CMU Fall 2001 18-760 VLSI CAD

[120 pts] HW 4. Out Tue Oct 30, Due TBD, November (V1)

1. Rectangle covering [20 pts]

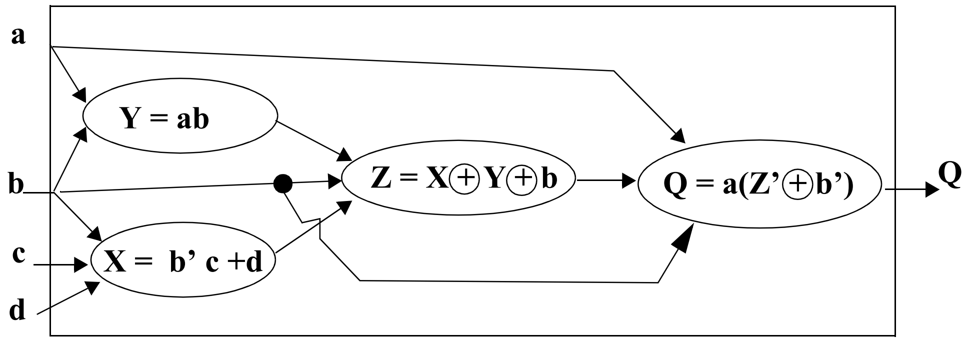
Consider this simple function: **Y = abg + acg + adf + aef + afg + bd + be + cd + ce**

Do this:

- Kernel this function using the recursive algorithm from class. Show the recursion tree as in the class notes.
- Next, build the cokernel-cube matrix for Y.
- Using rectangle covering, identify the best first multi-cube factor you can pull out the expression for Y and improve the network. (Note the "network" here starts with just one Boolean node. This is OK, all this stuff still works fine.) Show the updated network with the factor extracted.
- Repeat: show how to update the matrix to reflect this factor, and then show how to use rectangle covering to get the next best factor (**if** there **is** a next best factor that makes the network better). If there is, show it, and redraw the network to reflect this second factor that you found. If there is no "next factor" that improves the network, say why. (You don't have to update the matrix again.)

2. Multi-Level Don't Cares [20 pts]

Consider this Boolean Logic Network.



Do this:

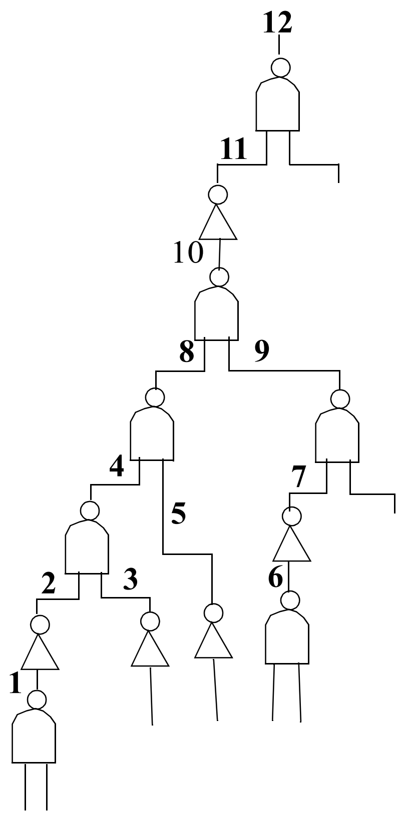
1. Derive the **Satisfiability** don't cares (SDCs) for node X and Y. Show an SOP cover of the don't care functions SDC_X and SDC_Y

2. Derive the **Controllability** don't cares (CDC) for the signals at the input to node Z. Show an SOP cover of the don't care function. Show how you use the formula to quantify away any unneeded variables.
3. Derive the **Observability** don't cares for node Z with respect to network output Q. Show how you use the ODC formula to quantify away any unneeded variables.
4. Draw a **Kmap** for node Z. Show the original "1"s in the Z function, and show all the new don't cares due to CDC and ODC don't cares. Minimize the 2-level form for the function at Z using these new don't cares and write a new SOP form for Z. Is the new form of Z any simpler?

Suggestion: Don't hesitate to use the KBDD calculator to do the quantifications and such here -- use it as a Boolean *calculator* to do the messy parts of this problem, and hand in any output from this as proof of your work. You don't need to do it all by hand if you don't want to.

3. Technology Mapping [20 pts]

Consider this uncommitted logic network made out of inverters and 2-input NANDs. We have labeled the internal nodes of the network with numbers 1-12 for convenience.



Suppose your technology library consists of:

1 input gates:	INVERTER ($1)
2 input gates:	NAND ($2), AND ($3), NOR ($2)
3 input gates:	NAND ($3), OAI21 ($3) (or-and-invert, it implements [(a+b)•c]')

Do this:

- First, **draw** the target patterns for each of these library cells, as they consist of NAND2s and INVERTERs.
- **Label** which library patterns match at each node of this network (you can do this by eye).
- Then, **show** how the dynamic programming tree covering algorithm will derive a minimum cost covering of this network (**show** the tables like in the notes; you don't have to manually do all the recursion stuff).
- **Draw** the final mapped network, and tell its final cost.

  And: **no** you **don't** have to do the back-to-back inverter insertion trick from the notes.

4. Using SIS to do Tech Mapping [20 pts]

Let's see how this works with a real techmapping tool. First, you have to describe the technology library in the **GENLIB** file format and then use this library for the technology mapping of the given circuit. For help on the GENLIB format, look at the assignments page on the class web page, a pdf about this is there.

Use 1 for <input-load>, 999 for <max-load>, 1.0 for all block delays, and 0.0 for all fanout delays. An example technology library, which consists of only an inverter and a 2-input AND gate, is described in the GENLIB format as:

```
GATE zero  0  O=CONST0;
GATE one   0  O=CONST1;
GATE inv   1  O=!a; PIN * INV 1 999 1.0 0.0 1.0 0.0
GATE and2  2  O=(a*b); PIN * INV 1 999 1.0 0.0 1.0 0.0
```

Then define the network to be covered in the **BLIF** file format, using one **`.names`** directive for each gate. Documentation on BLIF format can also be found on the class web page.

A BLIF description of a netlist which consists of a 2-input AND gate, followed by an inverter, is:

```
.model circuit.blif
.inputs a b
.outputs f

.names a b n1
11 1
```

```
.names n1 f
1 0

.end
```

Finally, assuming you have **circuit.blif** and **tech_libl.genlib** typed in as the netlist and tech files, respectively, in your UNIX directory, you run SIS, and the sequence of SIS commands that you will need is:

```
sis>rl circuit.blif
sis>rlib tech_lib1.genlib
sis>map
sis>p
sis>print_gate
```

Do this:

- Set up the problem just as in problem 3, and run the mapping. **Run** SIS, and handin the **printout**, and also **draw** the network it creates. **Interpret** this output: write a few sentences explaining what SIS did.
- Let's try a much richer library, and see if it helps as all. Use the genlib format to build this library:
  INVERTER: 1 input
  NAND: 2,3,4 inputs
  AND: 2,3,4 inputs
  NOR: 2,3,4 inputs
  OR: 2,3,4 inputs
  XOR: 2,3,4 inputs
  XNOR 2,3,4 inputs
  AOI: (2,1) as in the class lecture notes
  OAI: (2,1) as in the class lecture notes
  Run SIS, and **handin** the printout, and also **draw** the network it creates. **Interpret** this output: write a few sentences explaining what SIS did.
- Let's try a much weirder library, and see if this works at all. Use the genlib format to build this library:
  INVERTER: 1 input
  NOR: 2,3,4 inputs
  XOR: 2,3,4 inputs
  XNOR 2,3,4 inputs
  Run SIS, and **handin** the printout, and also **draw** the network it creates. **Interpret** this output: write a few sentences explaining what SIS did.

5. Annealing and the Travelling Salesman Problem (TSP) [10 pts]

Source code for the TSP annealer example is in the class ANDREW account. This implementation is pretty simple, and it works OK--but it could be better. In particular, the one obvious feature that the code lacks is any notion of "**range limiting**". Recall from the class notes that this is the idea that you should try to limit the “range” of proposed random moves to be smaller as the temperature cools, so that you avoid trying useless moves. In the context of TSP, its pretty clear that when its really cold, it’s a bad idea to try swapping completely random, distant cities that are already separated by many links in the overall TSP solution tour. But...it takes some thought to work out a real range limiter here.

Do this:

- Show **pseudocode** for a proposed range limiting solution. Tell where in the current TSP code this would go, eg, is it in evalSwap(), or in annealAtTemperature() or where? Explain what you think your range limiting scheme is supposed to do.

(**Hint**: think about the structure of the tour as it evolves. When it is hot, cities i, j in the tour can be very very far apart, even if i is very close to j in the tour order. As it cools, however, this should not be true any more: if i is close to j, then probably city i is closer to city j. Can you use this to suggest a way to pick cities to swap are likely to yield more useful moves? It is OK to assume some un-specificed function RANGE(Temperature) that gets tuned to limit the actual numerical “rangle” limit, as a function of the falling temperature. In any real annealing, this is something that gets carefully tuned, or set adaptively as annealing proceeds. But for this problem, we want to see you work out the mechanics of how you would use this info, i.e., what is RANGE(Temperature) actually generating? Is is a distance (in miles?) a separation (in index-units in the tour array solution?) or something else. You can specify anything you like here, just explain it clearly.

You might also want to go look at placer.cc, which is also on the web page, and is a simple checkerboard-style gridded placer. It has a real range limiter implemented for a placement problem.)

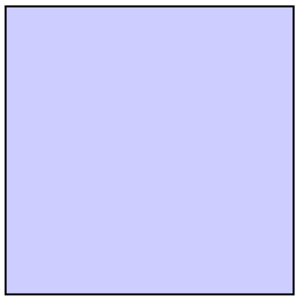
You **don’t** have to go implement this in the tsp.cc code. (But if you do, we will be mighty impressed...)

6. Very Simple Floorplanning via Annealing (TSP) [30 pts]

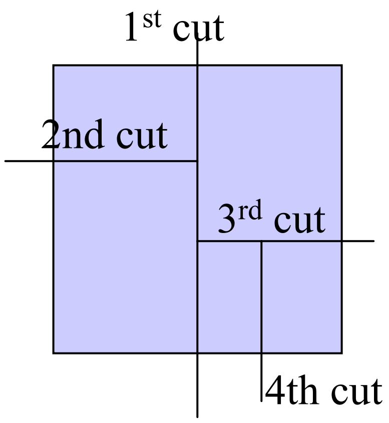
OK: this one, you have to **implement** something. But, something small, and interesting.

In class we showed a video tape of something sort of like a jigsaw puzzle assembling itself by annealing. (This was that fuzzy orange set of rectangles, with appropriate musical accompaniment.) You can actually build a very simple version of this, directly reusing most of the skeleton of the TSP annealing code.

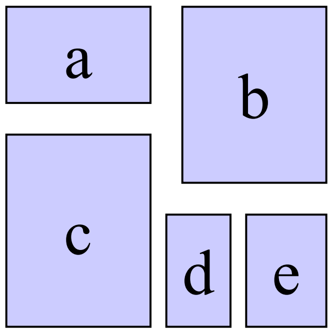
In general, this is a lot of work, but for this problem, we’re going to build the code to just one single, fixed problem, i.e., we will not read in any description of a general floorplan. We will just “bake” the floorplan right into the cost function and move set, so that we keep this very very simple to implement. Here is how to do it:



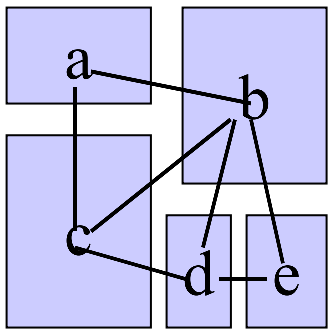
1. Get some graph paper, draw a big square.



2. Draw some “cut” lines thru the square, recursively like this.



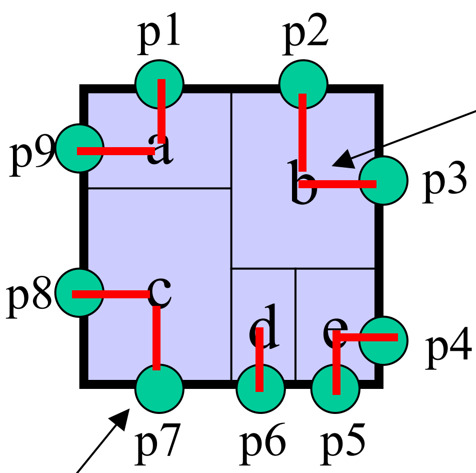
3. This is your trivial “jigsaw” puzzle, and you know it can be reassembled into a perfect square. Label the pieces.



4. Draw “wires” from center of each piece to other pieces it touches. Note – not all pieces will end up with these wire, since not all piece touch each other

	a	b	c	d	e
a		1	1		
b	1		1	1	1
c	1	1		1	
d		1	1		1
e		1		1	

Make a matrix C(i,j) that records which blocks have wires to which other blocks

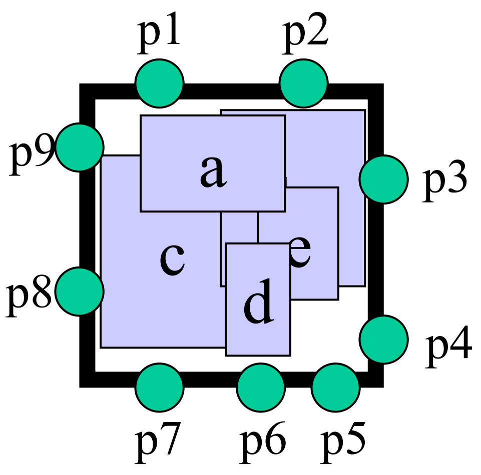


Create another wire
from each pin to the
center of the block it
touches. Note, each pin
is on just one wire.

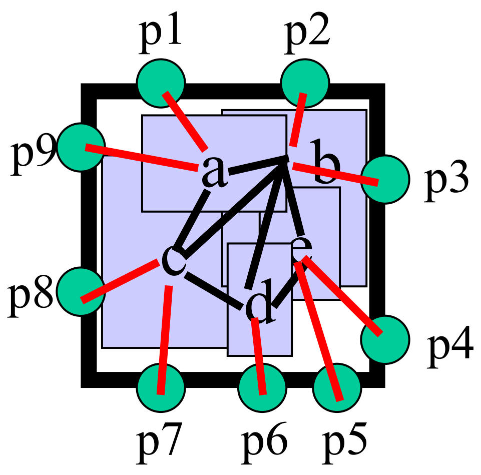
p1 (x1,y1,a)
p2 (x2,y2,b)
p3 (x3,y3,b)
p4 (x4,y4,e)
p5 (x5,y5,e)
p6 (x6,y6,d)
p7 (x7,y7,c)
p8 (x8,y8,c)
p9 (x9,y9,a)

5. With the puzzle "assembled", draw a "pin" on the perimeter with one same coord as the center of each block that touches the perimeter. In this small example, its all the blocks. It general, it won't be all the blocks.

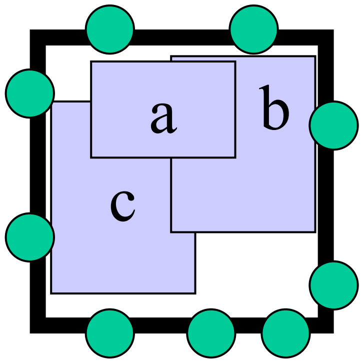
6. Create an array that lists each pin, its FIXED location on the perimeter of the square, and what block it connects to.

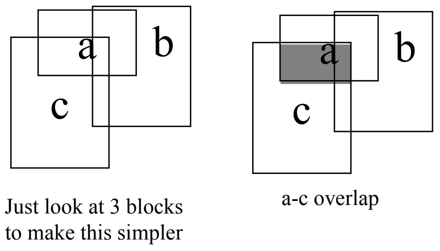


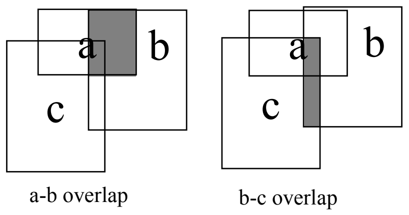
7. To anneal: just randomly relocate a block inside the overall square; these are your **moves**.
Yes—just let them overlap!
Your **problem state** is just the (X,Y) center location of each block.



8. The first part of your **cost function** is just the **lengths** of all the wires: The ones block-to-block, and the ones pin-to-block. Remember: blocks move, but pins don't.

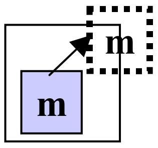






9. The second part of your **cost function** encourages the block **NOT** to overlap. This is called a **"penalty function".** For each pair of blocks that overlap, you add to the cost function the amount $(\textit{area of overlap})^2$

Here is the overall **pseudocode** for how to do this move-one-block-and-eval to anneal this problem:



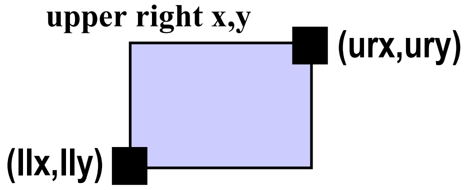
```
// to do a move
let m = random block in {a,b,c,d,e}
let x,y be a random new center location in the overall square
move block m to location (x,y)
// you can either let the blocks move partially "outside" the square
// or check when this happens, and slide them back inside.  Your call.

// to eval delta-cost for a move
newCost = 0;
// cost for wires from pins to blocks
for (pin p = 1; p<=9; p++)
   newCost += distanceFromPinToItsBlockCenter(p)
// cost of wires and overlaps block to block
for( block m = a;  m <=e;  m++) {
   for(block n = m+1; n<e;  n++) {
      // cost for block to block wires
      if (C[m][n] == 1)  //blocks m,n have a wire we must count
         newCost += distanceBetweenBlockCenters(m,n)
      // penalty for block to block overlap
      newCost +=  WEIGHT*overlapAreaBlocktoBlock(m,n)**2
   }
}
deltaCost = oldCost – newCost;
```

Do this:

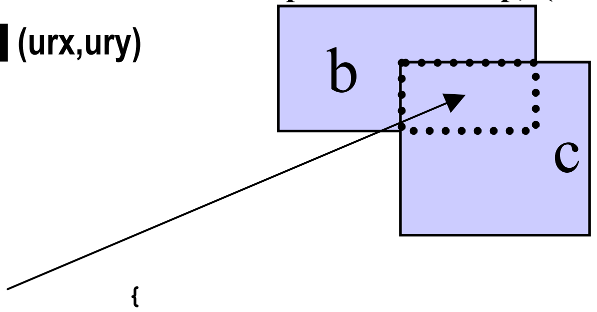
- Make up your own jigsaw-puzzle “floorplanning” problem as shown here. Use at least 10 blocks. We suggest a 100x100 grid for the overall square.
- Use the tsp.cc code as a starting point, but remove the “TSP” parts, and replace them with code to do the jigsaw puzzle. In particular, this is
  genTSP() -- generates a random TSP problem, you want genFloorplan()
  costTSP() -- rename as costFloorplan, and write a new one
  tclInitTSP() -- just rename it to tclInitFloorplan()
  tclDrawTSP() -- change it to draw you blocks, wires, pins, instead of the TSP.
  tclEndTSP() -- just rename it to tclEndFloorplan()
  evalSwap() -- replace it with a new evalFloorplanMove()
  global variable “tourcost” -- rename it floorplanCost
  annealAtTemperature() -- set var “attempts” to MOVESPER * (your block count)
  replace the ~30 lines of code that swaps 2 cities with
  some code to pick a random block, move it, eval cost,
  accept reject. Its doable in about 20 lines.
  anneal() -- works OK as is as long as “tourcost” gets replaced with “floorplanCost”
  this gets set up globally before annealing starts
  main() -- get rid of any parameters you don’t want on the command line
- Your final cost will end up as:
  SUM(pin to block wirelengths) + SUM(block to block wire lengths)
  + **WEIGHT** * SUM(block-to-block overlaps)^{2}
  You have to play around with the “WEIGHT” number to balance the “attractive” part of the cost function, that pulls blocks with wires close to each other, with the “replulsive” part of the cost function, the push overlapping blocks apart. The idea is you want to minimize the wirelength (which should go to 0 in a perfect solution), and you want to minimize overlaps (which also should go to 0 in a perfect solution).
- See next page for how to compute these rectangle to rectangle overlaps easily.
- Play around with your implementation to set the annealing parameters to get something “interesting”. We don’t expect to see a perfect solution, with the floorplan reassembling itself quickly, perfectly, beautifully. This formulation is just a little to dumb, too simple. But, it should make some “progress” toward this final answer.

- Email asinghee@ece.cmu.edu a tcl movie file for the output, and in the HW assignment itself, print out the code, and tell us anything interesting you did to write it. See the class web page for the tcl drawing package cmuview, and instructions how to use it.

A rectangle is 4 coords:
lower-left x,y
upper right x,y



Want to compute how much rectangles b,c overlap, area of this potential overlap, (=0 of no overlap)



```
Overlap(b,c)            {
   rectangle a;
   //  try to build overlap rectangle itself - call it "a"
   a.llx = Max( b.llx, c.llx );
   a.urx = Min( b.urx, c.urx );
   a.lly = Max( b.lly, c.lly );
   a.ury = Min( b.ury, c.ury );

   if(  (a.llx > a.urx) || ( a.lly > a.ury) ) {
      // they don't really overlap
     return (0);
   }
   else return (  (a.urx - a.llx) * (a.ury - a.lly) )
}
```