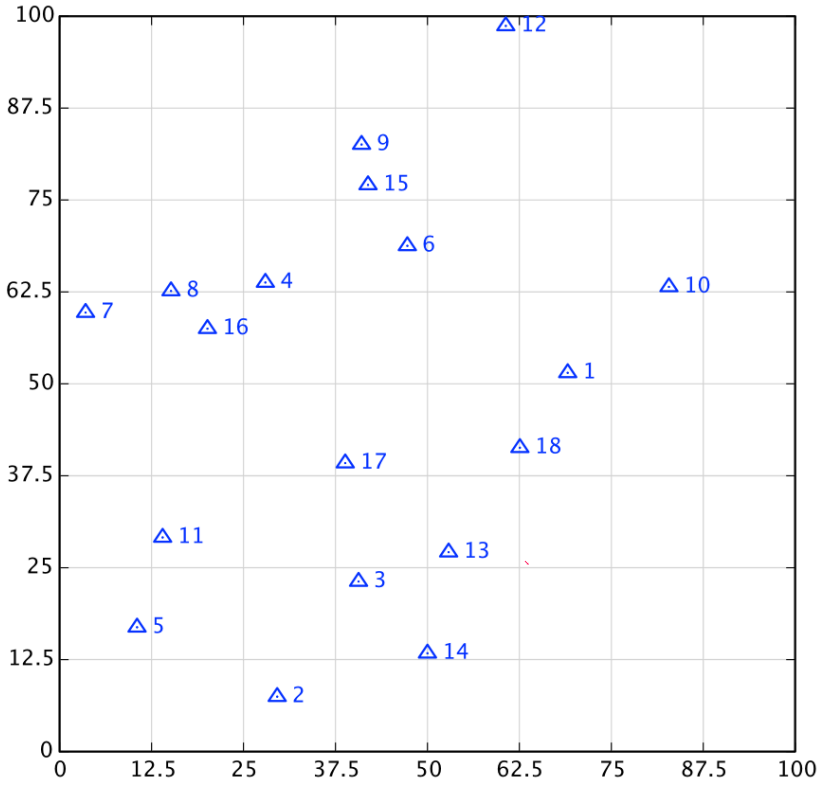
  GateID floatXcoordinate floatYcoordinate

- **floatXcoordinate** and **floatYcoordinate** denote the x and y coordinates of the gate, in floating numbers. Please output **8 digits of precision**.
- One gate per line, starting from gate #1.

For example, the following is a random placement output for benchmark "toy1", which contains 18 gates:

```
 1 69.07240213 51.50161048
 2 29.57983185  7.46039685
 3 40.64633508 23.12410747
 4 27.97125898 63.78669305
 5 10.53030157 16.89692394
 6 47.28134736 68.76447585
 7  3.54120030 59.69847626
 8 15.14752434 62.63403985
 9 41.05436997 82.53633055
10 82.83944283 63.19109279
11 14.01284420 29.11834908
12 60.66313090 98.67133022
13 52.88732594 27.11454812
14 50.00454944 13.37096091
15 41.92590214 77.04902278
16 20.09629265 57.50068350
17 38.83612814 39.22482778
18 62.57583558 41.32020682
```

This is what it looks like if we draw a figure according to the output:



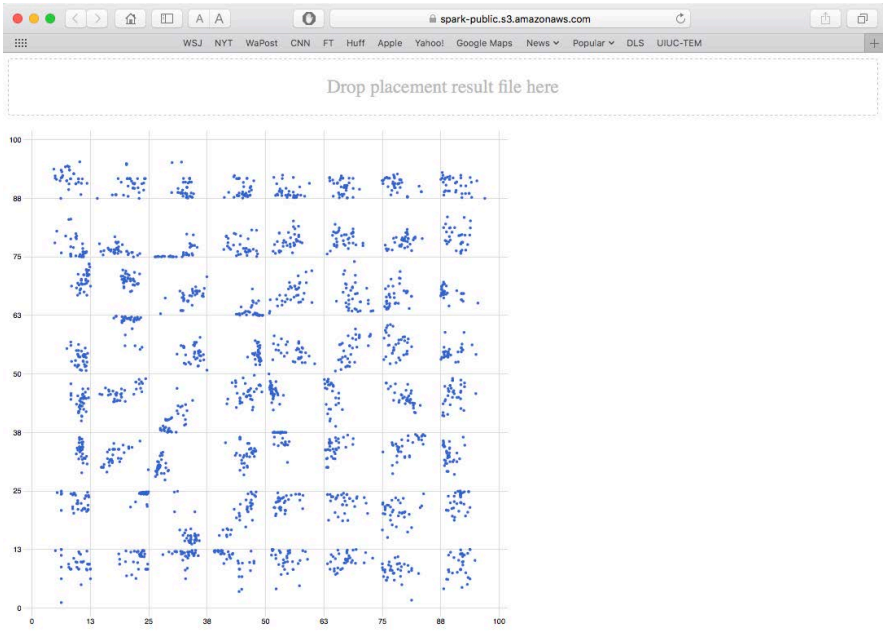
Matrix Solver Code: You need to be able to solve a large matrix to complete this assignment. So, we are providing you with a few options. We want to emphasize: you can solve the matrix *any way you want*. You can use one of our options, or use a different option. You can write this code in any language you want. But, because of the way we have specified the problem: *there is a right answer*. We know what to expect for the output, so you need to solve the matrix correctly. Here are some options:

- **C++ Source Code Conjugate Gradient Solver**: We will provide you with source code for a simple iterative matrix solver, and a small example of using the subroutines, to get you started. Conjugate Gradient (CG) methods are famous iterative methods that are very efficient for these types of systems of linear equations. 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.

- **Java Source Code Conjugate Gradient Solver:** Same as the C++ code, just written in Java, if you like Java better.
- **PYTHON linear solver:** NumPy and SciPy are excellent numerical computation packages. They provide convenient classes such as sparse matrix and optimization routines. We will provide source code to demonstrate how to use NumPy and Scipy to solve linear systems. Note: you need to install the packages using your favorite package tools, such as *pip*.
- **MATLAB / OCTAVE:** In the engineering and science domain, MATLAB is a very popular commercial tool from Mathworks, Inc. If you have access to MATLAB, feel free to write the code using it, since MATLAB has very powerful matrix solve capabilities built in, and you can do this in a single line of code. OCTAVE is a "public domain" version of MATLAB, with a similar syntax and similar (but smaller) set of capabilities. But for linear solvers, OCTAVE is also capable of doing the job. OCTAVE is also freely and widely available on many platforms; lots of MOOCs use OCTAVE for their programming assignments.

You can find the source code for the four languages above in the accompanying starter files for this assignment.

Visualization: Nobody builds layout tools without being able to *see the results*. Since this is an essential part of this effort, we have proved some very nice visualization tools for this assignment. You can take your textfile output, and visualize the results immediately in your browser, by following these simple instructions:



1) Go to this webpage to run the placer visualizer:
https://spark-public.s3.amazonaws.com/vlsicad/javascript_tools/visualize.html
2) *Drag and drop* your placer textfile output into the "Drop placement result file here" section on the top of the page.
3) After a few seconds, you should be able to see the results of your placement in the middle of the page.

The course website also has a short video tutorial about this visualizer tool
(and also, for the follow-on router project for the course:
https://www.coursera.org/learn/vlsi-cad-layout/lecture/h9eNy/tools-tutorial

The visualization uses the Google Charts API (specifically, the Scatter Chart) to visualize the results.

Do this:

- **Build** a **3QP** placer. You read out placement input file format, you write out placement file format.
- **Run** our benchmarks. We will provide you with 5 separate benchmarks, of increasing size.
- **Upload** your results to our Coursera website, and our autograder will tell you how you did.

3QP Grading Details

There are 5 benchmarks. Each is worth 20 points. Total assignment is worth 100 points.

- **[20 points] Toy1:** 18 gates, 20 nets, 6 pads.
- **[20 points] Toy2:** 32 gates, 42 nets, 10 pads.
- **[20 points] Fract:** 125 gates, 147 nets, 24 pads.
- **[20 points] Primary1:** 752 gates, 902 nets, 107 pads.
- **[20 points] Struct:** 1888 gates, 1920 nets, 64 pads.

Each benchmark is scored with respect to 3 criteria:

- **[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, you 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%.

- **[40%] HPWL Wirelength**: While we minimize quadratic wirelength (QL) in the 3QP placer, we will evaluate your solution quality based on HPWL. HPWL is a true good prediction of final layout quality, while QL is an approximation of HPWL. Plus**,** HPWL is easy to compute (for **you**, as well as for **us**) and it is widely used. **We** measure the total HPWL wirelength of your placer, and we compare it to the HPWL wirelength of our version of the 3QP placer. Your goal is to be "close" to this answer. We have a *very* generous definition of "close". You will get full points if your solution is no more than a certain percentage of our reference placer. Otherwise, points will be given on a logistic curve (up-side-down), based on how "far" your HPWL is from ours.
- **[40%] Balanced Placement:** 3QP should put half the gates on the left of the chip surface, and half the gates on the right. We check this. Again, we have a rather generous notion of "balanced" so you can be off a little bit (i.e., a few gates). But, since our recipe for what 3QP solves is quite precise, there really is a right answer, and we want to see how close you are.

There is some partial credit available for each part, depending on how far away from the "right" answer your solution is.

3QP Submission:

- You can submit as many times as you like.

Acknowledgements:

- The benchmark set is originally from the Microelectronics Center of North Carolina (MCNC) from the early 1990s. They are referred to in publications from that era as "**the MCNC benchmarks**". We have made some small modifications to these to make them suitable for our use in this MOOC.
- Thanks to **Prof Luke Olson** of the Department of Computer Science at the University of Illinois at Urbana-Champaign for the C++ version of the CG iterative matrix solver we provide for this assignment.
- Thanks for TA **Zigang (Ivan) Xiao** for the reference 3QP and 8x8 placers and the autograders.
- Thanks to TA **Nicholas Chen** for the Java port of the matrix solver, and the visualization drag-and-drop function, and the integration of our auto-graders.

[25 points] EXTRA CREDIT: 8x8 QP Placer

Since we saw some significant enthusiasm for programming in general, as part of the **Programming Assignment 1**, the URP Complement code, we have chosen to offer another part to this assignment. This is **extra credit**. You can get a maximum of **25** additional points.

This part is *optional*. This will *not* change any "class grading curve".

8x8 QP Placer

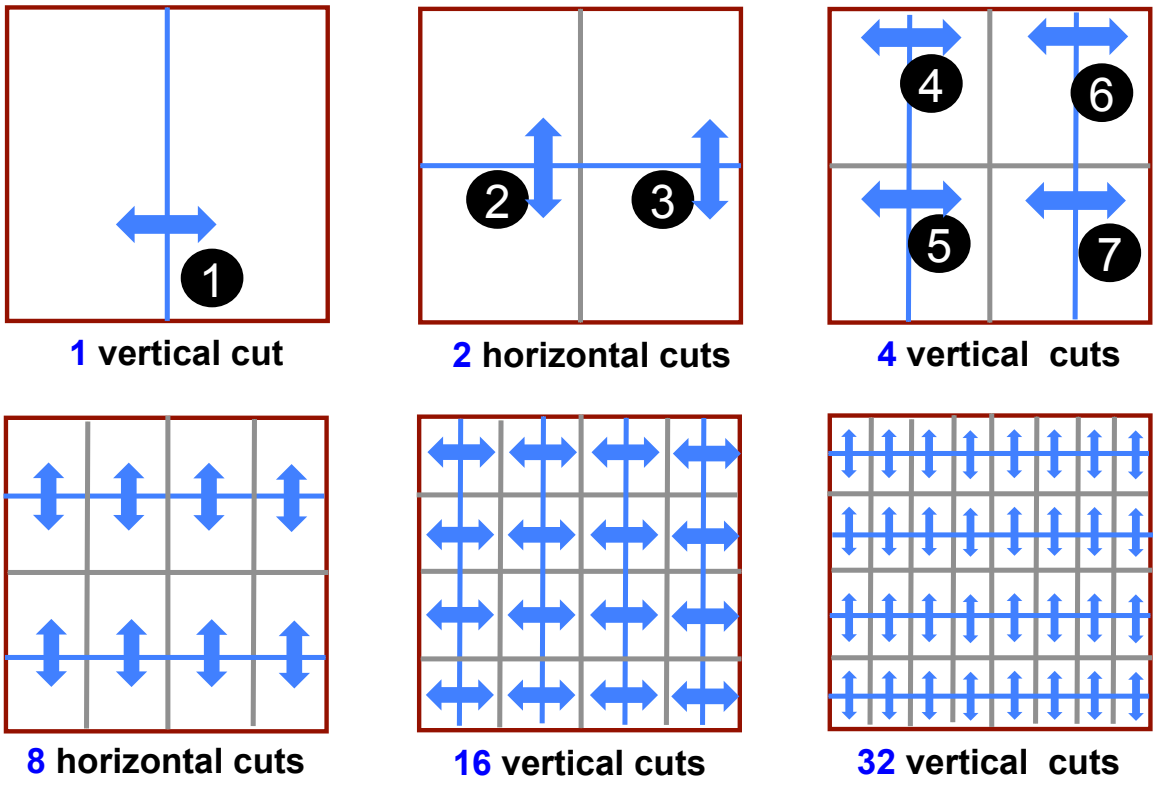
The **3QP** placer performs exactly 1 real partitioning cut. This involves: 1 initial QP matric solve, and then the assignment and containment problems on each side of the cut, which involve 1 sorting operation and 2 additional QP solves. This is the core of any real QP engine.

But this is *not* a real QP engine: it does *not* go deep, it does *not* perform both vertical and horizontal cuts. For the **8x8** QP placer, we ask you to build the *full* QP engine, and to run it with vertical and horizontal cuts recursively down to a level of partitioning in which you have 8x8=64 partitioned regions, each with about 1/64 of the total gates inside.

Note: This is *not* a lot more code. Our reference 8x8 solver has about **750** lines of C++ code, and it has lots of comments. The nice thing about the recursive partitioning QP strategy is – are we surprised? -- that it's recursive. Once you have the core recursion step working, you can run it on any size benchmark, and any number of partitioning steps (i.e., 8x8, or more, such as 16x16, 32x32, etc.).

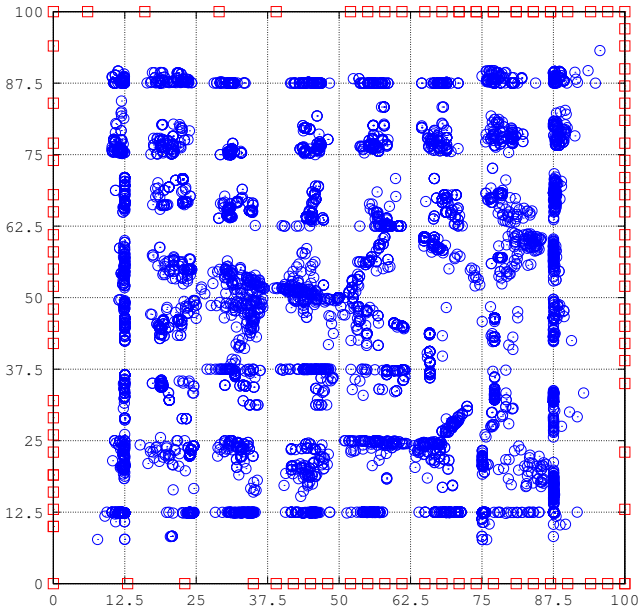
To be clear – the diagram on the following page shows what we are looking for, in terms of partitioning: a sequence of alternating vertical and horizontal cuts, performed until the chip is horizontally and vertically divided into 8x8=64 regions. Also shown on that page is an example of one of our benchmarks, placed with our reference 8x8 QP code, showing what you might expect as a final result. As always, the placement in each region of the 8x8 partition can be very, very unbalanced.

We will provide a set of bigger, more interesting industrial benchmarks to use for this extra credit part of the assignment.



8x8 QP placer performs this set of cuts, to yield 8x8 partition of placed chip

Note: This diagram shows the cuts at each level. If you do it recursively, the cut sequence will be 'depth-first', i.e., for each square do one vertical and two horizontal cuts. You get four smaller squares, and you recurse at each of them.

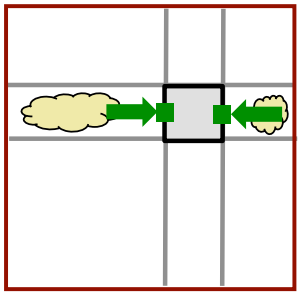


Example of one of our benchmarks placed with our "reference" 8x8 QP placer

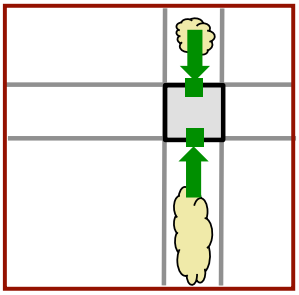
8x8 QP Logistics

The 3Qp placer is a big part of the core inner loop of the 8x8 placer. But there are a few key new pieces you need for this extra credit.

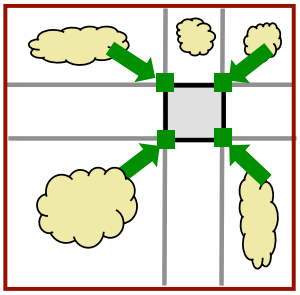
Containment and gate/pad propagation: In the 3QP placer, the propagation step was very easy: you located the objects on the "other side" of the cut that were connected, you push them to the cutline, and their new coordinate was just ($X=50$, $Y=Y_{old}$). In the 8x8 placer, you have to propagate from all sides of the region being contained, and you must really compute where the propagated gates and pad will be located. The following figure is instructive.



Gates located **left/right** of QP region propagate to **left/right edges** of region



Gates located **above/below** the QP region propagate to **top/bottom edges** of region



Gates located **diagonally** from QP region propagate to **4 corners** of region

In the diagram, the small gray region has gates inside, and we are propagating connected gates and pads located outside the region, to the boundary of the region, so that the QP solve will correctly contain the gates. In the more general 8x8 case, the "gates outside" can be located anyway. The figure shows the three important cases:

- Gates located left and right of the region are propagated to the left/right edges of the region. In other words, they keep their Y coordinate, and the X coordinate is replaced by the left/right X of the new region.
- Gates located above and below the region are propagated to the top/bottom edges of the region. In other words, they keep their X coordinate, and the Y coordinate is replaced by the top/bottom Y of the new region.
- Gates not falling in one of these two simple cases are "diagonally adjacent" to the region. They are both left/right, and above/below the region. So, the standard solution is we propagate them to the **nearest corner.** This is the important case that happens a *lot* in a real QP placer.

Wirelength and overall chip balance: One important thing to note is that if you run one of these big benchmarks through your **3QP** placer, and then run it through your **8x8** placer, the wirelength will *increase*. Likely it will increase by a *lot*. Why? Because your 8x8 placer forces the gates to be located roughly uniformly across the 8x8 regions of the chip, making the wires longer.

So, there is an important point to note: when we ask you to run the 8x8 placer, it means we want you to **stop** when you get to 8x8=64 regions. If you run it further, to a finer partition, your answer will **not match** the auto-grader's expected solution. So, this is a case where we want to do "just enough" placement, and no more!

8x8 Grading Details

There are 3 new benchmarks. Each is worth 10 points. You can get a maximum of 25 points from the 30 points available from these 3 test cases.

- **[10 points] Industry1:** 2271 gates, 2478 nets, 490 pads.
- **[10 points] Biomed:** 6417 gates, 5742 nets, 97 pads.
- **[10 points] Industry2:** 12142 gates, 13419 nets, 495 pads.

Each benchmark is scored with respect to 3 criteria:

- **[20%] Structural correctness:** Same as for **3QP**. Did you lose any gates? Are they all inside the 100x100 chip area?
- **[40%] HPWL Wirelength:** We again measure the total HPWL wirelength of your placer, and we compare it to the HPWL wirelength of our version of the **8x8** placer. Your goal is to be "close" to this answer. And again, we have a generous definition of "close".
- **[40%] Balanced Placement:** This is *new* and more precise than **3QP**. We look at each of the 8 columns and 8 rows of the 8x8 placement, and check that you have roughly 1/8 of the gates in each row/column. It turns out this is robust with respect to some of the low-level numerical issues of where the matrix solve chooses to put the precise gate locations. There is a "yes/no" decision checked for each of the 8 columns and 8 rows, and we are (again) generous in allowing you to be "close" to the right balance. So, you can get a score from 0/16 to 16/16 (multiplied by 40%) on this part.

Again, there is some partial credit available for each part, depending on how far away from the "right" answer your solution is.

8x8 Submission:

- You can submit as many times as you like.