DESIGN AUTOMATION

5MD20

local search
and partitioning

Ralph Otten
otten@ics.ele.tue.nl
February-March 2010
partitioning

- the basic idea:
  - divide the circuit into $n$ (typically 2) sub circuits recursively.
  - assign each sub circuit to a region.
- bad news: partitioning is NP-hard for most graphs
recursive partitioning: example

circuit

placement

cutline
a partitioning problem

definition:
given a graph $G(V,E)$
  a mapping $s: V \rightarrow \mathbb{R}_+$
  a mapping $w: E \rightarrow \mathbb{R}_+$

divide $V$ into $k$ subsets
that are mutually disjoint

such that
a cost function is minimized
under certain constraints

- the problem defined above is intractable
  - even with $k=2$, $A_i = S/2 + 1/2$, $s: V \rightarrow \{1\}$ and $w: E \rightarrow \{1\}$ (two-way balanced partitioning)
- settle for a "satisficing" solution:
  - satisfy the balancing criterion
  - approximate the minimum cost

example:
cost function:
$$\sum_{e \in F} w(e)$$

$$F = \left\{ e = (u,v) \mid \neg \exists V_i [u \in V_i \land v \in V_i] \right\}$$

constraints:
$$\forall 1 \leq i \leq k \left[ \sum_{v \in V_i} s(v) \leq A_i \right]$$
methods

- **move-based partitioning**
  - iterative improvement
  - stochastic methods

- **geometric representations**
  - hall partitioning
  - clusters to eigenvectors
  - probing
  - vector partitioning
  - order-based

- **combinatorial methods**
  - graph labelling
  - network flows
  - bipartite flow
  - probabilistic methods
  - mathematical programming
  - fuzzy methods
  - set covering

- **clustering**
  - agglomerative clustering, hierarchical strategies
  - boolean networks

- **integrating clustering and bipartitioning**
general structure for heuristic approach

- problem instance
  - constructive heuristic (can be a random partitioner)
    - iterative heuristic (well-known heuristics: kernighan-lin step (whether or not modified), fiduccia-mattheyses)
      - stopping criteria met?
        - stop; output best solution encountered so far
          - often first failure to improve
move based approaches

basis:
- configuration space: all partitions
- moves are transformations from one configuration to another
  - swapping two modules
  - moving a module over a cluster boundary
- configuration reachable within a move are "neighbours"

popularity:
- intuitive: improving a configuration by small changes
- simple to describe and "easy" to implement
- emanable to several general purpose techniques
  - annealing
  - genetic
  - tabu search
- no strict constraints on the objective function
graph partitioning: naive

- each time, try to move the vertex across the edge that has the biggest gain (difference between internal and external pull)

```
void GreedyPartitioning(GRAPH * g) {
    g->AssignRandomSide();
    while (true) {
        GRAPH_ITER git(g);
        vertex * v, bestVertex = 0;
        while(v = git.next()) {
            v->setGain(v->getExternalPull() - v->getInternalPull());
            if (gain > bestVertex->gain()) bestVertex = v;
        }
        if (bestVertex->gain() <= 0) break; // no improvements anymore
        else bestVertex->switchSide();
    }
}
```
let's try the greedy approach!

- oops!, this results in the trivial solution that one set is empty!
keeping area balance: alternating moves

- force the moves to come out of alternating sets

```c
void GreedyPartitioning(GRAPH * g) {
    g->AssignRandomSide();
    int previousSide = LEFT;
    while (true) {
        GRAPH_ITER git(g);
        vertex * v, bestVertex = 0;
        while (v = git.next()) {
            if (v->side() == previousSide) continue; // alternate!
            v->setGain(v->getExternalPull() - v->getInternalPull());
            if (gain > bestVertex->gain()) bestVertex = v;
        }
        if (bestVertex->gain() <= 0) break; // no improvement
        bestVertex->switchSide();
        previousSide = opposite(previousSide);
    }
}
```
let's try the improved greedy approach

iteration 1: vertex b has biggest gain: assign it to left

iteration 2: vertex c has biggest gain

iteration 3: none of the moves improves

we're quickly stuck in a (local?) optimum!
kernighan-lin: idea 1: swap a pair of nodes

- but if we move two at once, what is the overall gain?

\[ \text{SwapGain}(v_1, v_2) = \text{Gain}(v_1) + \text{Gain}(v_2) - 2 \times \text{cost}(e_{v_1,v_2}) \]

- \( \text{SwapGain}(b, c) = \text{Gain}(b) + \text{Gain}(c) - 2 \times \text{cost}(e_{b,c}) = 5 + 4 - 2 \times 1 = 7 \)

benefit: keeps area balance

\[ \# \text{crossing} = 4 \text{ (11 - 7)} \]
kernighan-lin: idea 1: swap gain for each pair

for each pair of vertices between the two sets, calculate the gain of swapping the pair.

so, it looks like we're still as stuck as before! or not?
void KLPartitioning(Graph * g) {

  g->AssignRandomSide();

  Initialize
  Bipartition G into V1 and V2, s.t., |V1| = |V2| ± 1
  n = |V|

  Repeat
    for i=1 to n/2
      Find a pair of unlocked vertices \( v_{ai} \in V1 \) and \( v_{bi} \in V2 \) whose
      exchange makes the largest decrease or smallest increase
      in cut-cost
      Mark \( v_{ai} \) and \( v_{bi} \) as locked
      Store the gain \( g_i \).

    Find \( k \), s.t. \( \sum_{i=1..k} g_i \) = Gain_k is maximized
    If Gain_k > 0 then
      move \( v_{a1},...,v_{ak} \) from V1 to V2 and
      \( v_{b1},...,v_{bk} \) from V2 to V1.

  Until Gain_k \leq 0
kernighan-lin: illustration (1)

Algorithm: pick the highest gain pair, and swap it tentatively

Swapping $g$ and $e$ results in a gain of $-2$
after swapping c and g, we 'lock' them, so that they will not be considered again.

swapping \(e\) and \(d\) results in an gain of 8!!

total gain of the 2 moves: -2 + 8 = 6: so, these 2 consecutive moves yield 6!
again, we perform the swap, and lock vertices d and e as well

swapping b and f results in a gain of -10

total gain of the 3 moves: -2 + 8 -10 = -4: these 3 moves give us a loss of 4, but we are not deterred, and continue!
swapping \( a \) and \( h \) results in an gain of 4

total gain of the 4 swaps:
\[
\begin{align*}
i &= 1: & -2 \\
i &= 2: & +8 \\
i &= 3: & -10 \\
i &= 4: & +4 \\
\text{sum:} & & 0
\end{align*}
\]
so, this chain of 4 moves gives us no gain
at the end of the pass, we execute just steps 1 and 2, which yields a gain of 6.

then, unlock all nodes, and start a new pass.

in the next iteration on this example will not yield an chain of swaps that improves.

... the algorithm will stop

cut amount = 2
kernighan-lin algorithm

- Kernighan-Lin bisection algorithm
  - Pair-swap neighborhood structure
  - Proceeds in "passes"
  - Start each "pass" with all modules "unlocked"
  - Select a swapping pair with highest "gain" and lock them
  - Finish the "pass" when all modules are "locked"
  - Move to the best bisection encountered during the "pass"
  - Stop when a "pass" fails to improve

- Implementation of pass
  - Straight-forward $O(n^3)$, possible: $O(n^2 \log n)$
kernighan-lin : analysis

inner (for) loop:
  • iterates n/2 times over
    • iteration 1: (n/2) x (n/2)
    • iteration i: (n/2 - i + 1)^2.
  - number of passes is independent of n
  - $O(n^3)$

• drawbacks:
  - its greedyness force it to local optimum
  - only unit vertex sizes
  - only bisections are handled
  - balanced partitions only
  - high time complexity
  - it counts edge(weight)s in the cut set, not nets:
    no multi-pin nets.
multi-pin nets in a graph

- in the graph:
  - vertices represent blocks
  - edges represent nets
- in a graph an edge always connects two vertices.
- so, how do we represent multi-pin nets???
- we could make edges between all cells of the nets
  - the net is a clique in the graph:

![Graph diagram](image-url)
algorithm set-up

clique-based, weight of an edge in a k-clique = $1/(k-1)$

<table>
<thead>
<tr>
<th>pair</th>
<th>$E_x - I_x$</th>
<th>$E_y - I_y$</th>
<th>$c(x, y)$</th>
<th>gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(a, c)$</td>
<td>0.5 – 0.5</td>
<td>2.5 – 0.5</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>$(a, f)$</td>
<td>0.5 – 0.5</td>
<td>1.5 – 1.5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$(a, g)$</td>
<td>0.5 – 0.5</td>
<td>1 – 1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$(a, h)$</td>
<td>0.5 – 0.5</td>
<td>0 – 1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>$(b, c)$</td>
<td>0.5 – 0.5</td>
<td>2.5 – 0.5</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>$(b, f)$</td>
<td>0.5 – 0.5</td>
<td>1.5 – 1.5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$(b, g)$</td>
<td>0.5 – 0.5</td>
<td>1 – 1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$(b, h)$</td>
<td>0.5 – 0.5</td>
<td>0 – 1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>$(d, c)$</td>
<td>1.5 – 0.5</td>
<td>2.5 – 0.5</td>
<td>0.5</td>
<td>2</td>
</tr>
<tr>
<td>$(d, f)$</td>
<td>1.5 – 0.5</td>
<td>1.5 – 1.5</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>$(d, g)$</td>
<td>1.5 – 0.5</td>
<td>1 – 1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$(d, h)$</td>
<td>1.5 – 0.5</td>
<td>0 – 1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$(e, c)$</td>
<td>2.5 – 0.5</td>
<td>2.5 – 0.5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$(e, f)$</td>
<td>2.5 – 0.5</td>
<td>1.5 – 1.5</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>$(e, g)$</td>
<td>2.5 – 0.5</td>
<td>1 – 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$(e, h)$</td>
<td>2.5 – 0.5</td>
<td>0 – 1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
swap (c,d)

<table>
<thead>
<tr>
<th>pair</th>
<th>$E_x - I_x$</th>
<th>$E_y - I_y$</th>
<th>$c(x, y)$</th>
<th>gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a, f)</td>
<td>0 – 1</td>
<td>1 – 2</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>(a, g)</td>
<td>0 – 1</td>
<td>1 – 1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>(a, h)</td>
<td>0 – 1</td>
<td>0 – 1</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>(b, f)</td>
<td>0.5 – 0.5</td>
<td>1 – 2</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>(b, g)</td>
<td>0.5 – 0.5</td>
<td>1 – 1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(b, h)</td>
<td>0.5 – 0.5</td>
<td>0 – 1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>(e, f)</td>
<td>1.5 – 1.5</td>
<td>1 – 2</td>
<td>0.5</td>
<td>-2</td>
</tr>
<tr>
<td>(e, g)</td>
<td>1.5 – 1.5</td>
<td>1 – 1</td>
<td>1</td>
<td>-2</td>
</tr>
<tr>
<td>(e, h)</td>
<td>1.5 – 1.5</td>
<td>0 – 1</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>
swap \((b, g)\)

<table>
<thead>
<tr>
<th>pair</th>
<th>(E_x - I_x)</th>
<th>(E_y - I_y)</th>
<th>(c(x, y))</th>
<th>gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>((a, f))</td>
<td>0 – 1</td>
<td>1.5 – 1.5</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>((a, h))</td>
<td>0 – 1</td>
<td>0.5 – 0.5</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>((e, f))</td>
<td>0.5 – 2.5</td>
<td>1.5 – 1.5</td>
<td>0.5</td>
<td>-3</td>
</tr>
<tr>
<td>((e, h))</td>
<td>0.5 – 2.5</td>
<td>0.5 – 0.5</td>
<td>0</td>
<td>-2</td>
</tr>
</tbody>
</table>
swap \((a,f)\)

\[
\begin{array}{lccr}
 i & \text{pair} & \text{gain}(i) & \sum \text{gain}(i) & \text{cutsize} \\
0 & - & - & - & 5 \\
1 & (d, e) & 2 & 2 & 3 \\
2 & (b, g) & 0 & 2 & 3 \\
3 & (a, f) & -1 & 1 & 4 \\
4 & (e, h) & -1 & 0 & 5 \\
\end{array}
\]
convert an incidence structure into a graph?

replace "nets" by cliques and divide the weight over the extra edges (overestimate of cost when \( n > 3 \))

often used: nets to cliques and weights proportional to the number of nets between the modules; then find a minimum spanning tree

there is no transformation from incidence structure to graph with equal bipartition costs!
single vertices instead of pairs are moved
  ◆ enabling other partitions than bisections
changes the gain-values to count nets
  ◆ associated with a vertex, not a pair
    in essence the cost of the nets
    that are removed from the cut
    minus the cost of the nets that enter the cut
selecting move-candidates to improve efficiency
fiduccia-mattheyses algorithm

- area constraint = [3,5]
- break ties in alphabetical order
- an initial partition

(a)

(b)
gain computation and bucket set up

\[ \text{FS}(x) = \# \text{nets having } x \text{ as its only cell in that block} \]
\[ \text{TE}(x) = \# \text{nets entirely located in that block} \]

\[ \text{gain}(x) = \text{FS}(x) - \text{TE}(x) \]

e.g. cell \( c \): contained in \( n_1, n_2, \) and \( n_3 \)
\( n_3 \) contains \( c \) as its only cell: \( \text{FS}(c) = 1 \)
\( n_1, n_2, \) and \( n_3 \) in both blocks: \( \text{TE}(c) = 0 \)
move 1: From the initial bucket we see that both cell $g$ and $e$ have the maximum gain and can be moved without violating the area constraint. We move $e$ based on alphabetical order. We update the gain of the unlocked neighbors of $e$, $N(e) = \{a, c, g, f\}$, as follows: $gain(a) = FS(a) - TE(a) = 0 - 1 = -1$, $gain(c) = 0 - 1 = -1$, $gain(g) = 1 - 1 = 0$, $gain(f) = 2 - 0 = 2$. 
move 2: \( f \) has the maximum gain, but moving \( f \) will violate the area constraint. So we move \( d \). We update the gain of the unlocked neighbors of \( d \), \( N(d) = \{b, c, f\} \), as follows: \( \text{gain}(b) = 0 - 0 = 0 \), \( \text{gain}(c) = 1 - 1 = 0 \), \( \text{gain}(f) = 1 - 1 = 0 \).
move 3: Among the maximum gain cells \( \{g, c, h, f, b\} \), we choose \( b \) based on alphabetical order. We update the gain of the unlocked neighbors of \( b \), \( N(b) = \{c\} \) as follows: \( \text{gain}(c) = 0 - 1 = -1 \).
move 4: Among the maximum gain cells \( \{g, h, f\} \), we choose \( g \) based on the area constraint. We update the gain of the unlocked neighbors of \( g \), \( N(g) = \{f, h\} \), as follows: \( gain(f) = 1 - 2 = -1 \), \( gain(h) = 0 - 1 = -1 \).
fifth move

move 5: We choose a based on alphabetical order. We update the gain of the unlocked neighbors of a, $N(a) = \{c\}$, as follows: $gain(c) = 0 - 0 = 0$. 
sixth move

move 6: We choose $f$ based on the area constraint and alphabetical order. We update the gain of the unlocked neighbors of $f$, $N(f) = \{h, c\}$, as follows: $gain(h) = 0 - 0 = 0$, $gain(c) = 0 - 1 = -1$. 
move 7: We move $h$. $h$ has no unlocked neighbor.
move 8: We move c.

<table>
<thead>
<tr>
<th>i</th>
<th>cell</th>
<th>$g(i)$</th>
<th>$\sum g(i)$</th>
<th>cutsize</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>e</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>d</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>g</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>-1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td>-1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>h</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>c</td>
<td>-1</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>
compute all gains
find the "base module" 
  (maximum gain, no balance violation)
move it, lock it and update all gains
repeat the previous two steps 
  if there are still unlocked modules
select the best subsequence of moves 
  if there is a profitable subsequence (positive gain) 
  then perform the moves in the subsequence and unlock

else STOP
dependency on initial solution

21K random starts, 3K network -- Alpert/Kahng