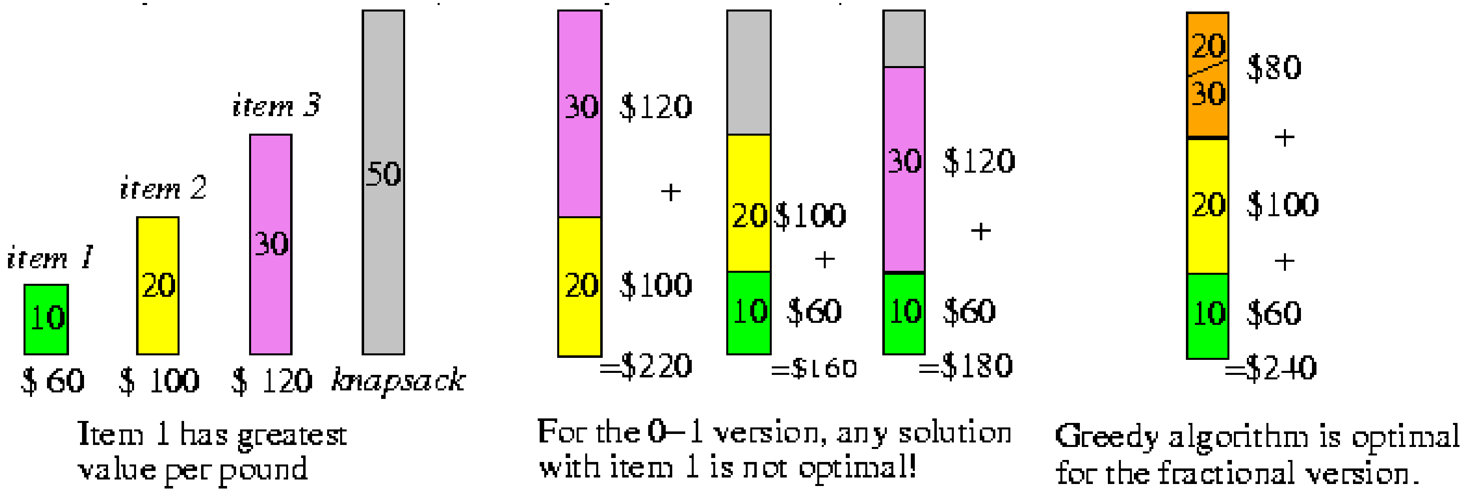
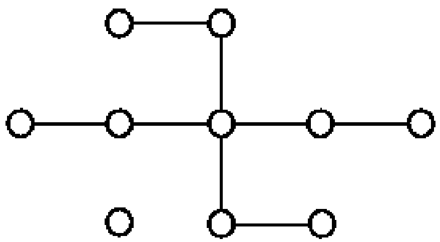
Unit 5: Greedy Algorithms

- **Course contents:**
  - Elements of the greedy strategy
  - Activity selection
  - Knapsack problem
  - Huffman codes
  - Task scheduling
- **Appendix: Process antenna effect fixing**
- **Reading:**

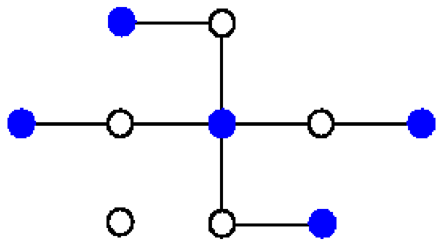


Greedy Heuristic: Vertex Cover

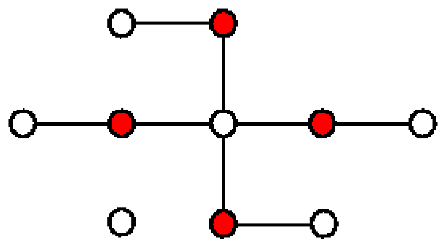
- A **vertex cover** of an undirected graph $G=(V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$, then $u \in V'$ or $v \in V'$, or both.
  - The set of vertices covers all the edges.
- The **size** of a vertex cover is the number of vertices in the cover.
- The **vertex-cover problem** is to find a vertex cover of minimum size in a graph.
- **Greedy heuristic:** cover as many edges as possible (vertex with the maximum degree) at each stage and then delete the covered edges.
- **The greedy heuristic cannot always find an optimal solution!**
  - The vertex-cover problem is NP-complete.



A graph instance



A vertex cover of size 5 obtained by the greedy algorithm.

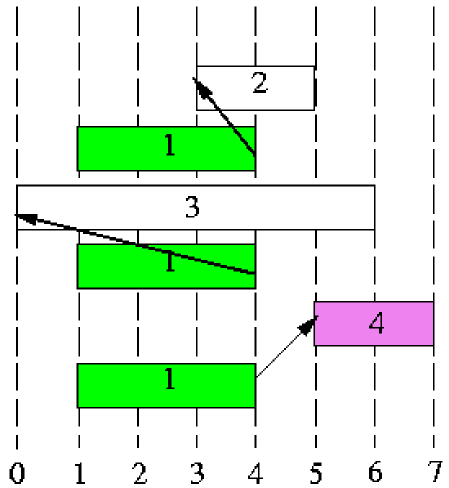


A vertex cover of size 4 optimal solution!!

A Greedy Algorithm

- A **greedy algorithm** always makes the choice that looks best at the moment.
- **An Activity-Selection Problem:** Given a set $S = \{1, 2, \ldots, n\}$ of n proposed activities, with a start time s_i and a finish time f_i for each activity i, select a maximum-size set of mutually compatible activities.
  - If selected, activity i takes place during the half-open time interval $[s_i, f_i)$.
  - Activities i and j are **compatible** if $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap (i.e., $s_i \geq f_j$ or $s_j \geq f_i$).

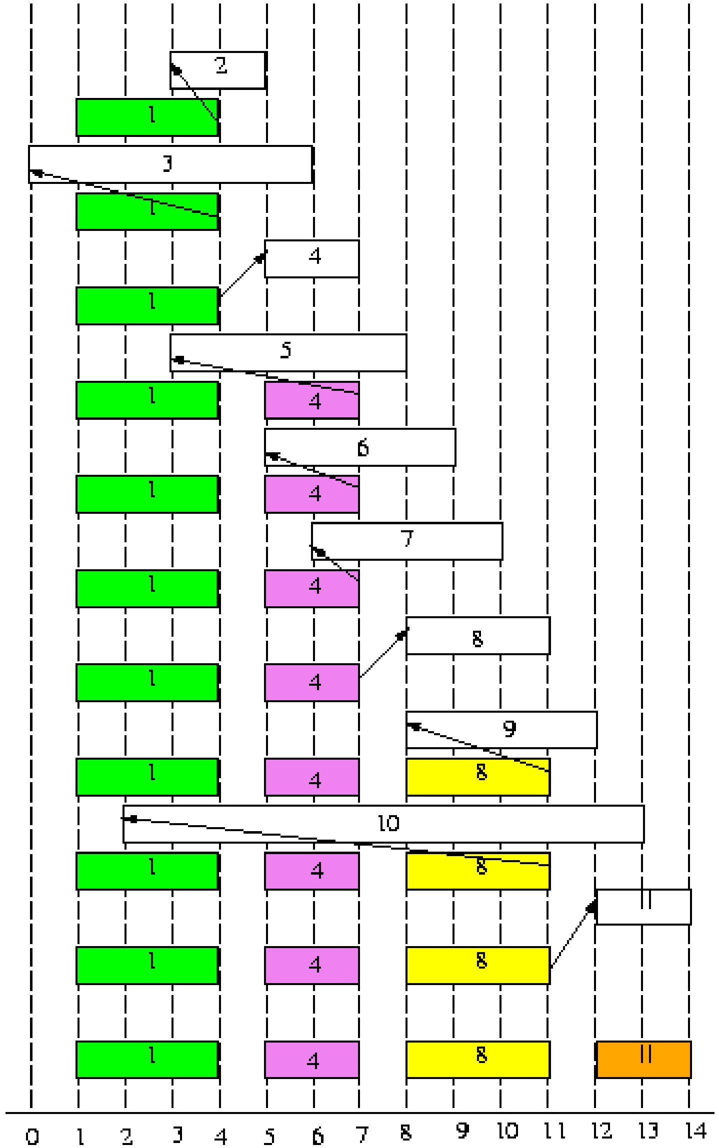
i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7



Compatible activities:
(1, 4), (2, 4)

Activity Selection

i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



1. Sort f_i
2. Select the first activity.
3. Pick the first activity i such that $s_i >= f_j$ where activity j is the most recently selected activity.

The Activity-Selection Algorithm

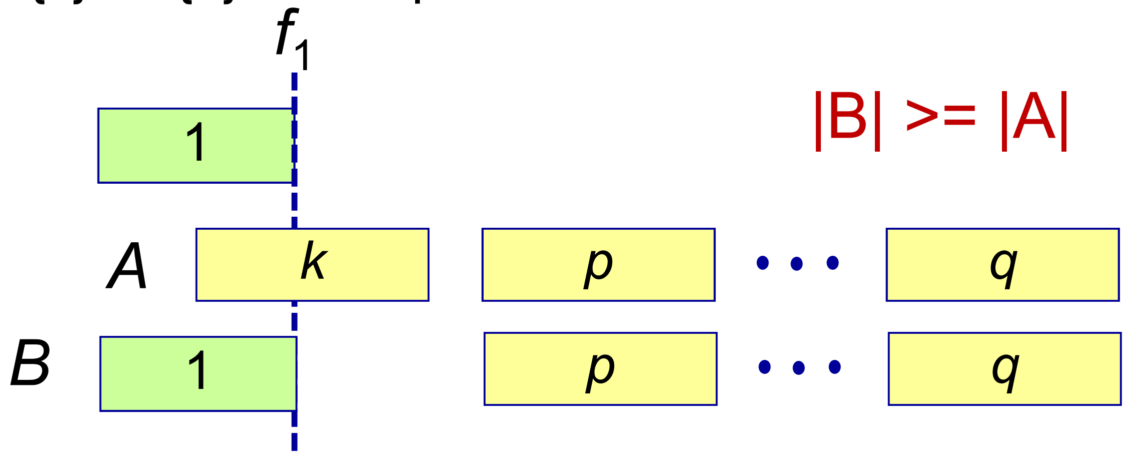
```
Greedy-Activity-Selector(s,f)
// Assume f_1 ≤ f_2 ≤ … ≤ f_n.
1. n = s.length
2. A = {1}   // a_1 in 3rd Ed.
3. j = 1
4. for  i = 2 to n
5.     if s_i ≥ f_j
6.         A = A ∪ {i}
7.         j = i
8. return  A
```

- Time complexity excluding sorting: $O(n)$

- **Theorem:** Algorithm Greedy-Activity-Selector produces solutions of maximum size for the activity-selection problem.
  - (**Greedy-choice property**) Suppose $A \subseteq S$ is an optimal solution. Show that if the first activity in A activity $k \neq 1$, then $B = A - \{k\} \cup \{1\}$ is an optimal solution.
  - (**Optimal substructure**) Show that if A is an optimal solution to S, then $A' = A - \{1\}$ is an optimal solution to $S' = \{i \in S: s_i \geq f_1\}$.
  - Prove by induction on the number of choices made.

Optimality Proofs

- (**Greedy-choice property**) Suppose $A \subseteq S$ is an optimal solution. Show that if the first activity in A activity $k \neq 1$, then $B = A - \{k\} \cup \{1\}$ is an optimal solution.



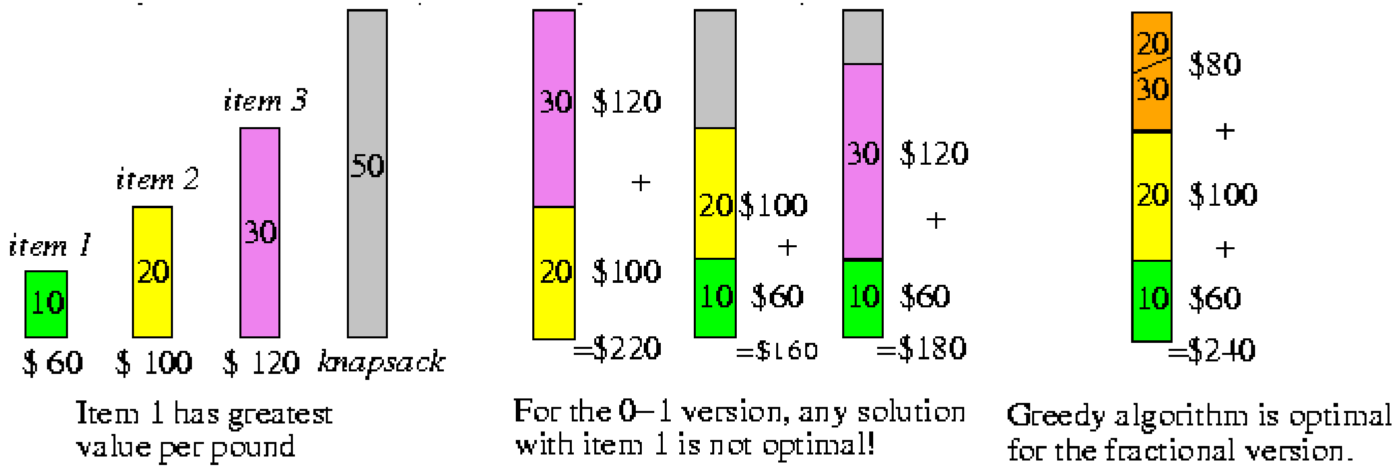
- (**Optimal substructure**) Show that if A is an optimal solution to S, then $A' = A - \{1\}$ is an optimal solution to $S' = \{i \in S: s_i \geq f_1\}$.
  - Exp: A'= {4, 8, 11}, S'= {4, 6, 7, 8, 9, 11} in the Activity Selection example
  - Proof by contradiction: If A' is not an optimal solution to S', we can find a "better" solution A'' (than A'). Then, A''U{1} would be a better solution than A'U{1} = A to S, contradicting to the original claim that A is an optimal solution to S. (Activity 1 is compatible with all the tasks in A''.)

Elements of the Greedy Strategy

- When to apply greedy algorithms?
  - **Greedy-choice property:** A global optimal solution can be arrived at by making a locally optimal (greedy) choice.
    - Dynamic programming needs to check the solutions to subproblems.
  - **Optimal substructure:** An optimal solution to the problem contains within its optimal solutions to subproblems.
    - E.g., if A is an optimal solution to S, then $A' = A - \{1\}$ is an optimal solution to $S' = \{i \in S: s_i \geq f_1\}$.
- Greedy ***heuristics*** do not always produce optimal solutions.
- Greedy algorithms vs. dynamic programming (DP)
  - Common: optimal substructure
  - Difference: greedy-choice property
  - DP can be used if greedy solutions are not optimal.

Knapsack Problem

- **Knapsack Problem:** Given n items, with ith item worth v_i dollars and weighing w_i pounds, a thief wants to take as valuable a load as possible, but can carry at most W pounds in his knapsack.
- **The 0-1 knapsack problem:** Each item is either taken or not taken (0-1 decision).
- **The fractional knapsack problem:** Allow to take fraction of items.
- **Exp:** $\vec{v}$ = (60, 100, 120), $\vec{w}$ = (10, 20, 30), W = 50



- Greedy solution by taking items in order of greatest value per pound is optimal for the fractional version, but not for the 0-1 version.
- The 0-1 knapsack problem is NP-complete, but can be solved in $O(nW)$ time by DP. (**A polynomial-time DP??**)

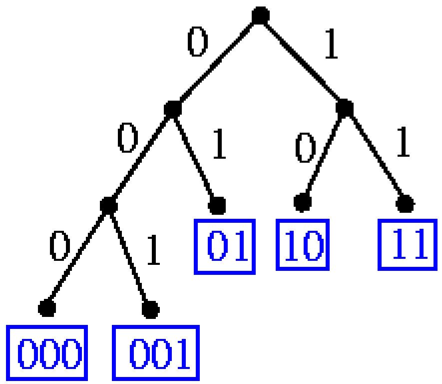
Coding

- Is used for data compression, instruction-set encoding, etc.
- **Binary character code:** character is represented by a unique binary string
  - **Fixed-length code (block code):** *a*: 000, *b*: 001, …, *f*: 101 ⇒ ace ↔ 000 010 100.
  - **Variable-length code:** frequent characters ⇒ short codeword; infrequent characters ⇒ long codeword

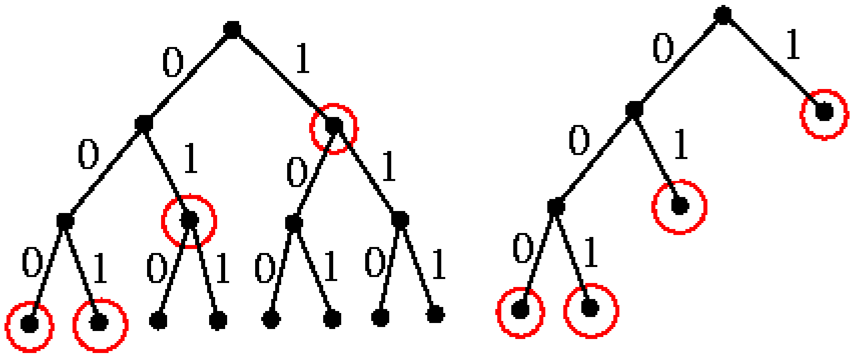
	a	b	c	d	e	f	cost / 100 characters
Frequency	45	13	12	16	9	5	
Fixed–length codeword	000	001	010	011	100	101	300
Variable–length codeword	0	101	100	111	1101	1100	224

Binary Tree vs. Prefix Code

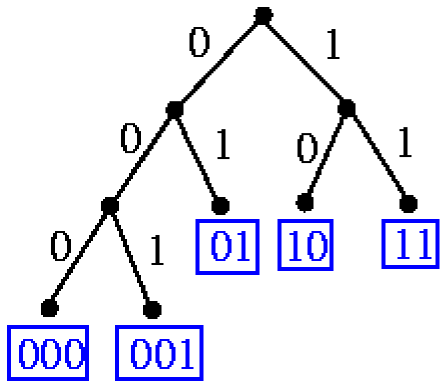
- **Prefix code:** No code is a prefix of some other code.



binary tree –> prefix code



prefix code {1, 01, 000, 001}–> binary tree

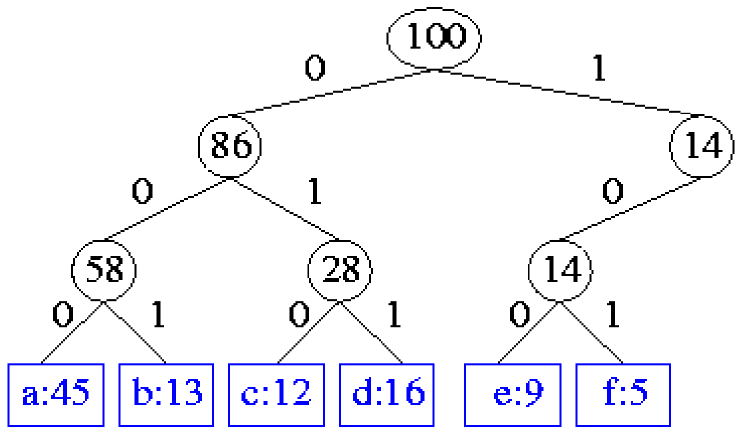


decoding: 01 10 000 000 001 11 11

arrive at a leaf, start at root

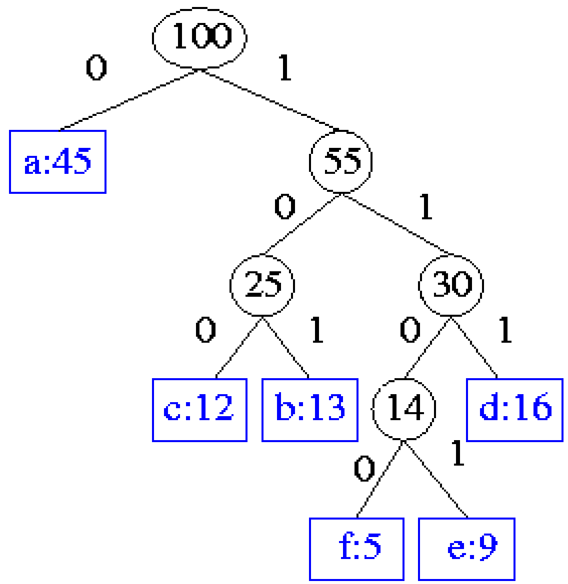
Optimal Prefix Code Design

- **Coding Cost** of T: $B(T)=\sum_{c \in C} c.freq \cdot d_T(c)$
  - c: character in the alphabet C
  - $c.freq$: frequency of c
  - $d_T(c)$: depth of c's leaf (length of the codeword of c)
- **Code design:** Given $c_1.freq$, $c_2.freq$, …, $c_n.freq$, construct a binary tree with n leaves such that $B(T)$ is minimized.
  - Idea: more frequently used characters use shorter depth.



Fixed–length cost: 3 * 100 = 300

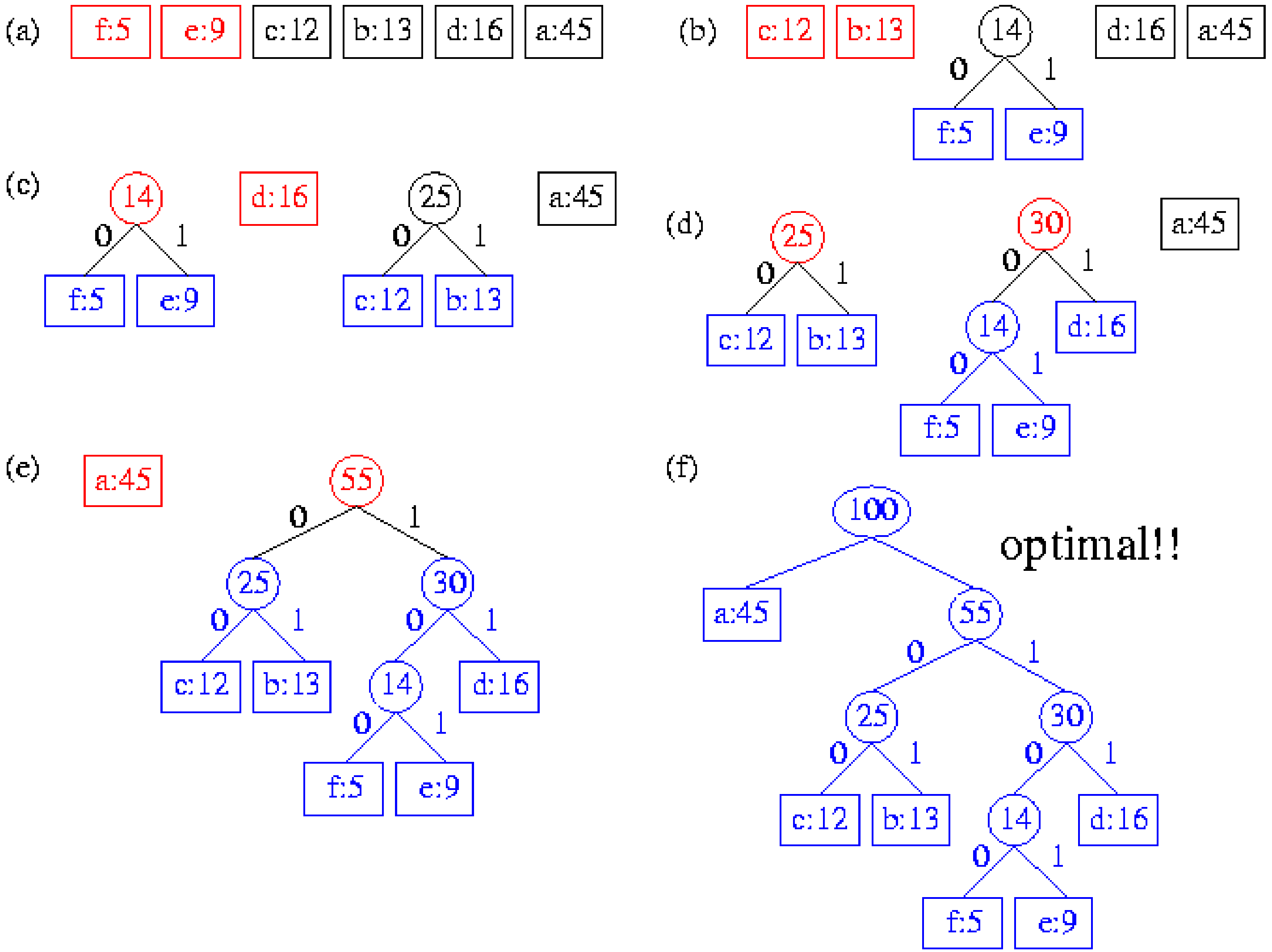
optimal code ––> full binary tree!!



Variable–length cost = 224

Huffman's Procedure

- **Pair** two nodes with the least costs at each step.



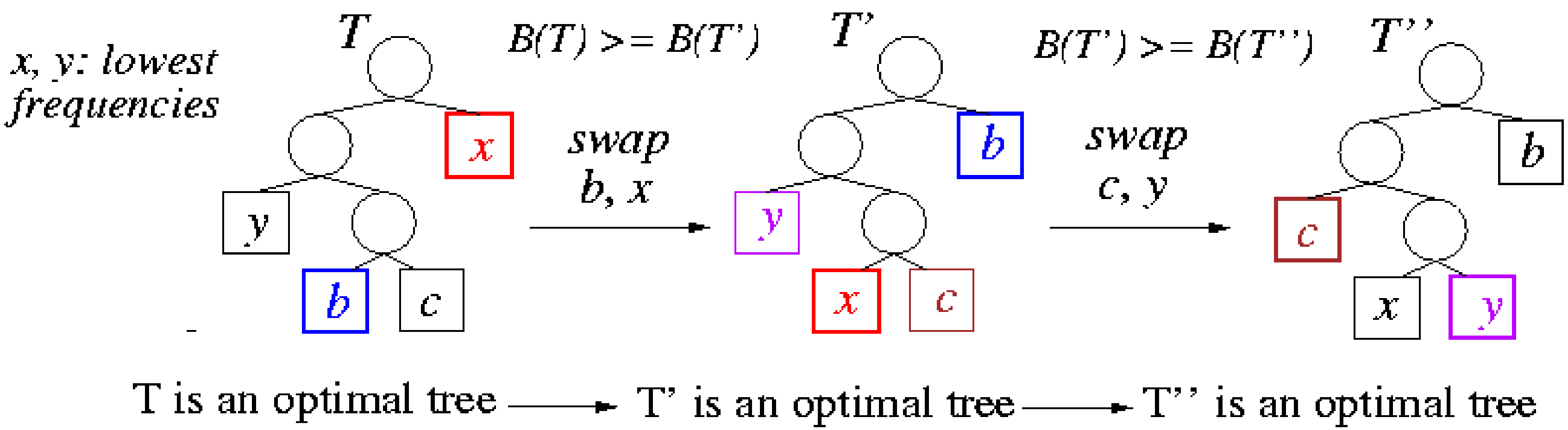
Huffman's Algorithm

```
Huffman(C)
1. n = |C|
2. Q = C
3. for i = 1 to n -1
4.       Allocate a new node z
5.       z.left = x = Extract-Min(Q)
6.       z.right = y = Extract-Min(Q)
7.       z.freq = x.freq + y.freq
8.       Insert(Q, z)
9. return Extract-Min(Q)  //return the root of the tree
```

- Time complexity: $O(n \lg n)$.
  - Extract-Min(Q) needs $O(\lg n)$ by a **heap** operation.
  - Requires initially $O(n \lg n)$ time to build a binary heap.

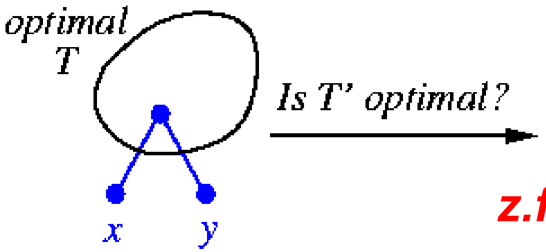
Huffman's Algorithm: Greedy Choice

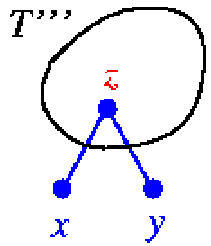
- **Greedy choice:** Two characters *x* and *y* with the lowest frequencies must have *the same length* and differ only in the last bit.



Huffman's Algorithm: Optimal Substructure

- **Optimal substructure:** Let T be a full binary tree for an optimal prefix code over C. Let z be the parent of two leaf characters x and y. If $z.freq = x.freq + y.freq$, tree $T' = T - \{x, y\}$ represents an optimal prefix code for $C' = C - \{x, y\} \cup \{z\}$.





$z.freq = x.freq + y.freq$

$B(T) = B(T') + x.freq + y.freq$
$(d_T(x) = d_T(y) = d_{T'}(z) + 1)$

If T' is not optimal, find T'' s.t. $B(T'') < B(T')$.
z in C' => z is a leaf of T''.
Add x, y as z's children (T''')

$$\Rightarrow B(T''') = B(T'') + x.freq + y.freq$$
$$< B(T') + x.freq + y.freq$$
$$= B(T)$$

contradiction!!

Task Scheduling

- **The task scheduling problem:** Schedule unit-time tasks with deadlines and penalties s.t. the total penalty for missed deadlines is minimized.
  - $S = \{1, 2, \ldots, n\}$ of n unit-time tasks.
  - **Deadlines** $d_1, d_2, \ldots, d_n$ for tasks, $1 \le d_i \le$ n.
  - **Penalties** $w_1, w_2, \ldots, w_n$: w_i is incurred if task i misses deadline.
- Set A of tasks is **independent** if $\exists$ a schedule with no late tasks.
- $N_t(A)$: number of tasks in A with deadlines t or earlier, $t = 1, 2, \ldots, n$.
- Three equivalent statements for any set of tasks A
  1. A is independent.
  2. $N_t(A) \le t$, $t = 1, 2, \ldots, n$.
  3. If the tasks in A are scheduled in order of nondecreasing deadlines, then no task is late.

Greedy Algorithm: Task Scheduling

- **The optimal greedy scheduling algorithm:**
  1. Sort penalties in non-increasing order.
  2. Find tasks of independent sets: no late task in the sets.
  3. Schedule tasks in a maximum independent set in order of nondecreasing deadlines.
  4. Schedule other tasks (missing deadlines) at the end arbitrarily.

Task	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

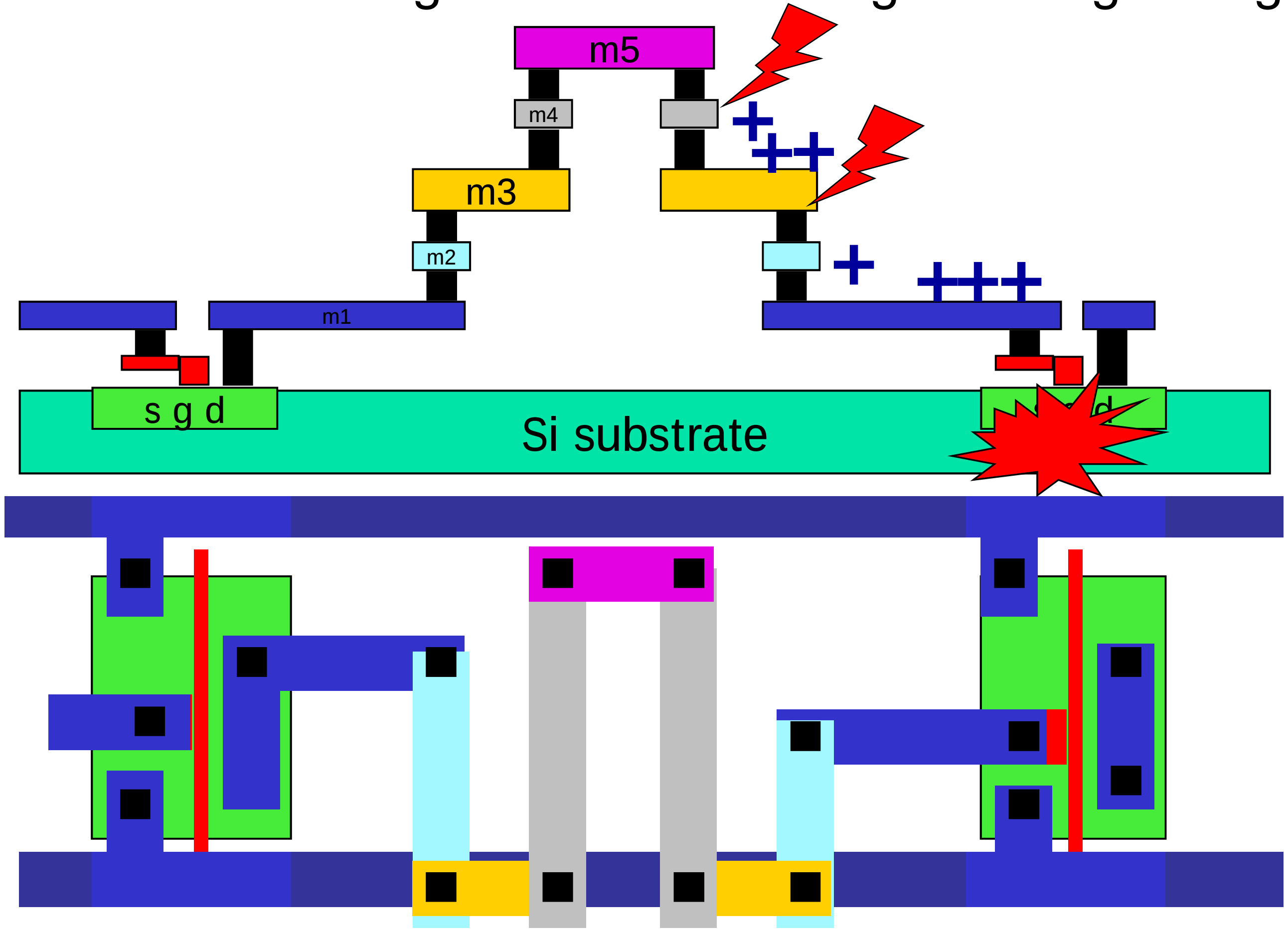
optimal scheduling: (2, 4, 1, 3, 7, 5, 6)
penalty: 30+20 = 50

$N_1(A) = 0 <= 1$
$N_2(A) = 1 <= 2$
$N_3(A) = 2 <= 3$
$N_4(A) = 4 <= 4$
$N_5(A) = 4 <= 5$
$N_6(A) = 5 <= 6$

$N_t(A) <= t$

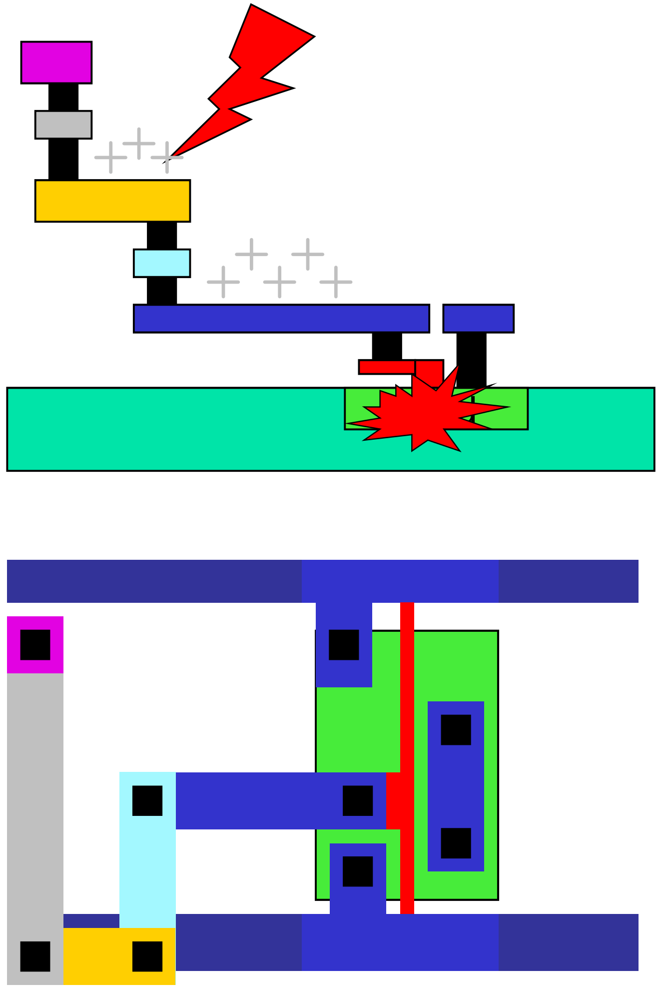
Appendix: Process Antenna Effect

- While the metal line is being manufactured, a long floating interconnect acts as a temporary capacitor to store charges induced from plasma etching.
- The accumulated charges on the wires might damage the gate.



Antenna Effect

- Depends on length of the wire that is "unshorted" (that is, not connected to a diffusion drain area)
  - The longer the wires, the more the charge.
  - Wires are always shorted in the highest metal layer.
- Depends on the gate size
  - Aggressive down sizing makes the problem worse!
- The calculation of this design rule is different per fab.

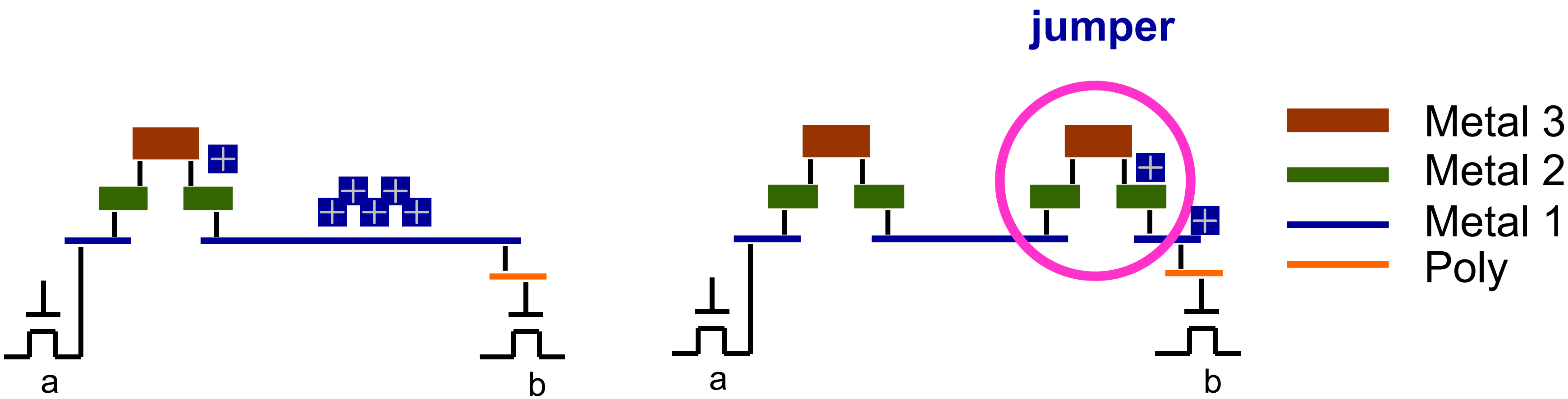


courtesy Prof. P. Groeneveld

Jumper Insertion

- Forces a routing pattern that "shoots up" to the highest layer as soon as possible.
- Reduces the charge amount for violated nets during manufacturing.

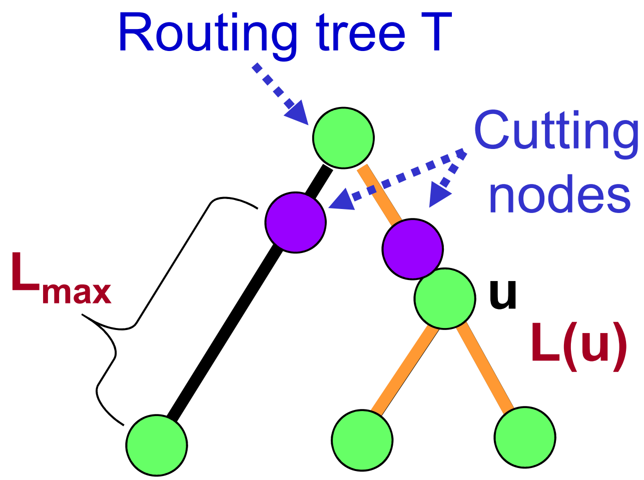
Side effects: Delay and congestion



A two-pin net **A two-pin net with jumper insertion**

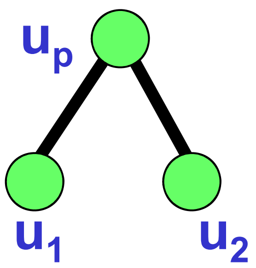
Jumper Insertion for Antenna Fixing/Avoidance

- Su and Chang, DAC-05 (& ISPD-06, IEEE TCAD-07).
- Formulate the problem of jumper insertion on a routing tree for antenna avoidance as a tree cutting problem.
- Problem ***JITA*** (*Jumper Insertion on a Routing Tree for Antenna Avoidance*): Given a routing tree $T = (V,E)$ and an upper bound *Lmax*, find the minimum set C of cutting nodes, $c \neq u$ for any $c \in C$ and $u \in V$, so that $L(u) \leq Lmax, \forall u \in V$.
  - $T = (V,E)$: a routing tree.
  - *Lmax*: antenna upper bound.
  - $L(u)$: sum of edge weights (antenna strengths) connected to node u



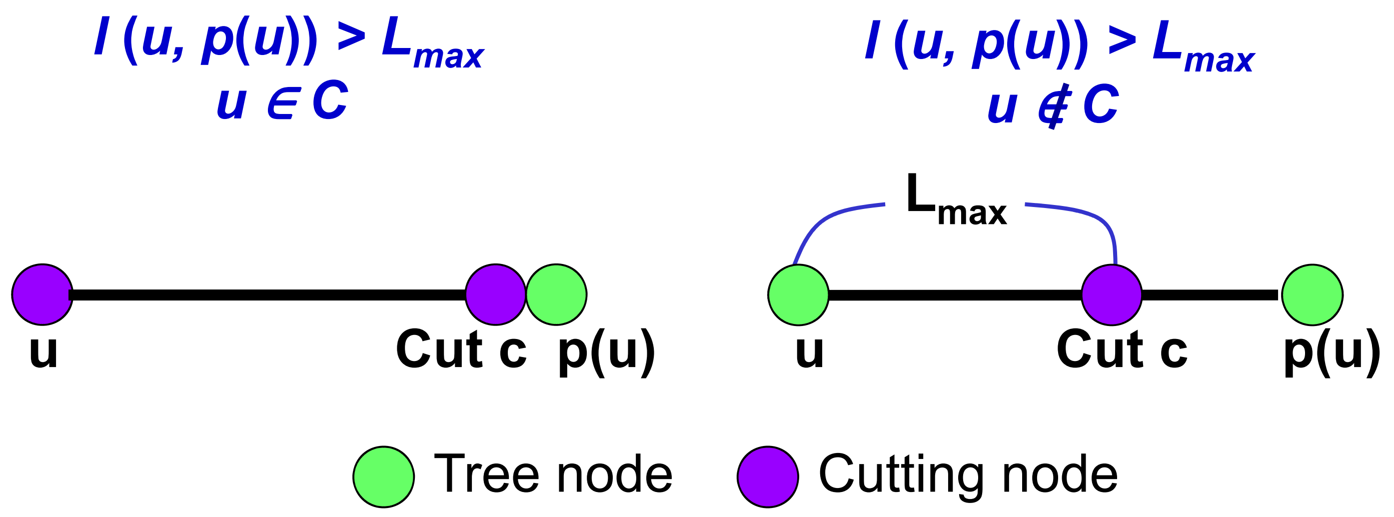
Algorithm BUJI

- The exact BUJI (Bottom-Up Jumper Insertion) algorithm for jumper insertion uses a bottom-up approach to insert cutting nodes on the routing tree.
  - Step 1: Make every leaf node satisfy the antenna rule.
  - Step 2: Make every subleaf node satisfy the antenna rule, then cut the subleaf node into a new leaf node.
- Definition: A subleaf is a node for which all its children are leaf nodes, and all the edges between it and its children have antenna weights ≤ Lmax.



Step 1: Leaf Node Processing

- Step 1: Prevent every leaf node from antenna violation.



L_{max}: upper bound on antenna
C : cutting set

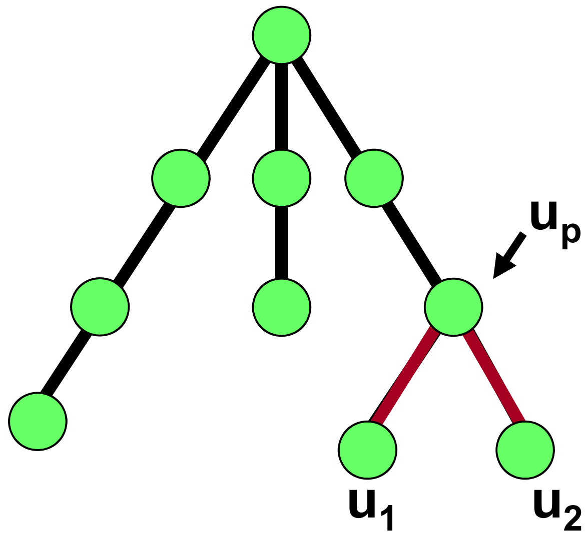
p(u): parent of u
l(e), l(u,v): weight of the edge e = (u,v)

Step 2: Subleaf Node Processing

- Step 2: Prevent every subleaf node from antenna violation
- *totallen*: sum of weights of the edges between the node and its children.

$$\boldsymbol{totallen = \sum_{i=1}^{k} l(u_i , u_p)}$$

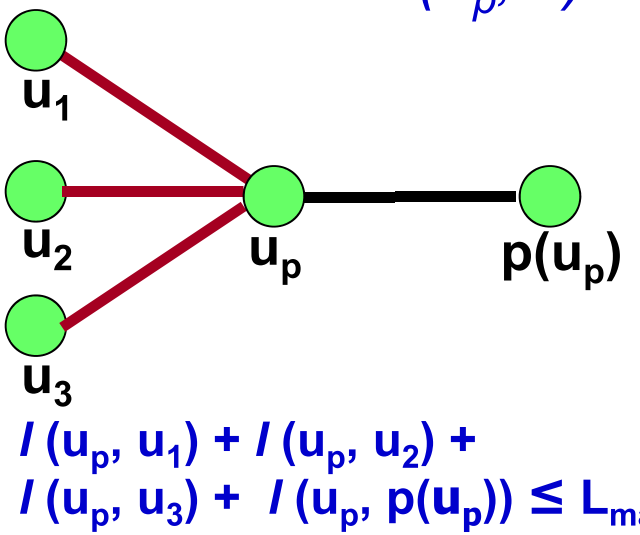
u_p : a subleaf node
u_i : subleaf's children, $\forall 1 \leq i \leq k$

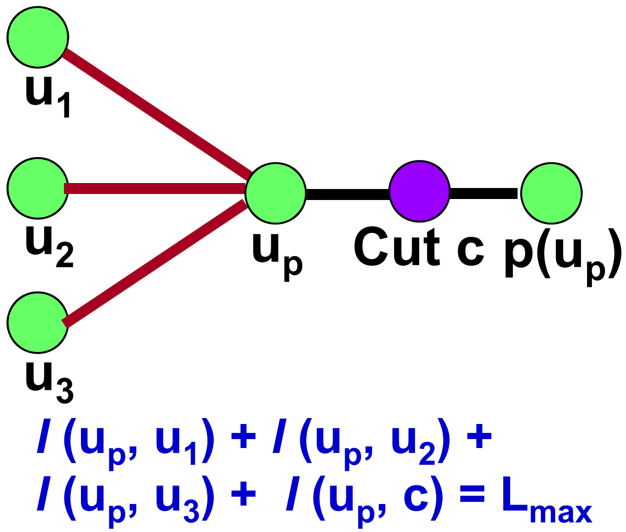


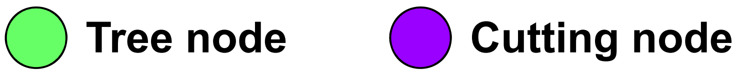
- Classify the subleaf nodes according to *totallen*.
  - Case 1: *totallen* $\leq L_{max}$
  - Case 2: *totallen* $> L_{max}$

Case 1: totallen ≤ L_{max}

- Case 1: *totallen* ≤ L_{max}
  - If u_p's parent exists
    - If *totallen* + $l(u_p, p(u_p)) \le L_{max}$, cut u_p's children from the tree
    - Else insert the cutting node that makes *totallen* + $l(u_p, c) = L_{max}$

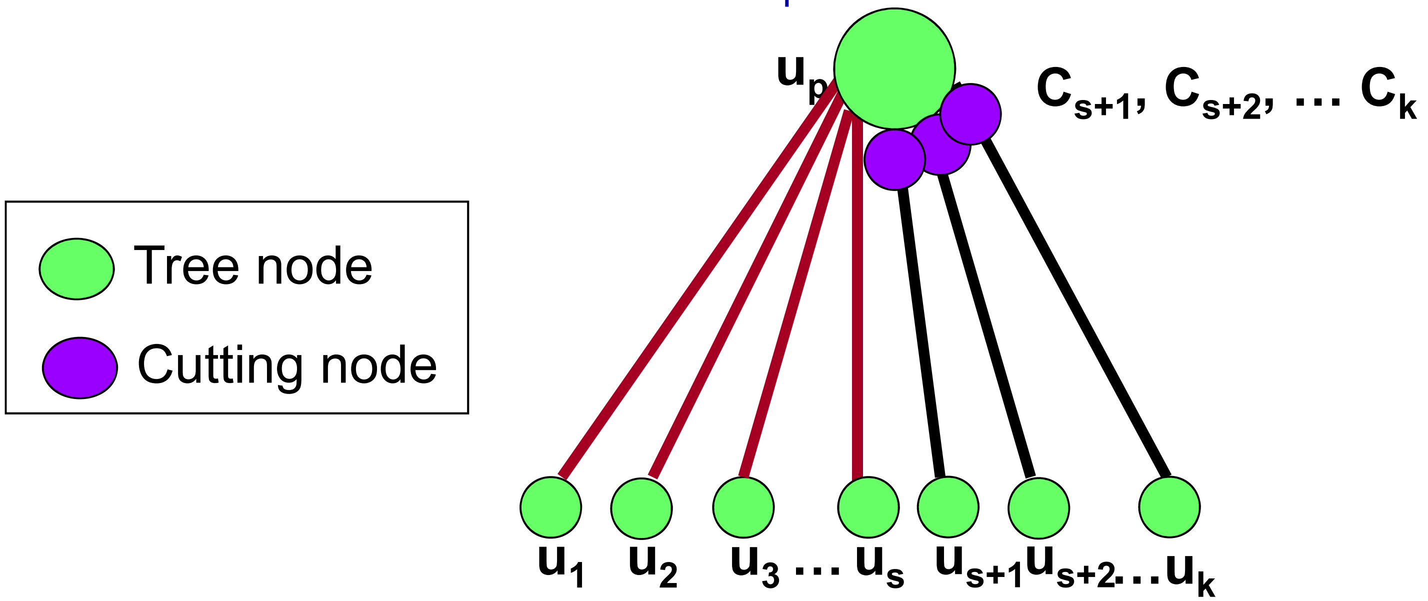






Case 2: totallen > L_{max}

- Case 2: *totallen* > L_{max}
  - Step 1: Let $A[i] \leftarrow l(u_i, u_p), \forall 1 \leq i \leq k$.
    Sort *A* in non-decreasing order.
  - Step 2: Find the maximum *s* such that $\sum_{j=1}^{s} A[j] \leq L_{max}$
  - Step 3: Add cutting nodes $c_{s+1}, \ldots, c_k$.
  - Step 4: Use Case 1 to cut u_p into a leaf node.



Complexity

- Algorithm BUJI optimally solves the JITA problem in $O(V \lg V)$ time using $O(V)$ space, where V is the number of vertices.
- With the SPLIT data structure proposed by Kundu and Misra, JITA can be done in $O(V)$ time and space.
  - Optimal algorithm in the theoretical sense.

Resulting Layout with Obstacles

- Lmax = 500 um, 1000 tree nodes (circles), 500 obstacles (rectangles), 426 jumpers (x)