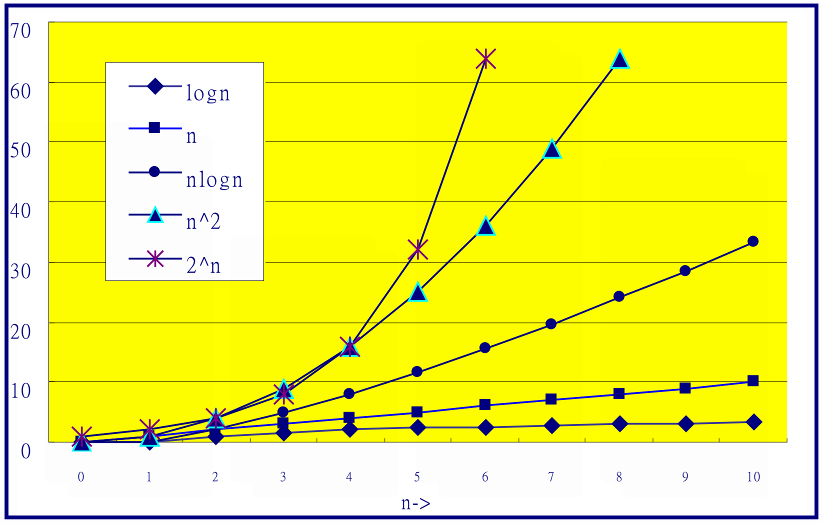
Basic Physical Design Algorithm Reviews and Layout System Fundamentals

- Course contents:
  - Computational complexity reviews
  - Basic algorithms
  - Layout system fundamentals



Introduction to This Unit

- VLSI design process: transformation of data from HDL code (logic), to schematics (circuit), to layout (physical)
  - Database management problem
- Layout therefore is represented as a collection of several layers of planar geometric elements or polygons
  - Symbolic database captures net and transistor attributes
  - Symbolic database is converted into a polygon database prior to tapeout
  - Symbolic database is used for technology independence, but never reached

Introduction (cont)

- CAD tools are needed due to sophisticated layout database manipulation
  - CAD tools require highly specialized algorithms and data structures
  - Three categories:
    - Helps human designers to manipulate layouts
      - Layout editor (e.g. Springsoft 'Laker')
    - Performs some task on the layout automatically
      - Channel router and placement tools (e.g. Cadence 'QPlace')
    - Checks and verifies layout
      - DRC and LVS (e.g. Mentor 'Calibre')
  - Research in physical design focused on the last two types
  - Algorithms for partitioning, placement and routing are based on graph theory (and computational geometry)

Asymptotic Notation -Big "oh"

- f(n)=O(g(n)) iff
  - ∃ positive const. c and $n_{0,}$ ∍ $f(n) \le cg(n)\ \forall\ n, n \ge n_0$
  - e.g.
    - **3n+2 =O(n)**

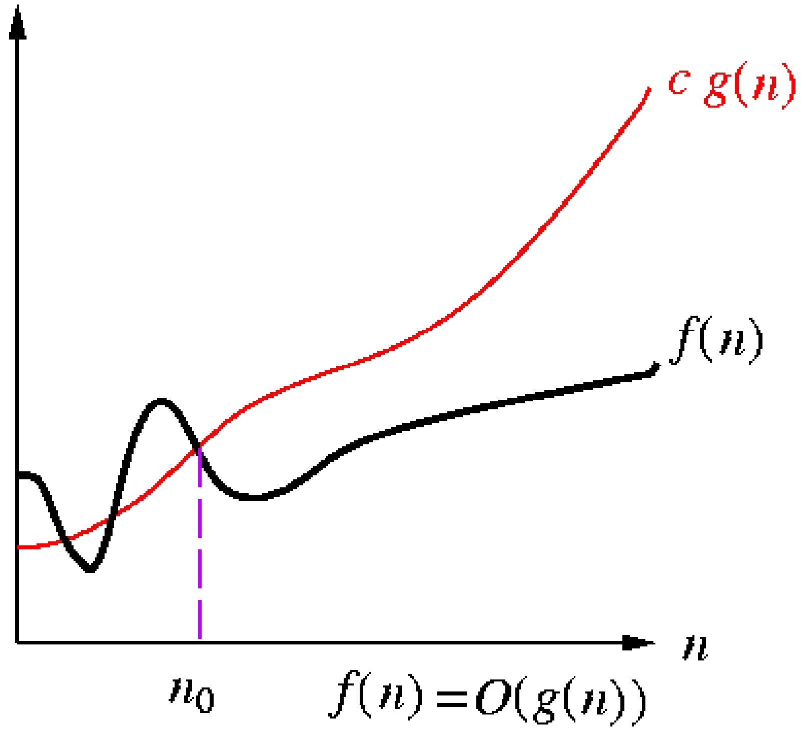
      $3n+2 \le 4n$ for all $n \ge 2$
    - **$10n^2+4n+2=O(n^2)$**

      $10n^2 +4n+2 \le 11n^2$

      for all $n \ge 10$
    - **$3n+2 = O(n^2)$**

      $3n+2 \le n^2$ for all $n \ge 4$

* g(n) should be a ***least upper bound***



Asymptotic Notation -Omega

- $f(n)=\Omega(g(n))$ iff
  - $\exists$ positive const. c and $n_{0,}$ $\ni$ $f(n) \geq cg(n)$ $\forall$ n, $n \geq n_0$
  - e.g.
    - **$3n+3 = \Omega(n)$** **$3n+3 \geq 3n$ for all $n \geq 1$**
    - **$6*2^n + n^2 = \Omega(2^n)$** **$6*2^n + n^2 \geq 2^n$ for all $n \geq 1$**
    - **$3n+3 = \Omega(1)$** **$3n+3 \geq 3$ for all $n \geq 1$**

* g(n) should be a ***most lower bound***

Asymptotic Notation -Theta

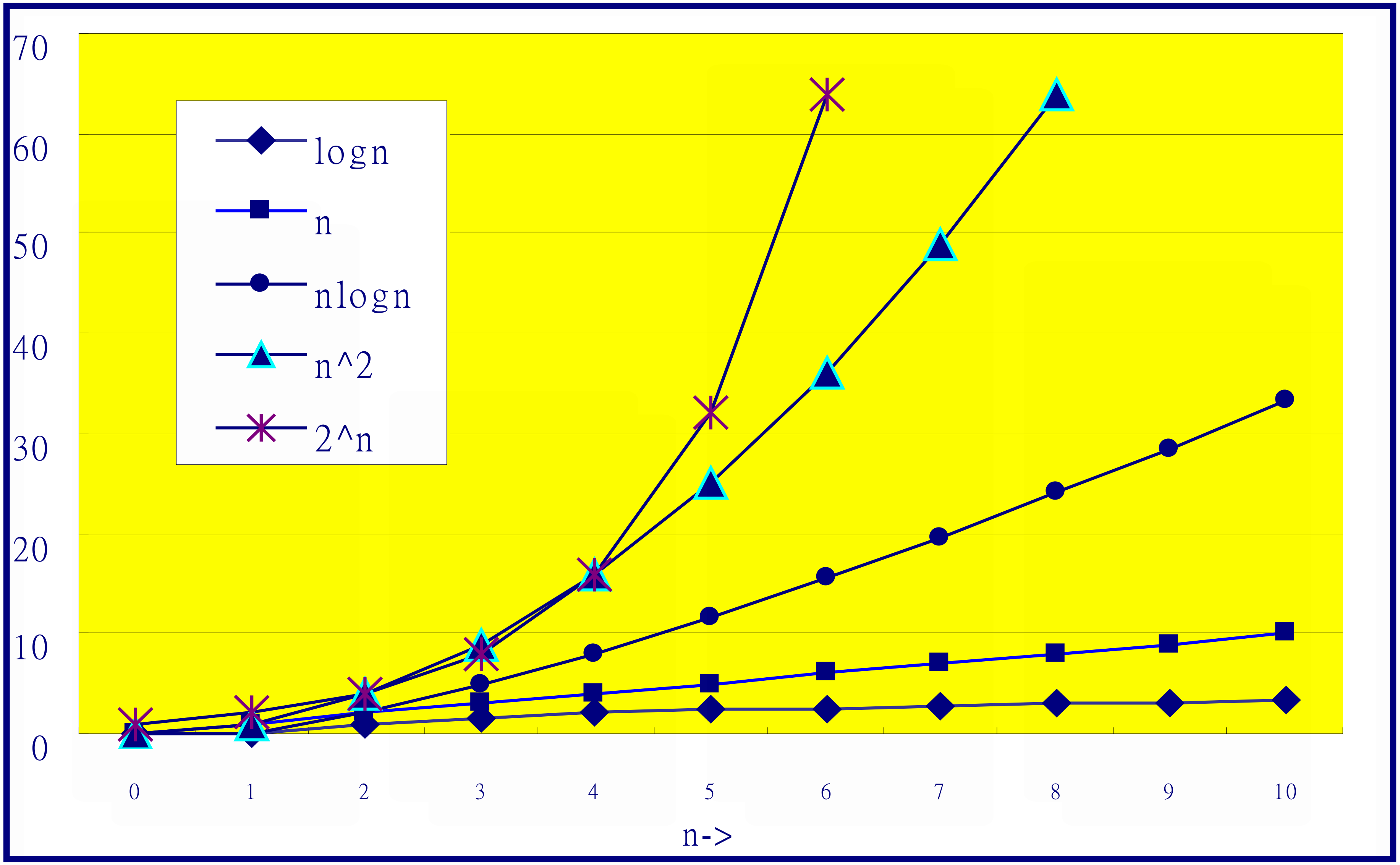
- $f(n)=\Theta(g(n))$ iff
  - $\exists$ positive constants c_1, c_2, and $n_0 \ni c_1 g(n) \le f(n) \le c_2 g(n) \quad \forall\ n,\ n \ge n_0$
  - e.g.
    - **$3n+2 = \Theta(n)$ $\quad$ $3n \le 3n+2 \le 4n$, for all $n \ge 2$**
    - **$10n^2+4n+2= \Theta(n^2)$ $\quad$ $10n^2 \le 10n^2+4n+2 \le 11n^2$,**
      for all $n \ge 5$

* g(n) should be both ***lower bound & upper bound***

Computational Complexity

- Computational complexity: an abstract measure of the **time** and **space** necessary to execute an algorithm as function of its "input size".
- Input size examples:
  - sort n words of bounded length $\Rightarrow n$
  - **the input is the integer $n \Rightarrow \lg n$**
  - the input is the graph $G(V, E) \Rightarrow |V|$ and $|E|$

Output Growing Curves



Amortized Analysis

- **Why Amortized Analysis?**
  - Find a tight bound of a sequence of data structure operations.
- No probability involved, guarantees the **average performance of each operation in the worst case**
- Three popular methods
  - Aggregate method
  - Accounting method
  - Potential method

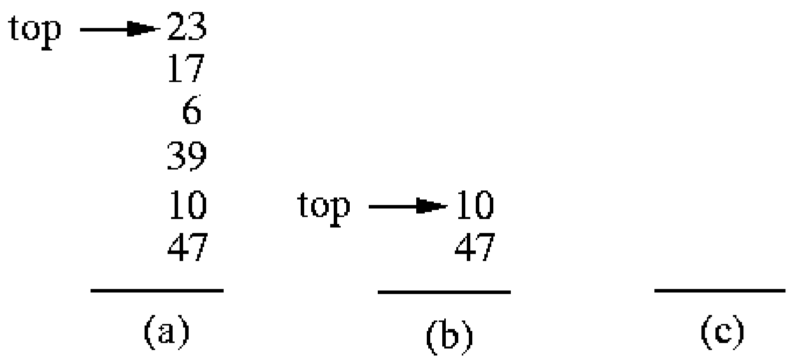
Methods for Amortized Analysis

- **Aggregate** method
  - n operations take $T(n)$ time.
  - Average cost of an operation is $T(n)/n$ time.
- **Accounting** method
  - Charge each type of operation an amortized cost.
  - Store the overcharge of early operations as "prepaid credit" in "bank."
  - Use the credit for later operations.
  - **Must guarantee nonnegative balance at all time**
- **Potential** method
  - View "prepaid credit" as "potential energy."

Aggregate Method: Stack and MULTIPOP

- *n* operations take *T*(*n*) time □ average cost of an operation is *T*(*n*)/*n* time.
- Consider a sequence of *n* PUSH, POP, and MULTIPOP operations on an initially empty stack.
  - Worst-case analysis: a MULTIPOP operation takes *O*(*n*).
  - Aggregate method: Any sequence of *n* PUSH, POP, MULTIPOP costs at most *O*(*n*) time **(why?)** $\Rightarrow$ amortized cost of an operation: *O*(*n*)/*n*=*O*(1).

```
Multipop(S, k)
1. while not Stack-Empty(S) and k > 0 do
2.     Pop(S)
3.     k ← k-1
```



Multipop(*S*, 4)   Multipop(*S*, 7)

Accounting Method

- Assign differing charges to different operations.
  - Amortized cost = actual cost + credit.
  - Credit is assigned to specific objects and must be nonnegative all the times.
- Stack operations (*s*: stack size):

	Actual cost	Amortized cost
PUSH	1	2
POP	1	0
MULTIPOP	Min(k, s)	0

  - For any sequence of *n* operations, total actual cost ≤ total amortized cost = $O(n)$.

Potential Method

- Represent the prepaid work as "potential" that can be released to pay for future operations.
  - Potential is associated with the whole data structure, not with specific items in the data structure. (cf. accounting method)
- The potential method:
  - D_0: initial data structure
    D_i: data structure after applying the *i*th operation to D_{i-1}
    c_i: actual cost of the *i*th operation
  - Define the potential function $\Phi : D_i \rightarrow \Re$.
  - Amortized cost $\hat{c}_i$, $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$.

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

  - Pick $\Phi(D_n) \geq \Phi(D_0)$ to make $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$
  - Often define $\Phi(D_0) = 0$ and then show that $\Phi(D_i) \geq 0$, $\forall\, i$.

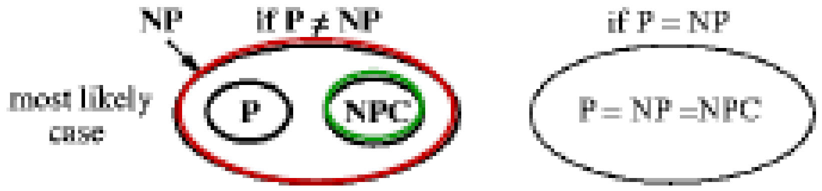
Potential Method: Stack Operations

- Amortized cost of each operation = $O(1)$.
- $\Phi(D)$ = # of objects in the stack D; $\Phi(D_0)=0$, $\Phi(D_i) \geq 0$.
- PUSH: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (s+1) - s = 2$.
- POP: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (-1) = \mathbf{0}$.
- MULTIPOP(S, k): $k' = \min(k, s)$ objects are popped off.

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1})\\ &= k' - k'\\ &= 0.\end{aligned}$$

Complexity Classes

- **The class P:** class of problems that can be **solved** in polynomial time in the size of input.
  - **Size of input:** size of encoded "binary" strings.
  - Edmonds: Problems in P are considered **tractable**.
- **The class NP** (**N**ondeterministic **P**olynomial): class of problems that can be **verified** in polynomial time in the size of input.
  - P = NP?
- **The class NP-complete (NPC): Any** NPC problem can be solved in polynomial time ⇒ **all** problems in NP can be solved in polynomial time (i.e., P = NP).



Coping with a “Tough” Problem: Trilogy I

“I can’t find an efficient algorithm.
I guess I’m just too dumb.”

Coping with a “Tough” Problem: Trilogy II

“I can’t find an efficient algorithm,

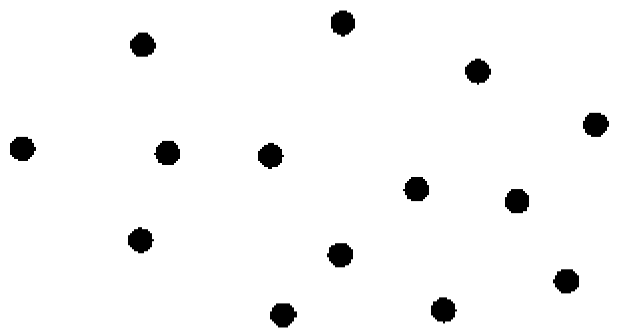
because no such algorithm is possible!”

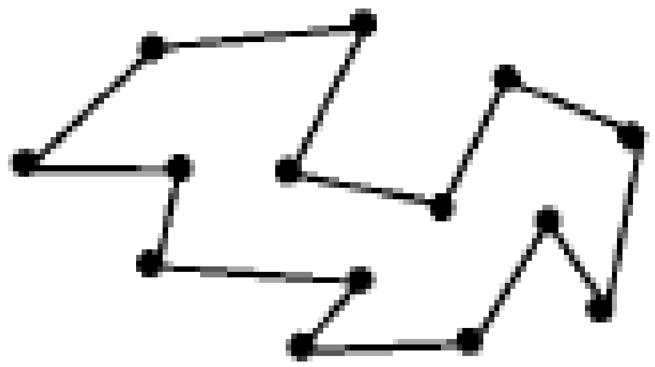
Coping with a “Tough” Problem: Trilogy III

“I can’t find an efficient algorithm, but neither can all these famous people.”

The Traveling Salesman Problem (TSP)

- **Instance:** a set of *n* cities, distance between each pair of cities, and a bound *B*.
- **Question:** is there a route that starts and ends at a given city, visits every city exactly once, and has total distance ≤ *B*?


A TSP instance


A TSP solution

NP vs. P

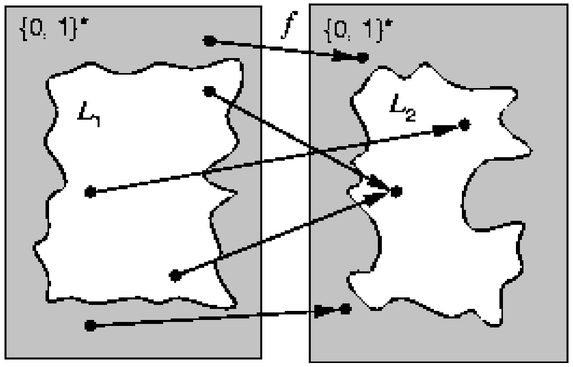
- TSP ∈ NP.
  - Need to **check** a solution (tour) in polynomial time.
    - Guess a tour.
    - Check if the tour visits every city exactly once, returns to the start, and total distance ≤ *B*.
- TSP ∈ P?
  - Need to solve (find a tour) in polynomial time.
  - Still unknown if TSP ∈ P.

Decision Problems and NP-Completeness

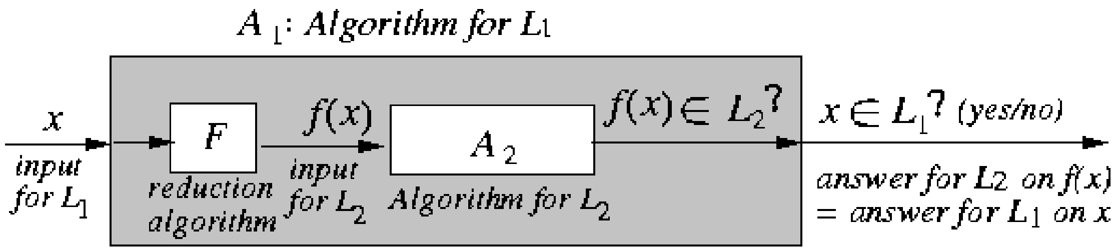
- **Decision problems:** those having yes/no answers.
  - TSP: Given a set of cities, distance between each pair of cities, and a bound *B*, **is there a route** that starts and ends at a given city, visits every city exactly once, and has total distance at most *B*?
- **Optimization problems:** those finding a legal configuration such that its cost is minimum (or maximum).
  - TSP: Given a set of cities and that distance between each pair of cities, **find the distance of a "minimum route"** that starts and ends at a given city and visits every city exactly once.
- Could apply binary search on decision problems to obtain solutions to optimization problems.
- NP-completeness is associated with decision problems.

Polynomial-time Reduction

- **Motivation:** Let *L*1 and *L*2 be two decision problems.
  Suppose algorithm *A*2 can solve *L*2. Can we use *A*2 to solve *L*1?
- **Polynomial-time reduction *f* from *L*1 to *L*2:** $L1 \le_P L2$
  - *f* reduces input for *L*1 into an input for *L*2 s.t. the reduced input is a "yes" input for *L*2 iff the original input is a "yes" input for *L*1.
    - $L1 \le_P L2$ if ∃ polynomial-time computable function *f*: $\{0, 1\}^* \rightarrow \{0, 1\}^*$ s.t. $x \in L1$ iff $f(x) \in L2$, $\forall x \in \{0, 1\}^*$.
    - ***L*2 is at least as hard as *L*1.**
- *f* is computable in polynomial time.

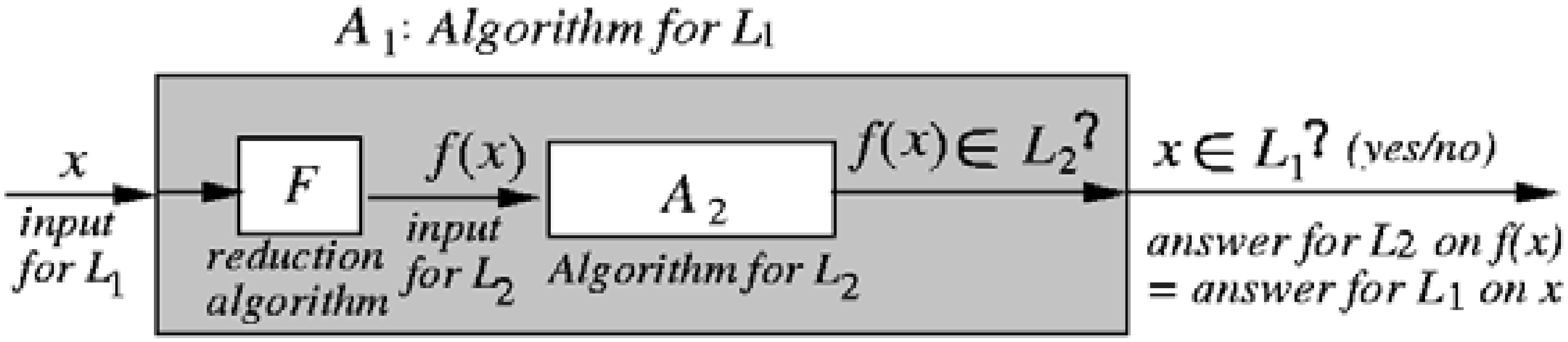


$L_1 \le_P L_2$



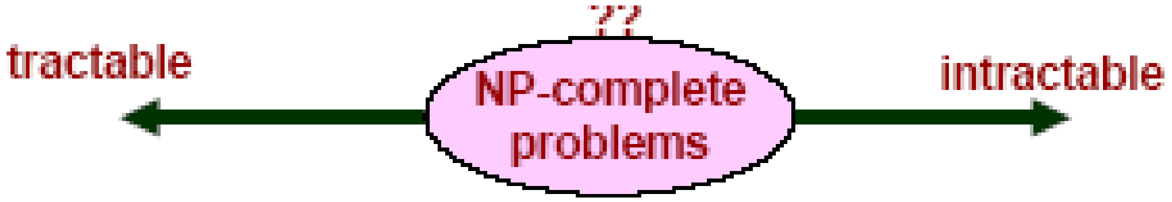
Significance of Reduction

- Significance of $L1 \leq_P L2$:
  - $\exists$ polynomial-time algorithm for $L2 \Rightarrow \exists$ polynomial-time algorithm for $L1$ ($L2 \in P \Rightarrow L1 \in P$).
  - $\nexists$ polynomial-time algorithm for $L1 \Rightarrow \nexists$ polynomial-time algorithm for $L2$ ($L1 \notin P \Rightarrow L2 \notin P$).
- $\leq_P$ is transitive, i.e., $L1 \leq_P L2$ and $L2 \leq_P L3 \Rightarrow L1 \leq_P L3$.



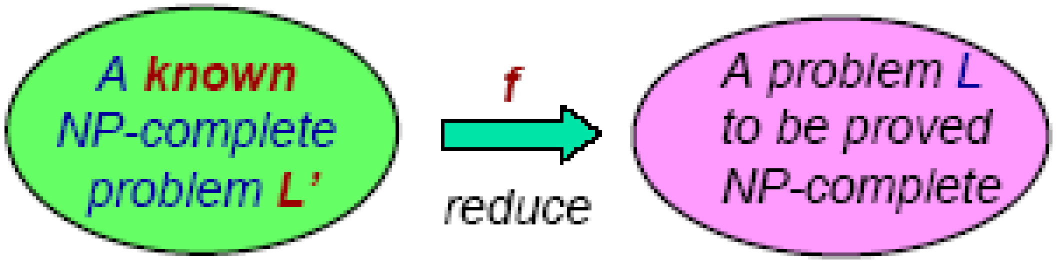
NP-Completeness

- **NP-completeness: worst-case** analyses for **decision** problems.
- A **decision** problem L is **NP-complete (NPC)** if
  1. $L \in$ NP, and
  2. $L' \le_P L$ for every $L' \in$ NP.
- **NP-hard:** If L satisfies property 2, but not necessarily property 1, we say that L is **NP-hard**.
- Suppose $L \in$ NPC.
  - If $L \in P$, then there exists a polynomial-time algorithm for every $L' \in$ NP (i.e., P = NP).
  - If $L \notin P$, then there exists no polynomial-time algorithm for any $L' \in$ NPC (i.e., P ≠ NP).



Proving NP-Completeness

- **Five steps for proving that *L* is NP-complete:**
  1. Prove $L \in$ NP. (easy)
  2. Select a known NP-complete problem ***L'***.
  3. Construct a reduction *f* transforming **every** instance of ***L'*** to an instance of *L*.
  4. Prove that $x \in$ ***L'*** iff $f(x) \in L$ for all $x \in \{0, 1\}^*$.
  5. Prove that *f* is a polynomial-time transformation.


A known NP-complete problem L'
f
reduce
A problem L to be proved NP-complete

Coping with NP-hard Problems

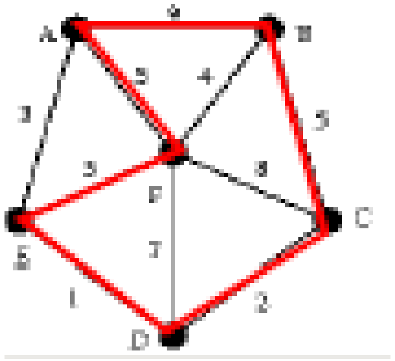
- **Exhaustive search/Branch and bound**
  - Is feasible only when the problem size is small.
- **Approximation algorithms**
  - Guarantee to be a fixed percentage away from the optimum.
  - E.g., MST for the minimum Steiner tree problem.
- **Pseudo-polynomial time algorithms**
  - Has the form of a polynomial function for the complexity, but is not to the problem size.
  - E.g., $O(nW)$ for the 0-1 knapsack problem. (W: maximum weight)
- **Restriction**
  - Work on some subset of the original problem.
  - E.g., the maximum independent set problem in circle graphs.
- **Heuristics:** No guarantee of performance.

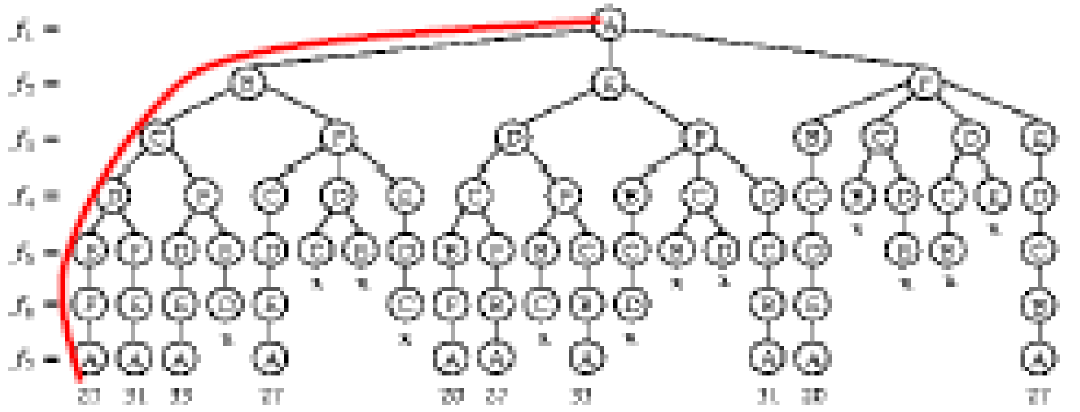
Algorithmic Paradigms

- **Exhaustive search:** Search the entire solution space.
- **Branch and bound:** A search technique with pruning.
- **Greedy method:** Pick a locally optimal solution at each step.
- **Dynamic programming:** Partition a problem into a collection of sub-problems, the sub-problems are solved, and then the original problem is solved by combining the solutions. (Applicable when the sub-problems are **NOT independent**).
- **Hierarchical approach:** Divide-and-conquer.
- **Simulated annealing:** An adaptive, iterative, non-deterministic algorithm that allows "uphill" moves to escape from local optima.
- **Genetic algorithm:** A population of solutions is stored and allowed to evolve through successive generations via mutation, crossover, etc.
- **Multilevel framework:** The bottom-up approach (coarsening) followed by the top-down one (uncoarsening); often good for handling large-scale designs.
- **Mathematical programming:** A system of solving an objective function under constraints.

Exhaustive Search vs. Branch and Bound

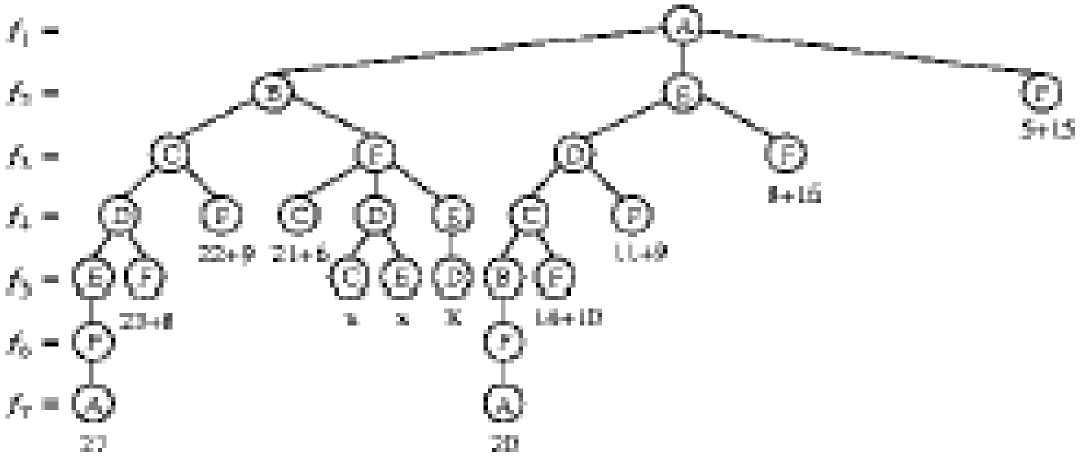
- TSP example





Backtracking/exhaustive search

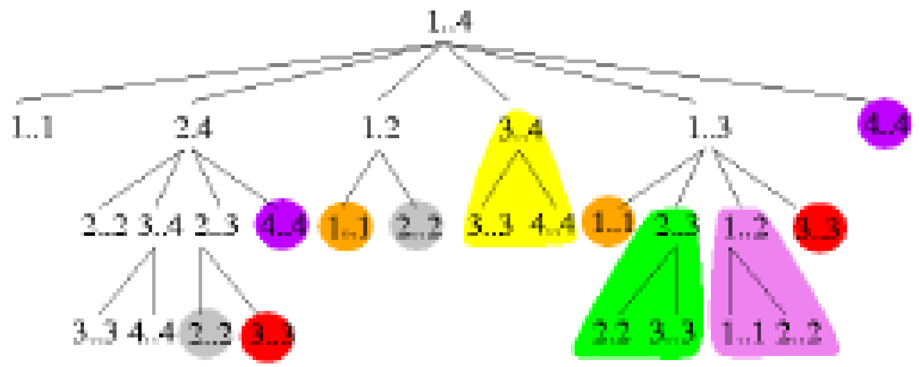
State-space trees



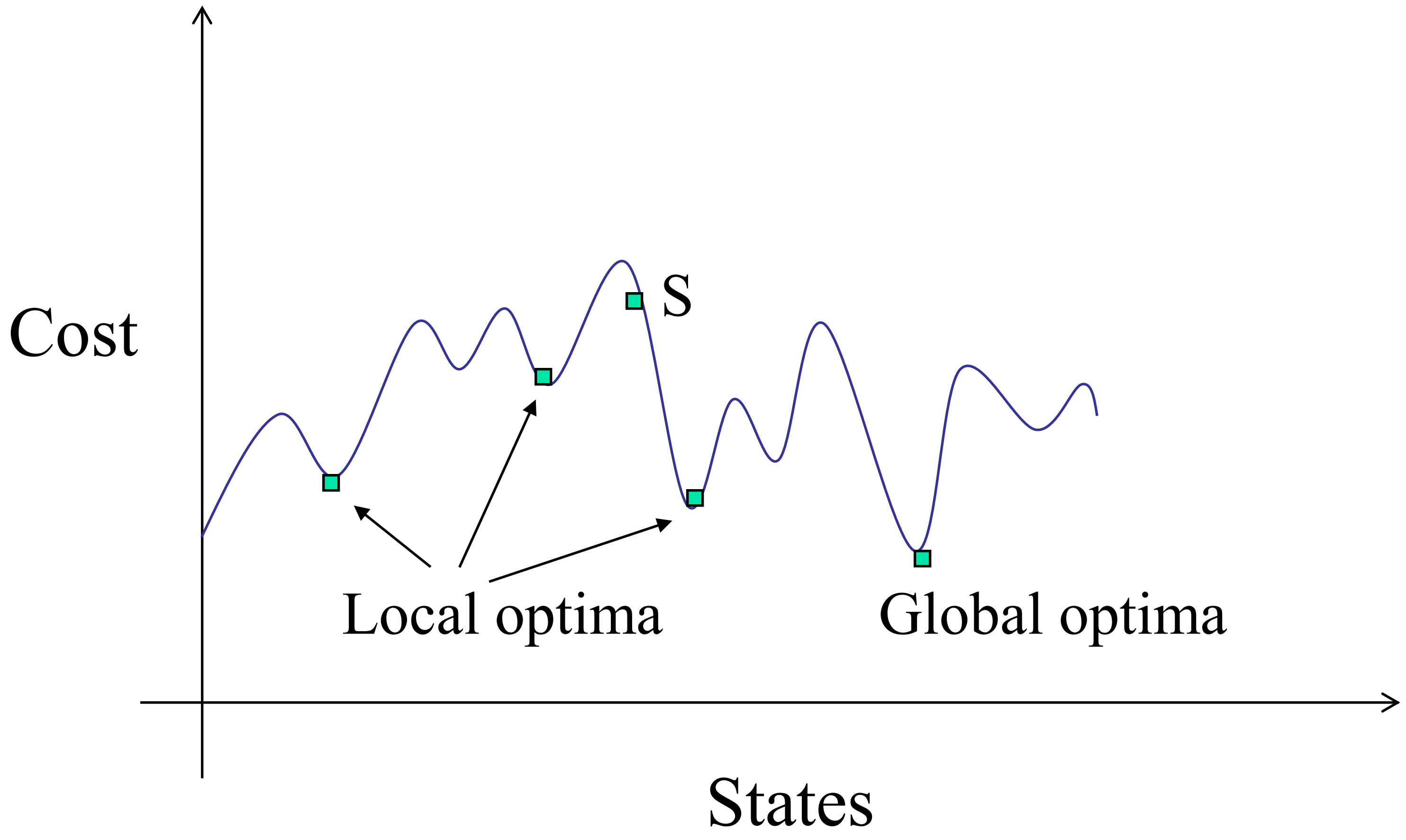
Branch and bound

Dynamic Programming (DP) vs. Divide-and-Conquer

- Both solve problems by combining the solutions to subproblems.
- Divide-and-conquer algorithms
  - Partition a problem into **independent** subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.
  - Inefficient if they solve the same subproblem more than once.
- Dynamic programming (DP)
  - Applicable when the subproblems are **not independent**.
  - DP solves each subproblem just once.



Simulated Annealing

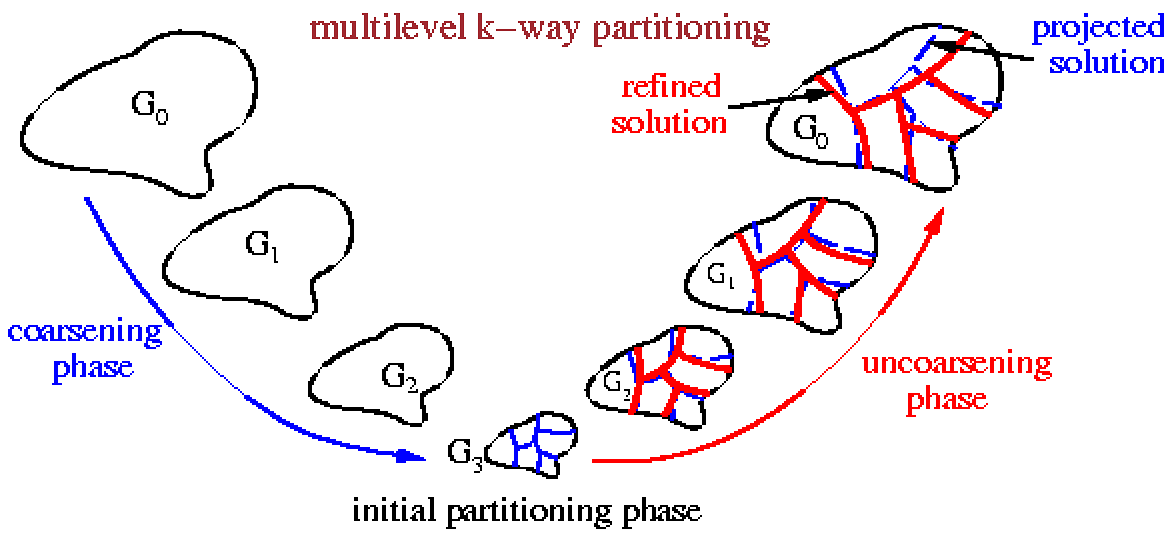


Simulated Annealing Algorithm

```
Begin
    Get an initial solution S and an initial temperature T > 0
    while not yet "frozen" do
        for 1 ≤ i ≤ P do
            Pick a random neighbor S' of S;
            Δ = Cost(S') - Cost(S)
            if Δ ≤ 0 then S ← S' // down-hill move
            if Δ > 0 then S ← S' with probability $e^{-\Delta/T}$ // up-hill
        T ← rT; // reduce temperature
    return S
End
```

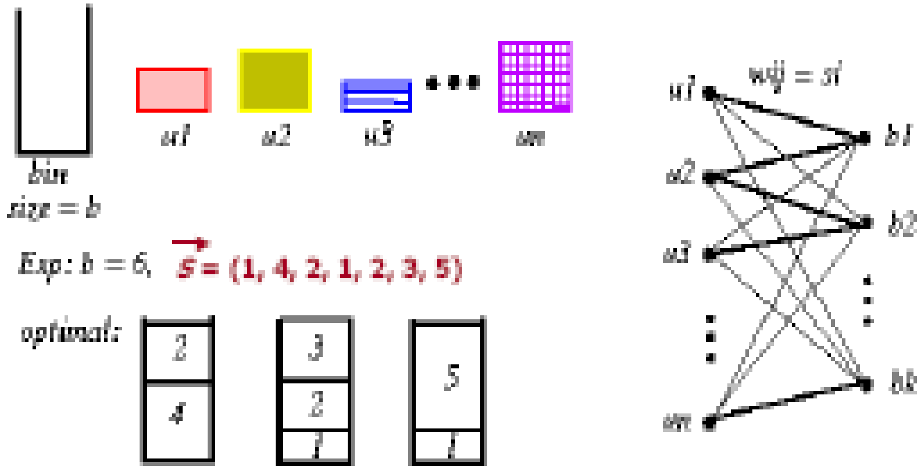
Multilevel Framework

- **Clustering:** Reduce the problem size by grouping highly connected components and treat them as a super node.
- **Multilevel partitioning**
  - **Coarsening:** Recursively clusters the instance until its size is smaller than a given threshold.
  - **Uncoarsening:** Declusters the instance while applying a partitioning refinement algorithm (e.g., F-M).

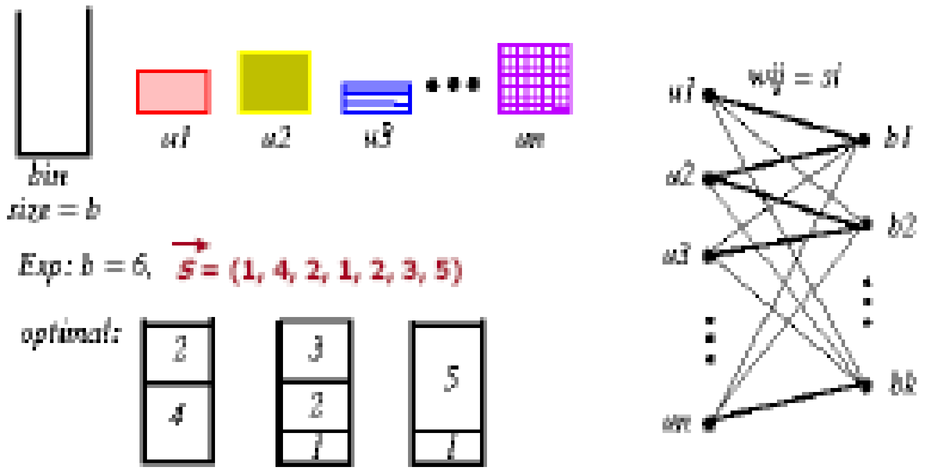


Example: Bin Packing

- **The Bin-Packing Problem Π :** Items $U = \{u1, u2, \ldots, un\}$, where u_i is of an integer size s_i; set B of bins, each with capacity b.
- **Goal:** Pack all items, minimizing # of bins used. (**NP-hard!**)



Algorithms for Bin Packing



- Greedy approximation alg.: First-Fit Decreasing (FFD)
- Dynamic Programming? Hierarchical Approach? Genetic Algorithm? …
- Mathematical Programming: Use **integer linear programming (ILP)** to find a solution using |*B*| bins, then search for the smallest feasible |*B*|.

ILP Formulation for Bin Packing

- 0-1 variable: *xij*=1 if item *ui* is placed in bin *bj*, 0 otherwise.

$$
\begin{array}{rlll}
\max & \displaystyle\sum_{(i,j)\in E} w_{ij}x_{ij} & & \\
\textit{subject to} & & & \\
& \displaystyle\sum_{\forall i\in U} w_{ij}x_{ij} \leq b_j, \forall j \in B & /*\ \textit{capacity constraint}\ */ & (1) \\
& \displaystyle\sum_{\forall j\in B} x_{ij} = 1, \forall i \in U & /*\ \textit{assignment constraint}\ */ & (2) \\
& \displaystyle\sum_{ij} x_{ij} = n & /*\ \textit{completeness constraint}\ */ & (3) \\
& x_{ij} \in \{0,1\} & /*\ 0,\ 1\ \textit{constraint}\ */ & (4)
\end{array}
$$

- **Step 1:** Set |*B*| to the lower bound of the # of bins.
- **Step 2:** Use the ILP to find a feasible solution.
- **Step 3:** If the solution exists, the # of bins required is |*B*|. Then exit.
- **Step 4:** Otherwise, set |*B*| ← |*B*| + 1. Goto Step 2.

Basic Graph Algorithms

- Basic terminology and representations
- Graph search algorithms
- Spanning tree algorithms
- Shortest path algorithms
- Maximum flow and matching
- Steiner tree algorithms

- References:
  - "Algorithms in C++" 3rd ed by R. Sedgewick
  - "Introduction to algorithms" 2nd ed by Cormen et.al.
  - "Introduction to the design and analysis of algorithms" 2nd ed by Levitin

Basic Terminology (1/3)

- A ***graph*** is a pair of sets G(V,E) where V is the set of vertices, and E [(u,v)]is a set of pair of distinct vertices called edges.
- A ***complete graph*** on n vertices is a graph in which every vertex is adjacent to every other vertex. (Denoted by K_n)
- A graph G’ = (V’,E’) is a ***subgraph*** of G iff V’ is a subset of V, and E’ is a subset of E.
- A ***walk*** P of a graph G is defined as a finite alternating sequence P = $v_0, e_1, \ldots, e_k, v_k$.
- A walk is an ***open walk*** if the terminal vertices (starting and ending) are distinct.
- A ***path*** is an open walk in which no vertex appears more than once.

Basic Terminology (2/3)

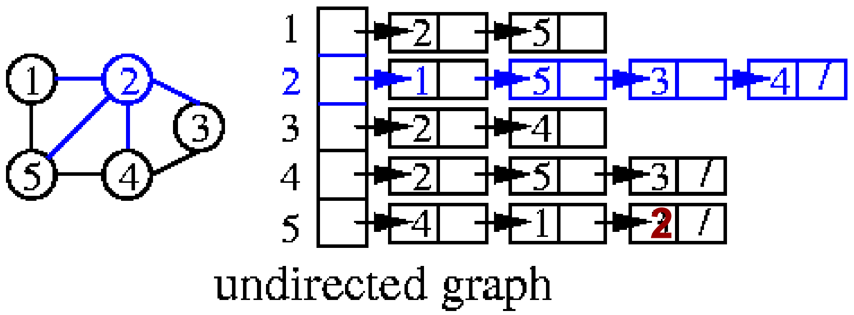
- The ***length*** of a path is the number of edges in it.
  - A path is a (u,v) path if u and v are the terminal vertices.
- A ***cycle*** is a path (v_0,v_k) of length k (k > 2) where $v_0 = v_k$.
  - Odd cycle if k is odd, Even cycle if k is even.
- A ***connected component*** of G is a subgraph of G that has a path from each vertex to every other vertex.
- An edge e in E is called a ***cut edge*** in G if its removal from G increases the number of connected components of G by at least one.
- A graph is called ***planar*** if it can be drawn in the plane without any two edges crossing

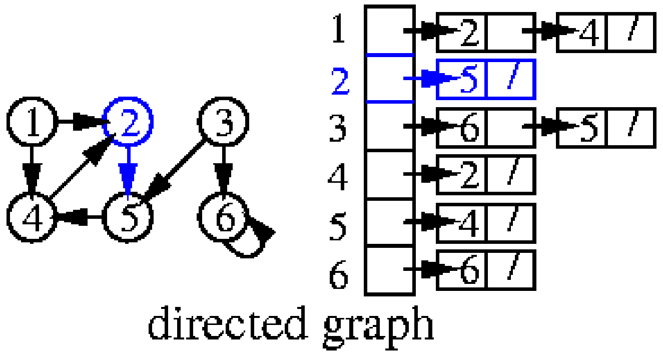
Basic Terminology (3/3)

- A ***tree*** is a connected subgraph with no cycles.
- A ***directed graph*** is a pair of sets (V,E) where E is a set of ordered pairs of distinct vertices, called directed edges.
- A ***directed acyclic graph*** (DAG) is a directed graph with no cycles.
- ***Hypergraph*** is a pair (V,E) where V is a set of vertices, and E is a family of sets of vertices.
  - Each e in E denoted by {v_0, v_1,…,v_k} is called a net.
- A ***bipartite graph*** is a graph that can be partitioned in to two sets X, and Y so that each edge has one end in X, and the other end in Y.
- Graph Problem – $G = (V,E)$, find a subset $V'/E' \subseteq V/E \rightarrow V'/E'$ has a property $\wp$

Representations of Graphs: Adjacency List

- **Adjacency list:** An array *Adj* of |*V*| lists, one for each vertex in *V*. For each $u \in V$, *Adj*[*u*] pointers to all the vertices adjacent to *u*.
- Advantage: *O*(*V*+*E*) storage, good for **sparse** graph.
- Drawback: Need to traverse list to find an edge.



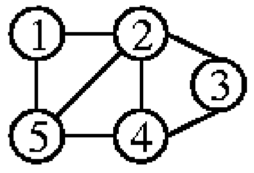


Representations of Graphs: Adjacency Matrix

- **Adjacency matrix:** A $|V| \times |V|$ matrix $A = (a_{ij})$ such that

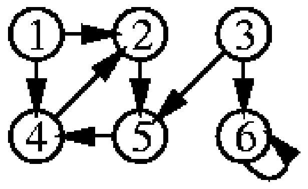
$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

- Advantage: $O(1)$ time to find an edge.
- Drawback: $O(V^2)$ storage, more suitable for **dense** graph.
- How to save space if the graph is undirected?



$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

undirected graph



$$\begin{array}{c|cccccc|} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 & 1 \\ 4 & 0 & 1 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 1 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline \end{array}$$

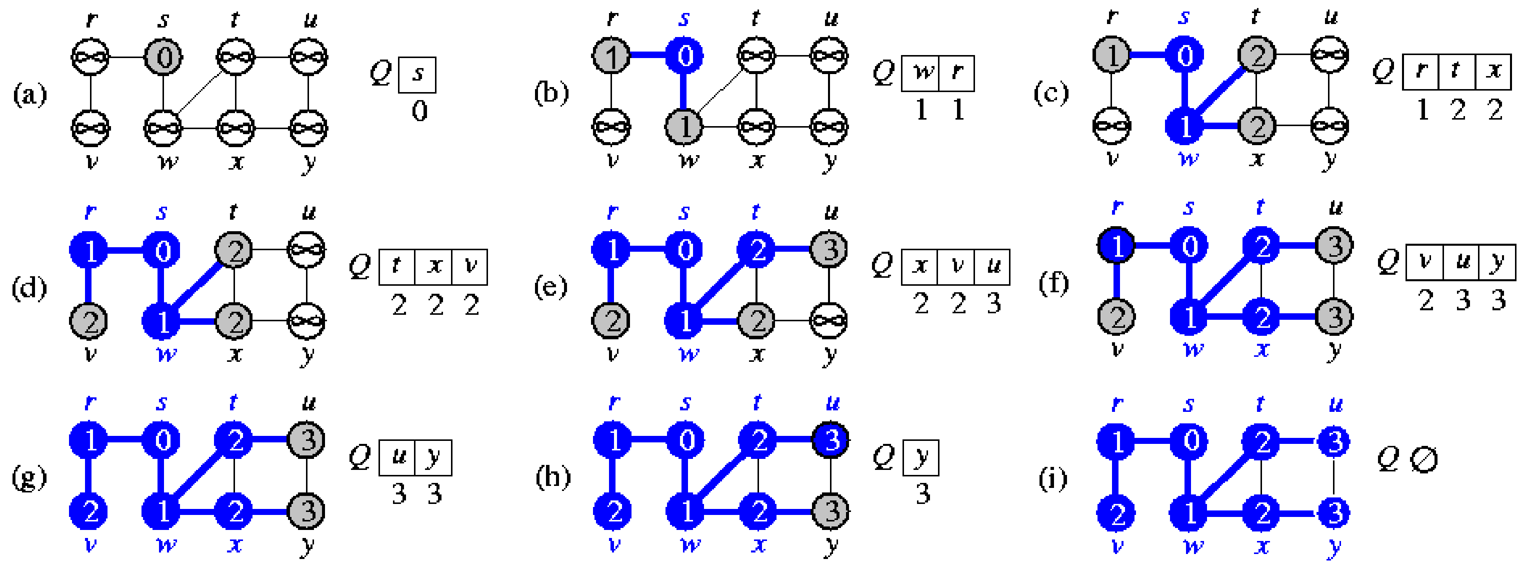
directed graph

Breadth-First Search (BFS)

```
BFS($G$, $s$)
1.  for each vertex $u \in V[G]-\{s\}$ do
2.      $color[u] \leftarrow$ WHITE
3.      $d[u] \leftarrow \infty$
4.      $\pi[u] \leftarrow$ NIL
5.  $color[s] \leftarrow$ GRAY
6.  $d[s] \leftarrow 0$
7.  $\pi[s] \leftarrow$ NIL
8.  $Q \leftarrow \varnothing$
9.  Enqueue($Q$, $s$)
10. while $Q \neq \varnothing$ do
11.   $u \leftarrow$ Dequeue[$Q$]
12.   for each $v \in Adj[u]$ do
13.       if $color[v] =$ WHITE then
14.           $color[v] \leftarrow$ GRAY
15.           $d[v] \leftarrow d[u]+1$
16.           $\pi[v] \leftarrow u$
17.           Enqueue($Q$, $v$)
18.   $color[u] \leftarrow$ BLACK
```

- *color*[*u*]: white (undiscovered) $\rightarrow$ gray (discovered) $\rightarrow$ black (explored: out edges are all discovered)
- $d[u]$: distance from source s; $\pi[u]$: predecessor of u.
- Use queue for gray vertices.
- Time complexity: $O(V+E)$ (adjacency list).

BFS Example



- *color*[*u*]: white (undiscovered) → gray (discovered) → black (explored: out edges are all discovered)
- Use queue *Q* for gray vertices.
- Time complexity: *O*(*V*+*E*) (adjacency list) using aggregate analysis
  - Each vertex enqueued and dequeued once: *O*(*V*) time.
  - Each edge considered once: *O*(*E*) time.
- Breadth-first tree: $G_\pi = (V_\pi, E_\pi)$, $V_\pi = \{v \in V \mid \pi[v] \neq \text{NIL}\} \cup \{s\}$, $E_\pi = \{(\pi[v], v) \in E \mid v \in V_\pi - \{s\}\}$.

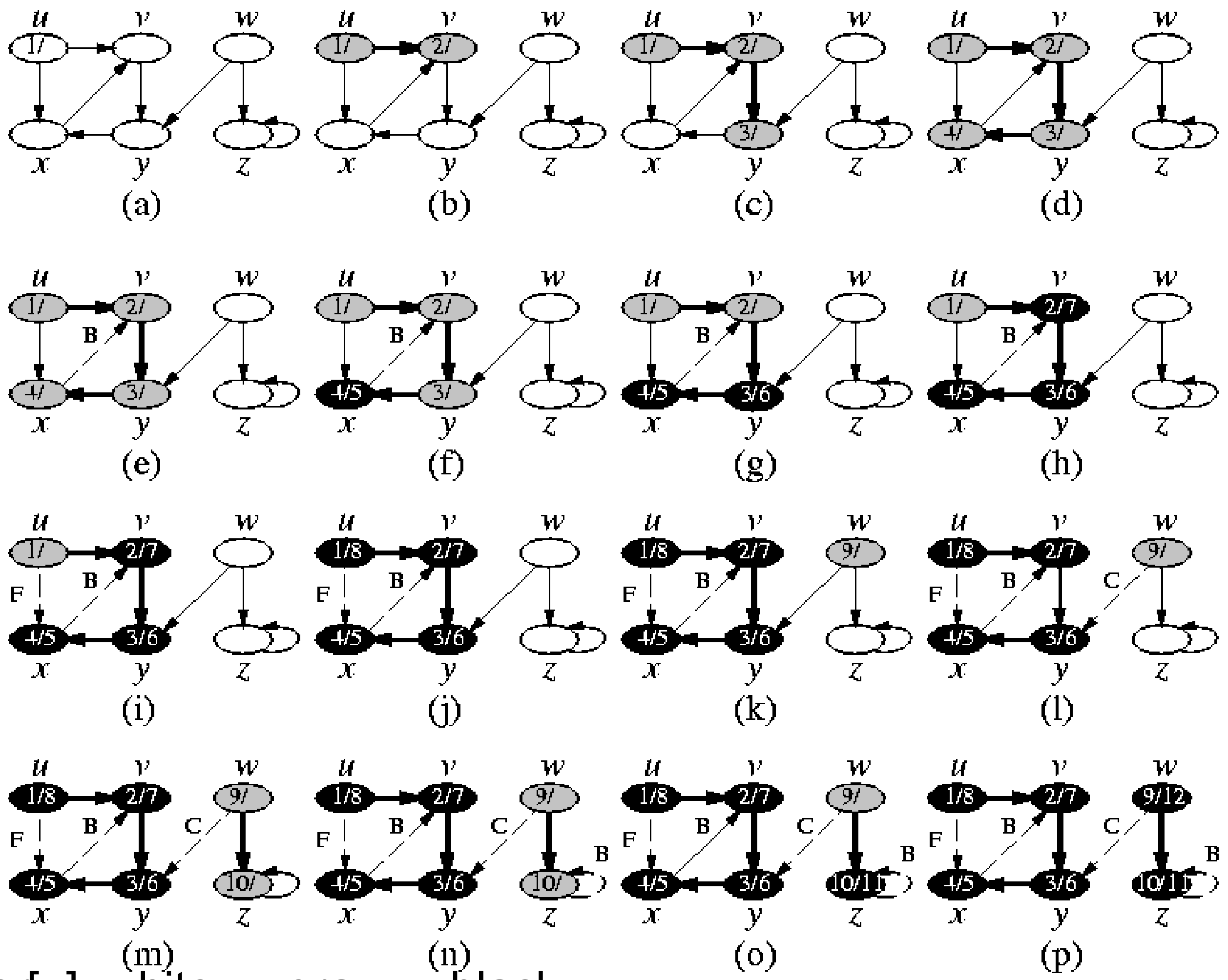
Depth-First Search (DFS)

```
DFS(G)
1. for each vertex $u \in V[G]$ do
2.    $color[u] \leftarrow$ WHITE
3.    $\pi[u] \leftarrow$ NIL
4. $time \leftarrow 0$
5. for each vertex $u \in V[G]$ do
6.     if $color[u]$ = WHITE then
7.         DFS-Visit($u$)

DFS-Visit($u$)
1. $color[u] \leftarrow$ GRAY
  /* white vertex $u$ has just been
      discovered. */
2. $d[u] \leftarrow time \leftarrow time + 1$
3. for each vertex $v \in Adj[u]$ do
      /* Explore edge $(u,v)$. */
4.    if $color[v]$ = WHITE then
5.        $\pi[v] \leftarrow u$
6.        DFS-Visit($v$)
7. $color[u] \leftarrow$ BLACK
   /* Blacken $u$; it is finished. */
8. $f[u] \leftarrow time \leftarrow time + 1$
```

- *color*[*u*]: white (undiscovered) $\rightarrow$ gray (discovered) $\rightarrow$ black (explored: out edges are all discovered)
- *d*[*u*]: discovery time (gray); *f*[*u*]: finishing time (black); $\pi[u]$: predecessor.
- Time complexity: $O(V+E)$ (adjacency list).

DFS Example



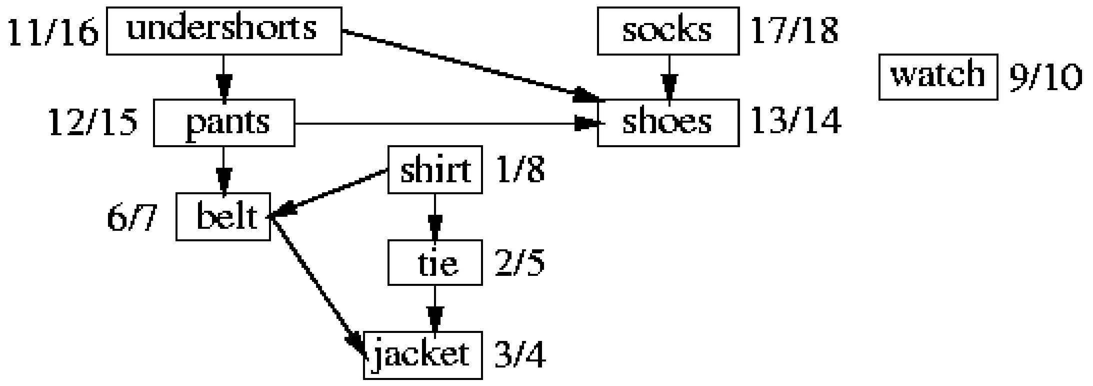
- *color*[*u*]: white → gray → black.
- Depth-first **forest**: $G_\pi = (V, E_\pi)$, $E_\pi = \{(\pi[v], v) \in E \mid v \in V, \pi[v] \neq \text{NIL}\}$.

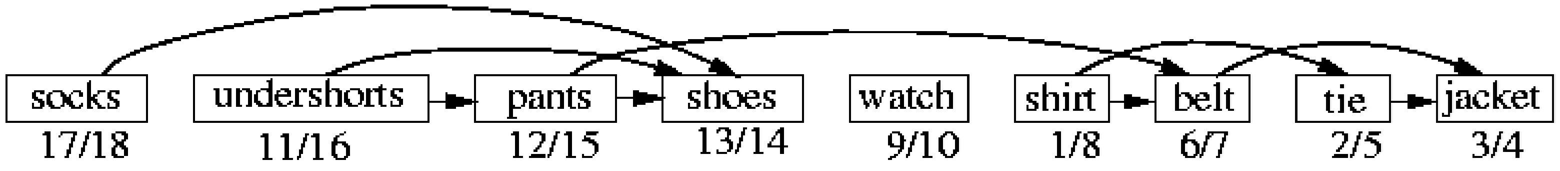
Topological Sort

- A **topological sort** of a directed acyclic graph (DAG) $G = (V, E)$ is a linear ordering of V s.t. $(u, v) \in E$ □ u appears before v.

```
Topological-Sort(G)
1. call DFS(G) to compute finishing times f[v] for each vertex v
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices
```

- Time complexity: $O(V+E)$ (adjacency list).
- Correctness: Any edge (u, v) in a dag, we have $f[v] < f[u]$.





Vertices are arranged from left to right in order of decreasing finishing times.

Topological Sort: Another Way

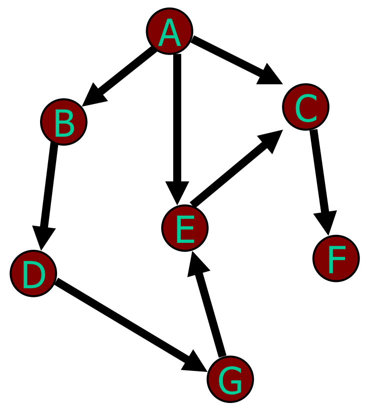
- A directed acyclic graph always contains a vertex with indegree 0.

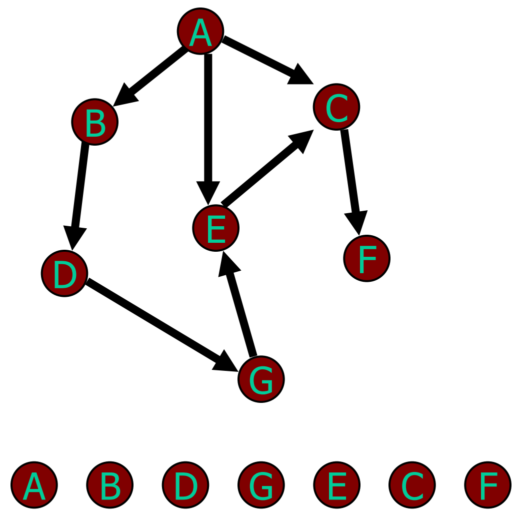
```
Topological-Sort2(G)
1. Call DFS(G) to compute indegree[v] for each vertex $v \in V[G]$
2. $Q \leftarrow \emptyset$
3. $label \leftarrow 0$
4. for each vertex $v \in V[G]$ do
5.       if $indegree[v] = 0$ then
6.          Enqueue(Q, v)
7. while $Q \neq \emptyset$ do
8.     $u \leftarrow$ Dequeue(Q)
9.     $label[u] \leftarrow label \leftarrow label+1$
10.    for each $v \in Adj[u]$ do
11.        $indegree[v] = indegree[v]-1$
12.        if $indegree[v] = 0$ then
13.           Enqueue(Q, v)
```

- Time complexity: $O(V+E)$ (adjacency list).

Topological Sort Illustration

- Topological Search/Sort (DAG only)
  - A node is visited when all its parents are visited
  - Two algorithms:
    - Simple application of DFS: perform DFS traversal and note the order in which vertices become dead ends (popped off the traversal stack)
    - Direct implementation of the decrease-and conquer technique: repeatedly, identify in a remaining digraph a node which has no incoming edges, and delete it along with all the edges outgoing from it.



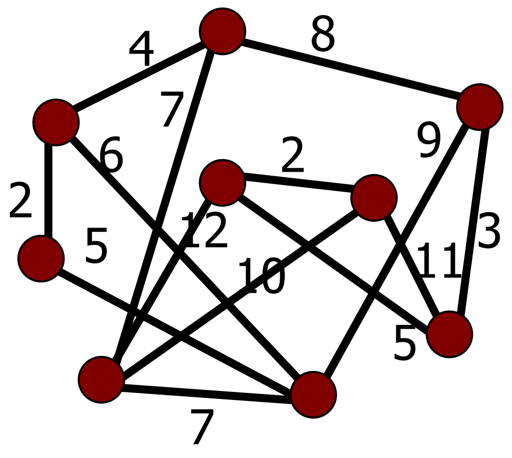


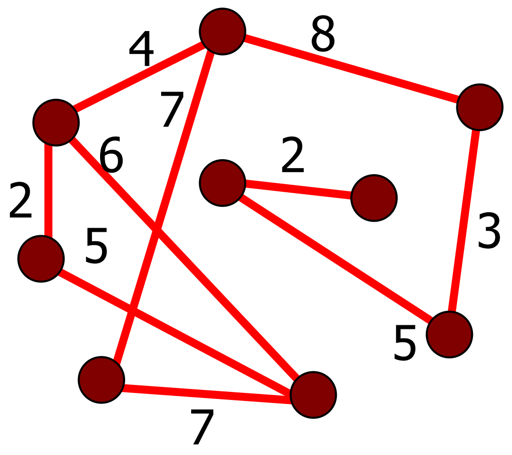
Spanning Tree Algorithms

- Minimum Spanning Tree (MST) – $\wp: E'$ induces a tree and $\Sigma_{e_i \in E'} wt(e_i)$ is minimum over all such trees
- Kruskal's Algorithm (greedy)
  - *n* sets (*n* nodes) where each represents a partial spanning tree
  - Select an edge to merge two spanning trees until all sets join together to be a single tree
- O(|E|log|E|)
  - Sorting edges dominates: O(|E|log|E|) = O(|E|log|V|) ($|E| < |V|^2$)

Kruskal's Spanning Tree Algorithm

```
Algorithm MST()
begin
    $E$ = {All the edges are in a non-decreasing
          weight-sorted order};
    for each node $N_i$  $T_i = \{N_i\}$;
    $n$ = the number of nodes; $Sum = 1$;
    while ($Sum \neq n$)
        Select $e_0$, say $(N_i, N_j)$, and $E = E - \{e_0\}$;
        where $N_i \in T_m$, $N_j \in T_n$;
        if ($m == n$)    continue;
        $T_m = T_m \cup T_n$; $T_n = \varnothing$; $Sum$++;
end
```



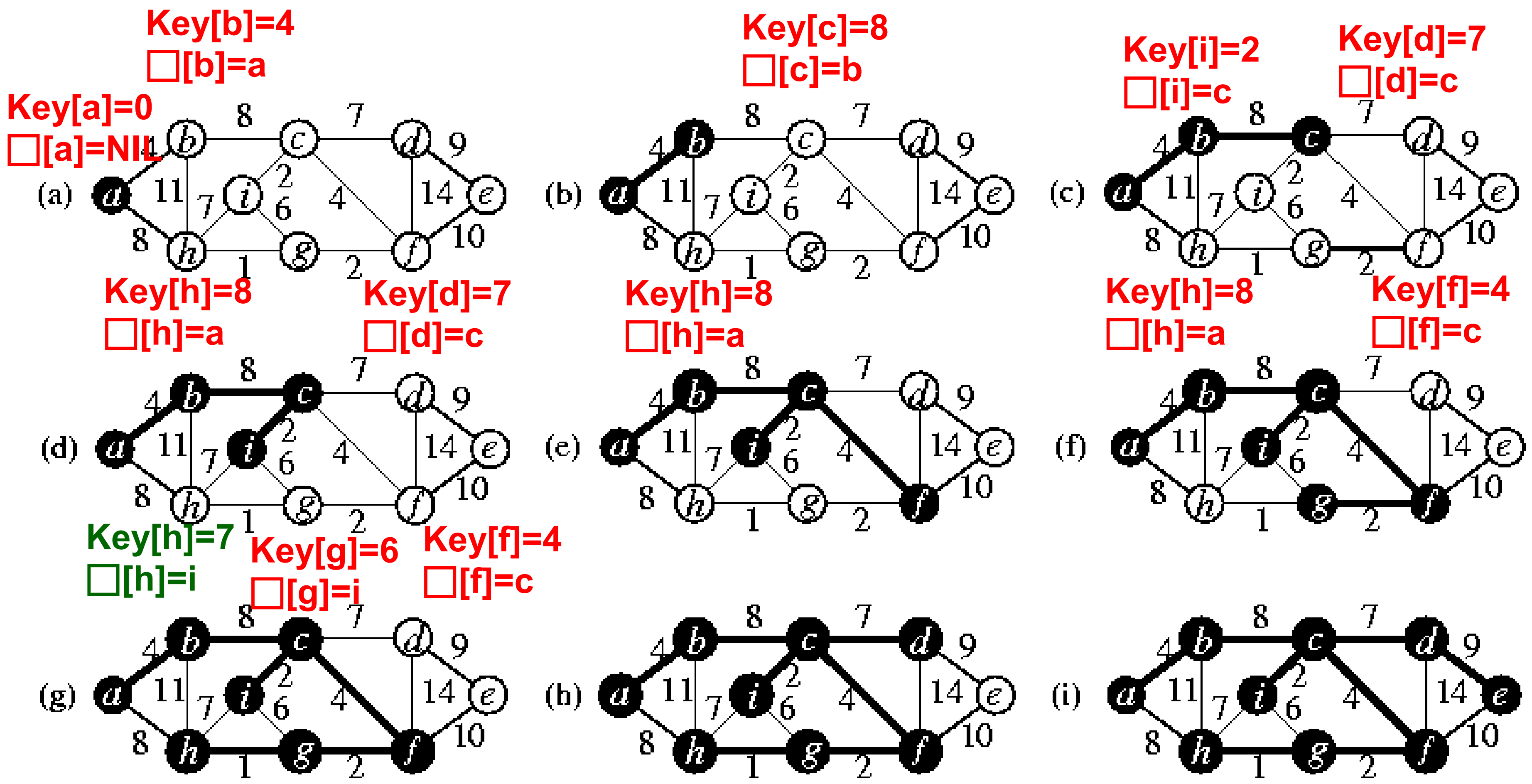


Prim's (Prim-Dijkstra's?) MST Algorithm

```
MST-Prim($G,w,r$)
/* $Q$: min-priority queue for vertices not in the tree, based on key[]. */
/* $key$: min weight of any edge connecting to a vertex in the tree. */
1. for each vertex $u \in V[G]$ do
2.     $key[u] \leftarrow \infty$
3.     $\pi[u] \leftarrow$ NIL
4. $key[r] \leftarrow 0$
5. $Q \leftarrow V[G]$
6. while $Q \neq \emptyset$ do
7.     $u \leftarrow$ Extract-Min($Q$)
8.     for each vertex $v \in Adj[u]$ do
9.         if $v \in Q$ and $w(u,v) < key[v]$ then
10.            $\pi[v] \leftarrow u$
11.            $key[v] \leftarrow w(u,v)$
```

- Starts from a vertex and grows until the **tree** spans all the vertices.
  - The edges in A always form a single tree.
  - At each step, a safe, a light edge connecting a vertex in A to an isolated vertex in $V - A$ is added to the tree.
  - $A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$

Example: Prim's MST Algorithm



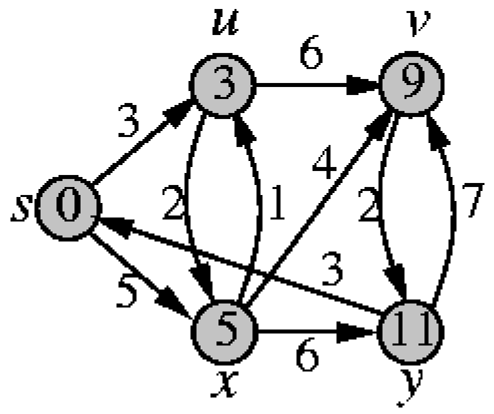
Time Complexity of Prim's MST Algorithm

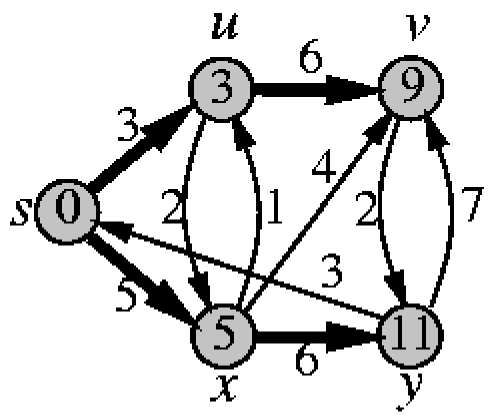
```
MST-Prim($G,w,r$)
1. for each vertex $u \in V[G]$ do
2.     $key[u] \leftarrow \infty$
3.     $\pi[u] \leftarrow$ NIL
4. $key[r] \leftarrow 0$
5. $Q \leftarrow V[G]$
6. while $Q \neq \emptyset$ do
7.     $u \leftarrow$ Extract-Min($Q$)
8.     for each vertex $v \in Adj[u]$ do
9.         if $v \in Q$ and $w(u,v) < key[v]$ then
10.            $\pi[v] \leftarrow u$
11.            $key[v] \leftarrow w(u,v)$
```

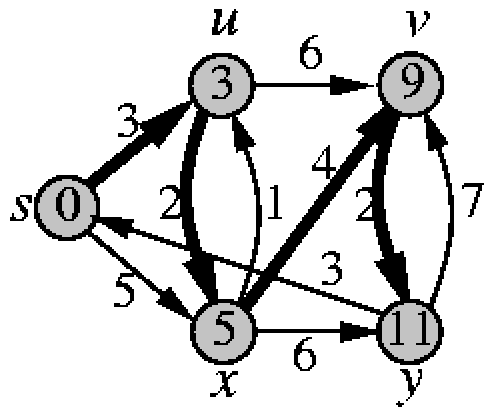
- Q is implemented as a binary min-heap: $O(E \lg V)$.
  - Lines 1—5: $O(V)$.
  - Line 7: $O(\lg V)$ for Extract-Min, so $O(V \lg V)$ with the **while** loop.
  - Lines 8—11: $O(E)$ operations, each takes $O(\lg V)$ time for Decrease-Key (maintaining the heap property after changing a node).
- Q is implemented as a Fibonacci heap: $O(E + V \lg V)$. (**Fastest to date!**)
- $|E| = O(V)$ □only $O(E \lg^* V)$ time. (Fredman & Tarjan, 1987)

Single-Source Shortest Paths (SSSP)

- **The Single-Source Shortest Path (SSSP) Problem**
  - **Given:** A **directed** graph *G=(V, E)* with edge weights, and a specific **source node** *s*.
  - **Goal:** Find a minimum weight path (or cost) from *s* to every other node in *V*.
- Applications: weights can be distances, times, wiring cost, delay. etc.
- **Special case:** BFS finds shortest paths for the case when all edge weights are 1 (the same).





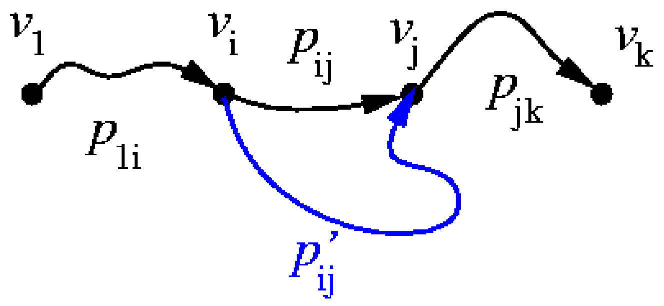


Variants on Shortest-Paths Problem

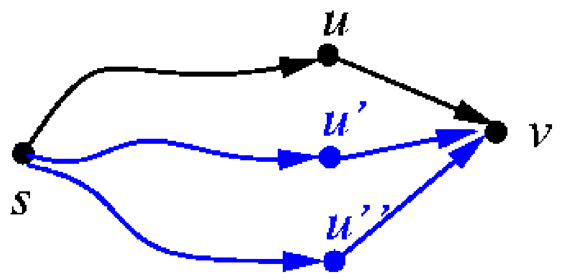
- Single-source shortest-paths problem
- Single-destination shortest-paths problem
- Single-pair shortest-path problem
- All-pairs shortest-paths problem

Optimal Substructure of a Shortest Path

- Subpaths of shortest paths are shortest paths.
  - Let $p = <v_1, v_2, \ldots, v_k>$ be a shortest path from vertex v_1 to vertex v_k, and let $p_{ij} = <v_i, v_{i+1}, \ldots, v_j>$ be the subpath of p from vertex v_i to vertex v_j, $1 \le i \le j \le k$. Then, p_{ij} is a shortest path from v_i to v_j.
- Suppose that a shortest path p from a source s to a vertex v can be decomposed into $s \overset{p'}{\leadsto} u \to v$. Then, $\delta(s, v) = \delta(s, u) + w(u, v)$.
- For all edges $(u, v) \in E$, $\delta(s, v) \le \delta(s, u) + w(u, v)$.



subpaths of shortest paths

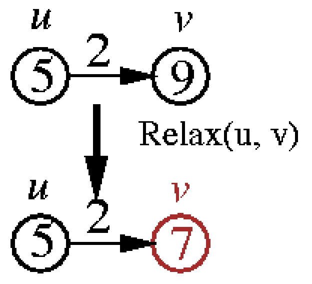


Relaxation

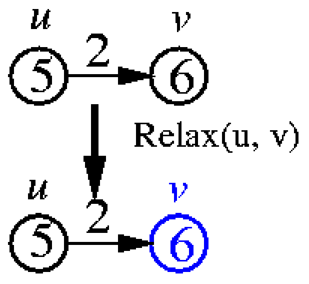
```
Initialize-Single-Source($G$, $s$)
1. for each vertex $v \in V[G]$ do
2.     $d[v] \leftarrow \infty$
       /* shortest-path estimate, upper
          bound on the weight of a shortest
          path from $s$ to $v$ */
3.     $\pi[v] \leftarrow$ NIL /* predecessor of $v$ */
4. $d[s] \leftarrow 0$
```

```
Relax($u$, $v$, $w$)
1. if $d[v] > d[u]+w(u, v)$ then
2.     $d[v] \leftarrow d[u]+w(u, v)$
3.     $\pi[v] \leftarrow u$
```

- $d[v] \leq d[u] + w(u, v)$ after calling Relax(u, v, w).
- $d[v] \geq \delta(s, v)$ during the relaxation steps; once $d[v]$ achieves its lower bound $\delta(s, v)$, it never changes.
- Let $s \rightsquigarrow u \rightarrow v$ be a shortest path. If $d[u] = \delta(s, u)$ prior to the call Relax(u, v, w), then $d[v] = \delta(s, v)$ after the call.



$d[v] > d[u] + w(u, v)$

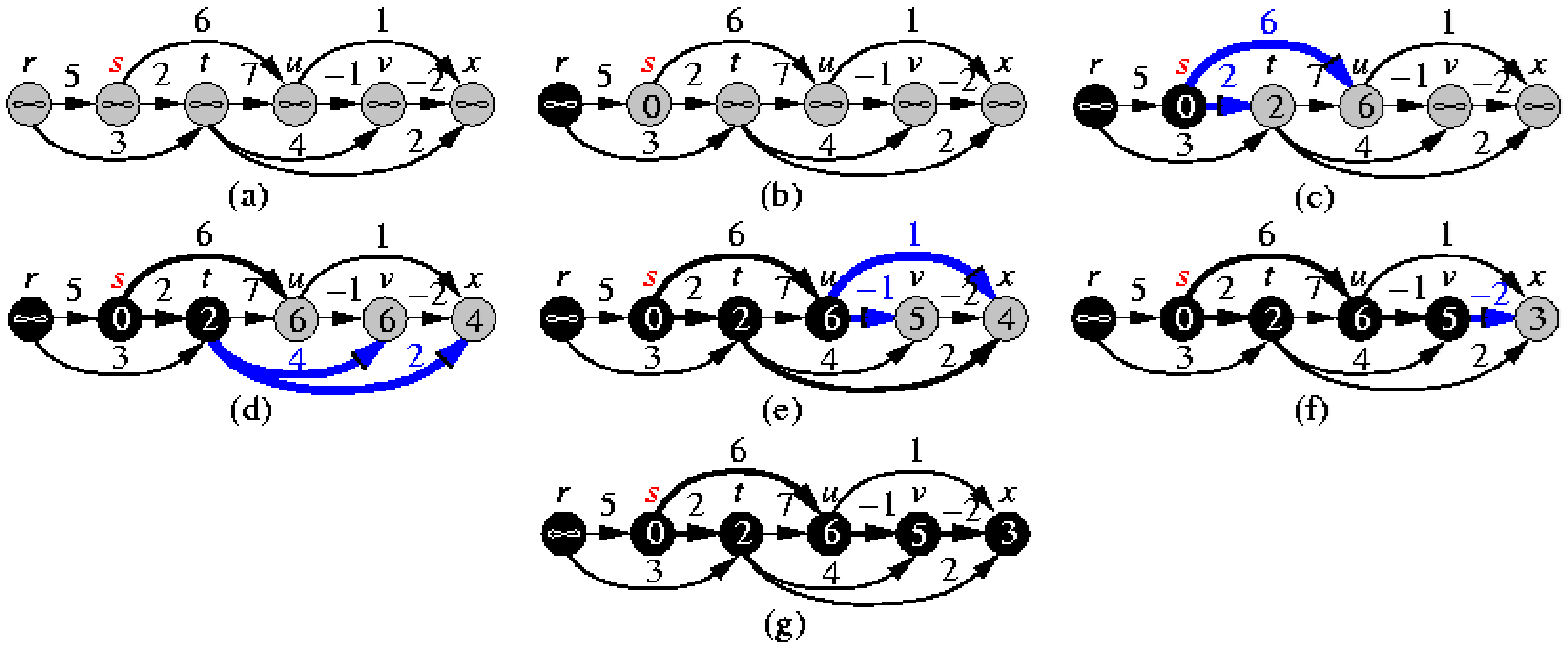


$d[v] <= d[u] + w(u, v)$

SSSPs in Directed Acyclic Graphs (DAGs)

```
DAG-Shortest-Paths($G$, $w$, $s$)
1. topologically sort the vertices of $G$
2. Initialize-Single-Source(G, $s$)
3. for each vertex $u$ taken in topologically sorted order do
4.     for each vertex $v \in Adj[u]$ do
5.         Relax($u$, $v$, $w$)
```

- Time complexity: $O(V+E)$ (adjacency-list representation).
- What if critical paths?

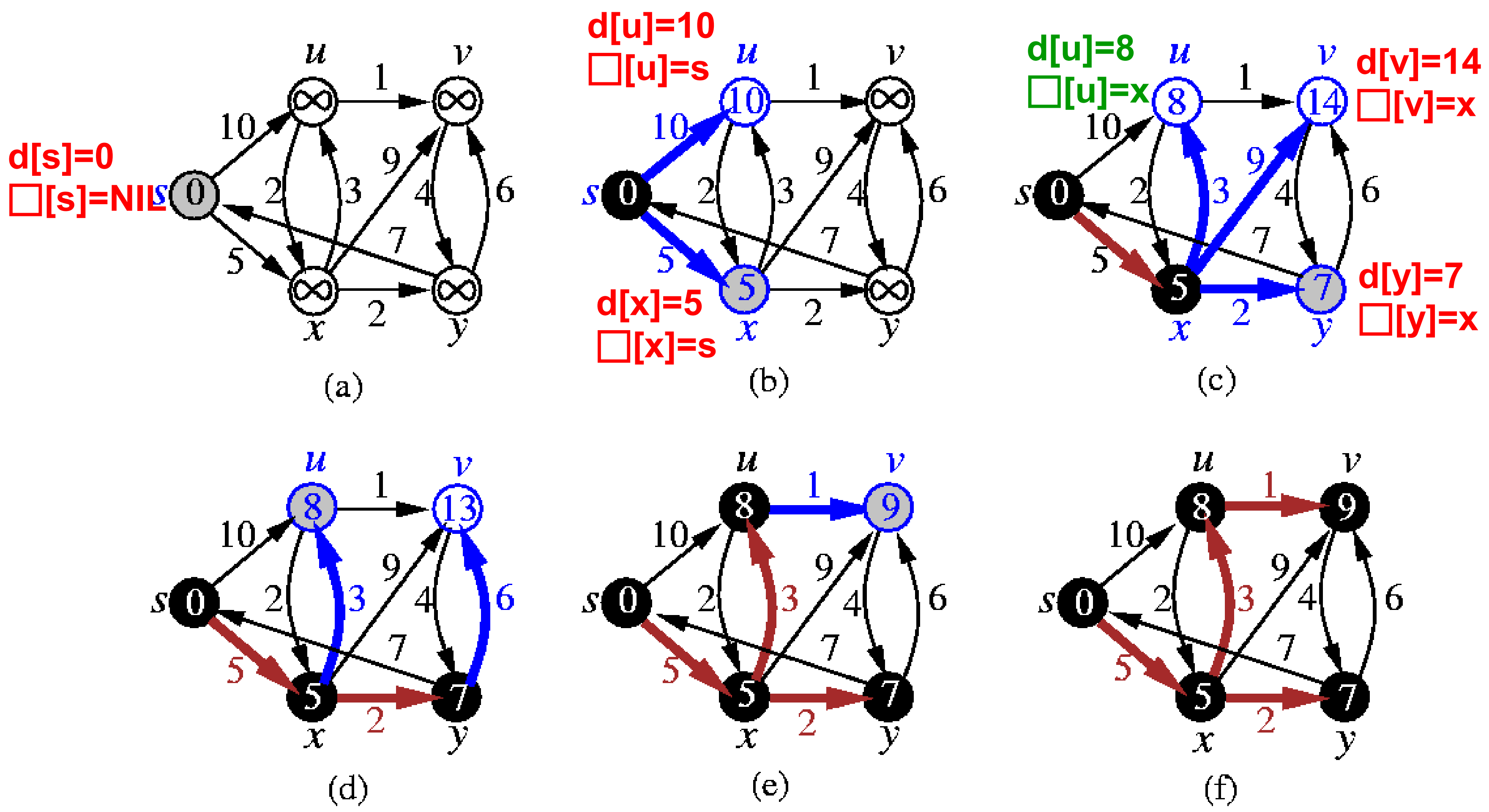


Dijkstra's Shortest-Path Algorithm

```
Dijkstra($G$, $w$, $s$)
/* $S$: final shortest-path weights determined */
/* $Q$: min-priority queue of $V$-$S$, keyed by $d$ values */
1. Initialize-Single-Source($G$, $s$)
2. $S \leftarrow \varnothing$
3. $Q \leftarrow V[G]$
4. while $Q \neq \varnothing$ do
5.     $u \leftarrow$ Extract-Min($Q$)
6.     $S \leftarrow S \cup \{u\}$
7.     for each vertex $v \in Adj[u]$ do
8.         Relax($u$, $v$, $w$)
```

- Combines a greedy and a dynamic-programming schemes.
  - Loop invariant: at the start of each iteration of the while loop, $d[v] = \delta(s, v)$ for each vertex $v \in S$.
- Works only when all **edge weights are nonnegative.**
- Executes essentially the same as Prim's algorithm.
  - Except the definition of key values.
- Naive analysis: $O(V^2)$ time by using adjacency lists.

Example: Dijkstra's Shortest-Path Algorithm



Runtime Analysis of Dijkstra's Algorithm

```
Dijkstra($G, w, s$)
1. Initialize-Single-Source($G, s$)
2. $S \leftarrow \varnothing$
3. $Q \leftarrow V[G]$
4. while $Q \neq \varnothing$ do
5.     $u \leftarrow$ Extract-Min($Q$)
6.     $S \leftarrow S \cup \{u\}$
7.     for each vertex $v \in Adj[u]$ do
8.         Relax($u, v, w$)
```

- Q is implemented as a linear array: $O(V^2)$.
  - Line 5: $O(V)$ for Extract-Min, so $O(V^2)$ with the **while** loop.
  - Lines 7—8: $O(E)$ operations, each takes $O(1)$ time.
- Q is implemented as a binary heap: $O(E \lg V)$.
  - Line 5: $O(\lg V)$ for Extract-Min, so $O(V \lg V)$ with the **while** loop.
  - Lines 7—8: $O(E)$ operations, each takes $O(\lg V)$ time for Decrease-Key (maintaining the heap property after changing a node).
- Q is implemented as a Fibonacci heap: $O(E + V \lg V)$.

All-Pairs Shortest Paths (APSP)

- **The All-Pairs Shortest Path (APSP) Problem**
  - **Given:** A **directed** graph G=(V, E) with edge weights
  - **Goal:** Find a minimum weight path (or cost) between **every pair** of vertices in V.
- **Method 1:** Extends the SSSP algorithms.
  - No negative-weight edges: Run Dijkstra's algorithm $|V|$ times, once with each $v \in V$ as the source.
    - Adjacency **list** + Fibonacci heap: $O(V^2 \lg V + VE)$ time.
  - With negative-weight edges: Run the Bellman-Ford algorithm $|V|$ times, once with each $v \in V$ as the source.
    - Adjacency **list**: $O(V^2 E)$ time.
- **Method 2:** Applies the Floyd-Warshall algorithm (negative-weight edges allowed).
  - Adjacency **matrix**: $O(V^3)$ time.
- **Method 3:** Applies Johnson's algorithm for sparse graphs (negative-weight edges allowed).
  - Adjacency **list**: $O(V^2 \lg V + VE)$ time.

Overview of Floyd-Warshall APSP Algorithm

- Applies dynamic programming.
  1. Characterize the structure of an optimal solution?
  2. Recursively define the value of an optimal solution?
  3. Compute the value of an optimal solution in a bottom-up fashion?
  4. Construct an optimal solution from computed information?
- Uses adjacency matrix *A* for *G* = (*V*, *E*):

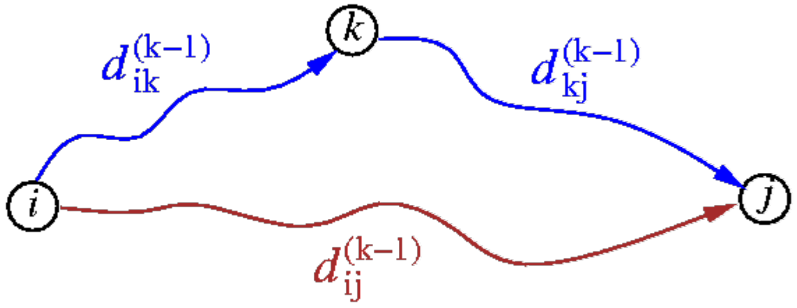
$$A[i,j] = a_{ij} = \begin{cases} 0, & \text{if } i = j \\ w_{ij}, & \text{if } (i,j) \in E \\ \infty, & \text{if } i \neq j \text{ and } (i,j) \notin E \end{cases}$$

- **Goal:** Compute $|V| \times |V|$ matrix *D* where $D[i, j] = d_{ij}$ is the weight of a shortest *i*-to-*j* path.
- Allows negative-weight edges.
- Runs in $O(V^3)$ time.

Shortest-Path Structure

- **An intermediate vertex** of a simple path $<v_1, v_2, \ldots, v_l>$ is any vertex in $\{v_2, v_3, \ldots, v_{l-1}\}$.
- Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex *i* to vertex *j* with all intermediate vertices in {1, 2, …, *k*}.
  - The path does not use vertex *k*: $d_{ij}^{(k)} = d_{ij}^{(k-1)}$
  - The path uses vertex *k*: $d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$
- **Def:** $D^k[i, j] = d_{ij}^{(k)}$ is the weight of a shortest *i*-to-*j* path with intermediate vertices in {1, 2, …, *k*}.

$$d_{ij}^{(k)} = \begin{cases} w_{ij}, & \text{if } k = 0, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right), & \text{if } k \geq 1. \end{cases}$$


$d_{\text{ik}}^{(\text{k}-1)}$
k
$d_{\text{kj}}^{(\text{k}-1)}$
i
j
$d_{\text{ij}}^{(\text{k}-1)}$
all intermediate vertices in {1, 2, ..., k−1}

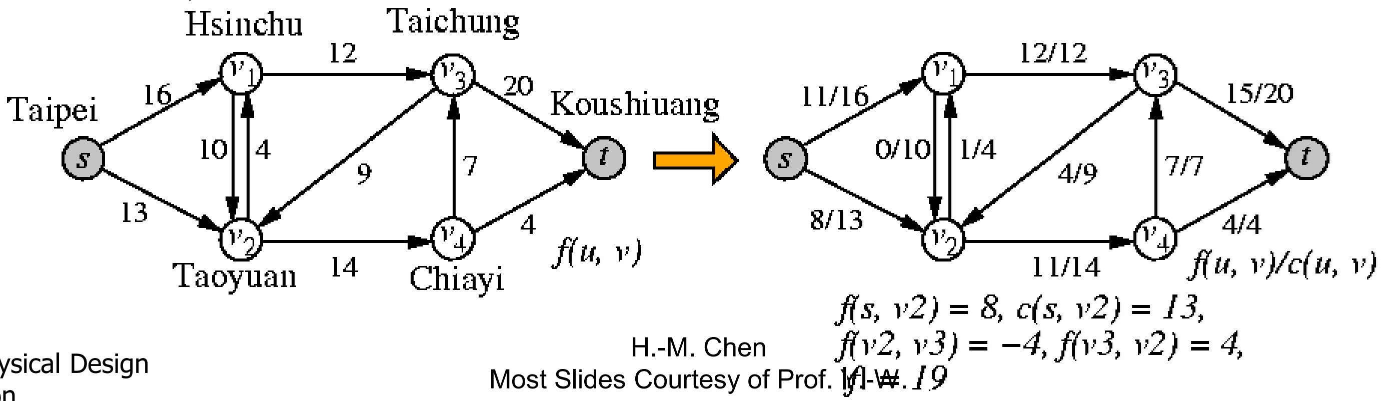
The Floyd-Warshall Algorithm for APSP

```
Floyd-Warshall($W$)
1. $n \leftarrow rows[W]$  /* $W = A$ */
2. $D^{(0)} \leftarrow W$
3. **for** $k \leftarrow 1$ **to** $n$ **do**
4.     **for** $i \leftarrow 1$ **to** $n$ **do**
5.         **for** $j \leftarrow 1$ **to** $n$ **do**
6.             $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
7. **return** $D^{(n)}$
```

- $D^{(k)}[i, j] = d_{ij}^{(k)}$: weight of a shortest i-to-j path with intermediate vertices in $\{1, 2, \ldots, k\}$.
  - $D^{(0)} = A$: original adjacency matrix (paths are single edges).
  - $D^{(n)} = (d_{ij}^{(n)})$: the final answer ($d_{ij}^{(n)} = \delta(i, j)$).
- Time complexity: $O(V^3)$.
- **Question: How to detect negative-weight cycles?**

Maximum Flow

- **Flow network:** directed $G=(V, E)$
  - **capacity** $c(u, v)$: $c(u, v) > 0, \forall (u, v) \in E$; $c(u, v) = 0, \forall (u, v) \notin E$.
  - Exactly one node with no incoming (outgoing) edges, called the **source** s (**sink** t).
- **Flow** f: $V \times V \rightarrow \mathbf{R}$ that satisfies
  - **Capacity constraint:** $f(u, v) \leq c(u, v), \forall u, v \in V$.
  - **Skew symmetry:** $f(u, v) = -f(v, u)$.
  - **Flow conservation:** $\sum_{v \in V} f(u, v) = 0, \forall u \in V - \{s, t\}$.
- **Value** of a flow f: $|f| = \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t)$, where $f(u, v)$ is the net flow from u to v.
- **The maximum flow problem:** Given a flow network G with source s and sink t, find a flow of maximum value from s to t.



Basic Ford-Fulkerson Method

```
Ford-Fulkerson-Method(G, s, t)
1. Initialize flow f to 0
2. while there exists an augmenting path p do
3.     Augment flow f along p
4. return f
```

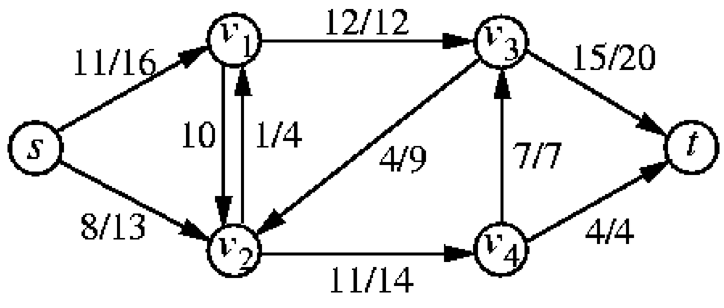
- Ford & Fulkerson, 1956
- **Augmenting path:** A path from *s* to *t* along which we can push more flow.
- Need to construct a **residual network** to find an augmenting path.

Residual Network

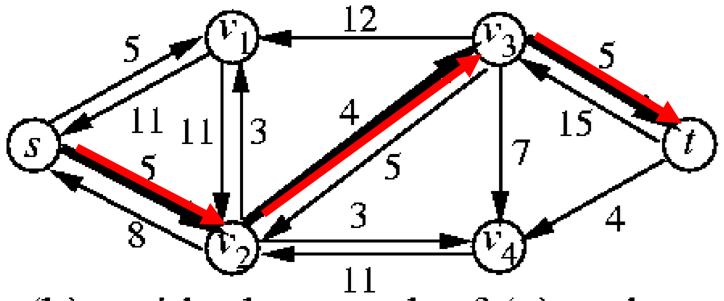
- Construct a **residual network** to find an augmenting path.
- **Residual capacity of edge (u, v), $c_f(u, v)$:** Amount of **additional** net flow that can be pushed from u to v before exceeding $c(u, v)$,

  $c_f(u, v) = c(u, v) - f(u, v)$.
- $G_f = (V, E_f)$: **residual network** of $G = (V, E)$ induced by f, where

  $E_f = \{(u, v) \in V \times V: c_f(u, v) > 0\}$.
- The residual network contains **residual edges** that can admit a positive net flow ($|E_f| \leq 2|E|$).
- Let f and f' be flows in G and G_f, respectively. The **flow sum** $f + f'$: $V \times V \rightarrow$ R : $(f + f')(u, v) = f(u, v) + f'(u, v)$ is a flow in G with value $|f + f'| = |f| + |f'|$.



(a)



(b) residual network of (a) and an augmenting path <s, v2, v3, t>

Augmenting Path

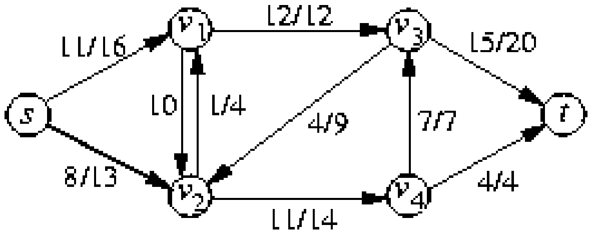
- An **augmenting path** p is a simple path from s to t in the residual network G_f.
  - $(u, v) \in E$ on p in the **forward** direction (a **forward edge**), $f(u, v) < c(u, v)$.
  - $(u, v) \in E$ on p in the **reverse** direction (a **backward edge**), $f(u, v) > 0$.
- **Residual capacity** of p, $c_f(p)$: Maximum amount of net flow that can be pushed along the augmenting path p, i.e.,

$$c_f(p) = \min\{c_f(u, v) : (u, v) \text{ is in } p\}.$$

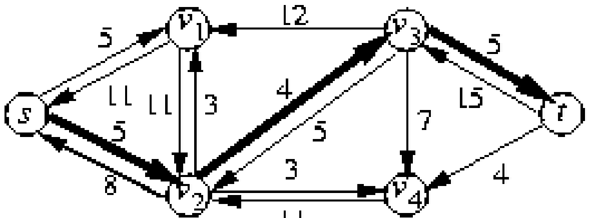
- Let p be an augmenting path in G_f. Define f_p: $V \times V \rightarrow \mathbf{R}$ by

$$f_p(u,v) = \begin{cases} c_f(p), & \text{if } (u,v) \text{ is on } p, \\ -c_f(p), & \text{if } (v,u) \text{ is on } p, \\ 0, & \text{otherwise.} \end{cases}$$

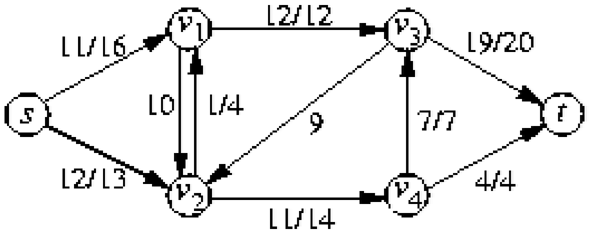
Then, f_p is a flow in G_f with value $|f_p| = c_f(p) > 0$.



(a)



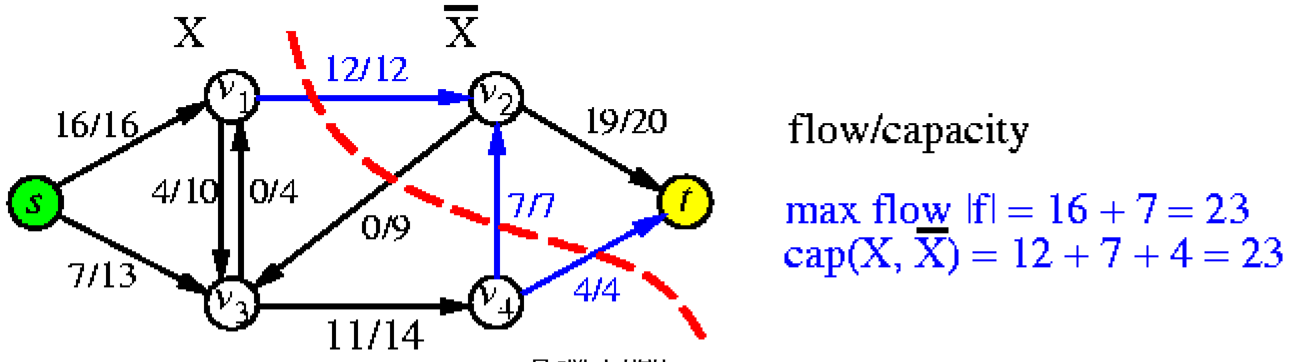
(b) residual network of (a) and an augmenting path <s, v2, v3, t>



(c) push a flow of 4-unit along the augmenting path found in (b)

Cuts of Flow Networks

- A **cut** (S, T) of flow network $G=(V, E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$.
  - **Capacity of a cut:** $c(S, T) = \sum_{u \in S, v \in T} c(u, v)$. (Count only **forward** edges!)
  - $f(S, T) = |f| \leq c(S, T)$, where $f(S, T)$ is net flow across the cut (S, T).
- **Max-flow min-cut theorem:** The following conditions are equivalent
  1. f is a max-flow.
  2. G_f contains no augmenting path.
  3. $|f| = c(S, T)$ for some cut (S, T).

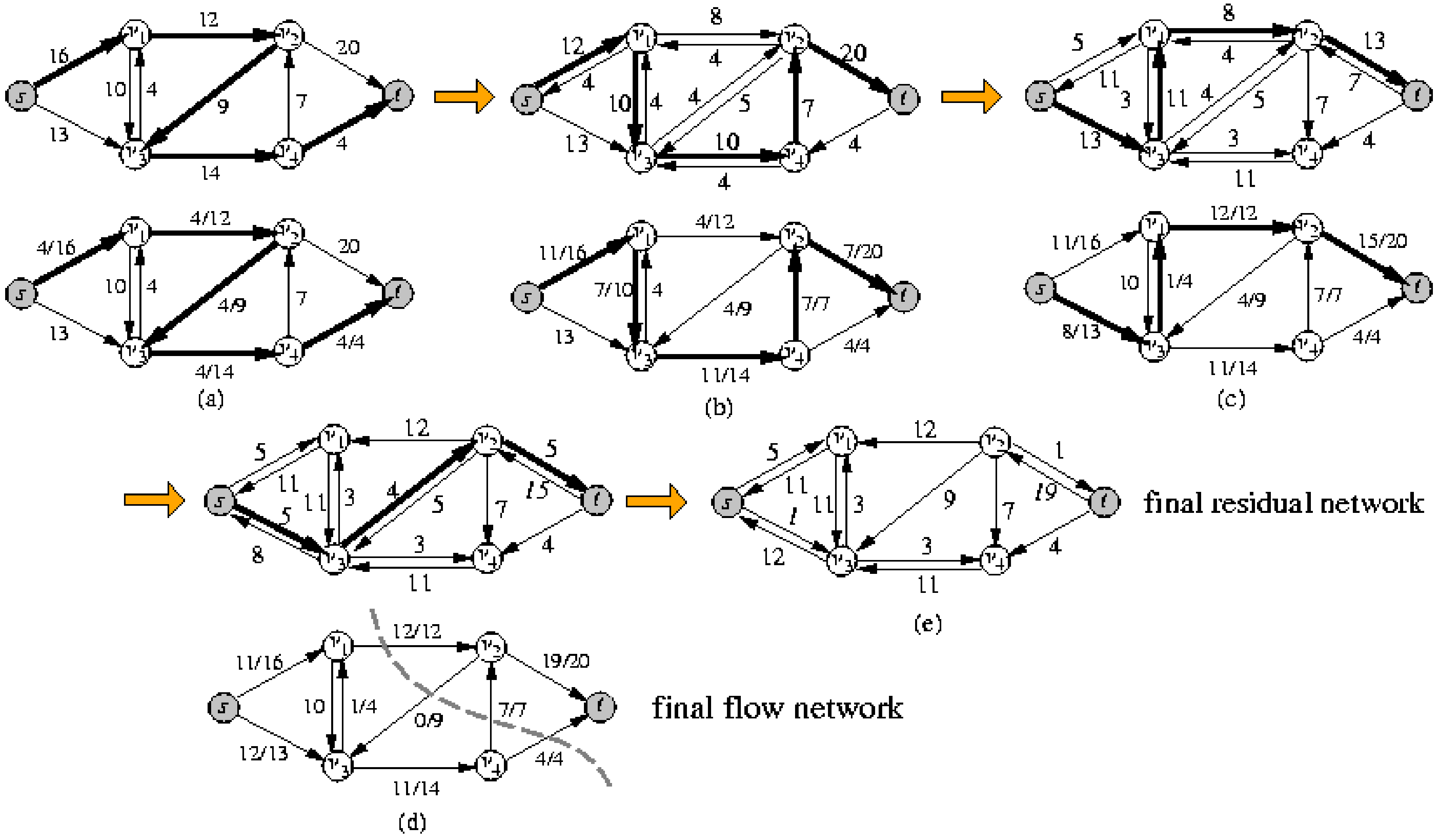


Ford-Fulkerson Algorithm

```
Ford-Fulkerson($G$, $s$, $t$)
1. for each edge $(u, v) \in E[G]$ do
2.     $f[u, v] \leftarrow 0$
3.     $f[v, u] \leftarrow 0$
4. while there exists a path $p$ from s to t in the residual network $G_f$ do
5.     $c_f(p) \leftarrow \min\{c_f(u, v)$: $(u, v)$ is in p}
6.     for each edge $(u, v)$ in $p$ do
7.         $f[u, v] \leftarrow f[u, v] + c_f(p)$
8.         $f[v, u] \leftarrow -f[u, v]$;
```

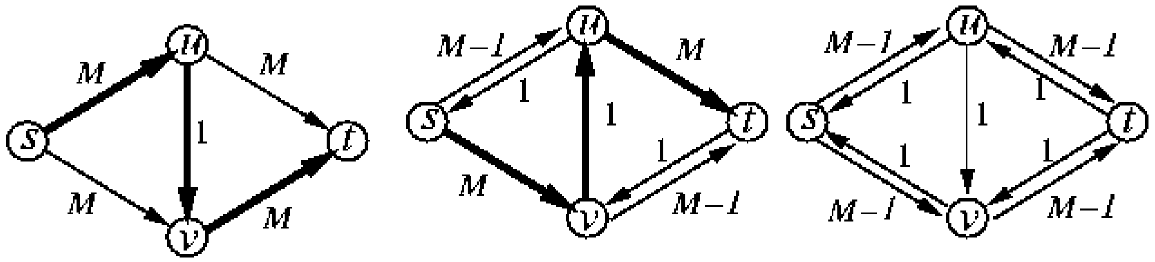
- Time complexity (assume **integral capacities**): $O(E\,|f^*|)$, where f^* is the maximum flow.
  - Each run augments at least flow value 1 □ at most $|f^*|$ runs.
  - Each run takes $O(E)$ time (using BFS or DFS).
  - **Polynomial-time algorithm?**

Example: Ford-Fulkerson Algorithm



Edmonds-Karp Algorithm

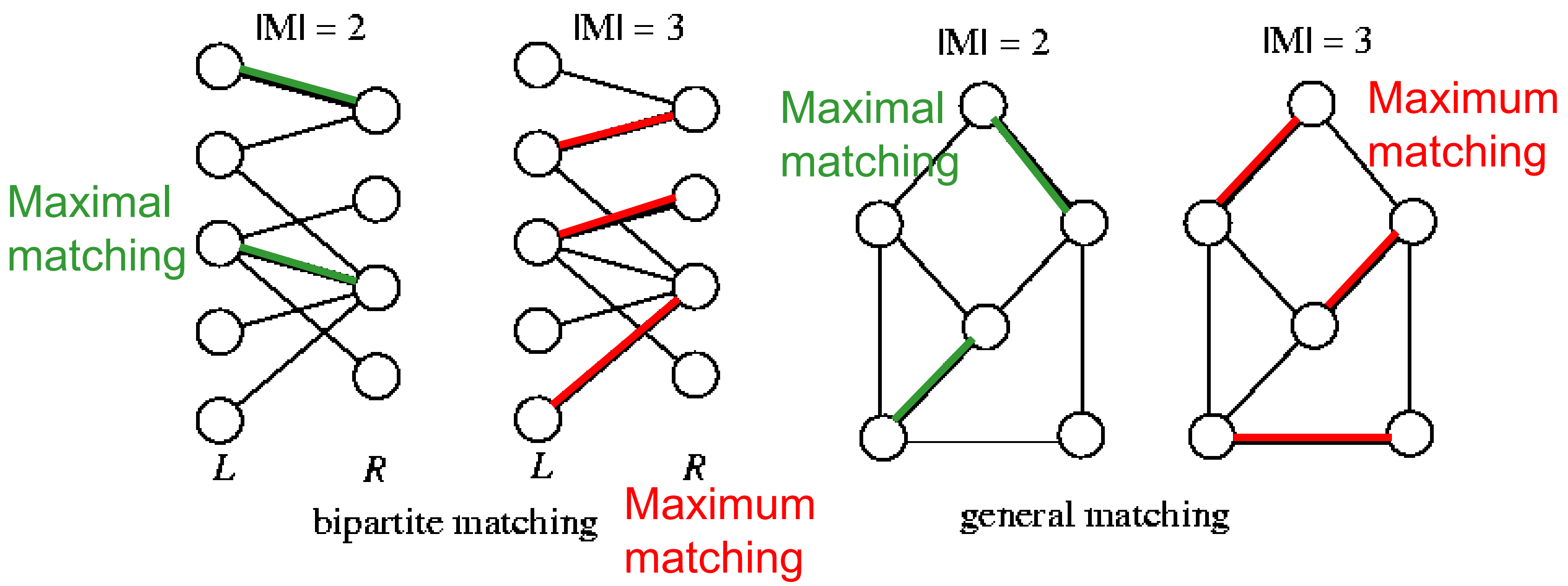
- The Ford-Fulkerson algorithm may be bad when $|f^*|$ is very large.
  - $2M$ iterations in the worst case v.s. 2 iterations.



- Edmonds-Karp Algorithm (1969): Find an augmenting path by shortest path (minimum # of edges on the path), i.e., use breadth-first search (BFS).
  - Using the Edmonds-Karp algorithm, the shortest-path distance $\delta_f(s, v)$ in the residual network G_f increases monotonically with each flow augmentation.
  - The # of flow augmentations performed by the Edmonds-Karp algorithm is at most $O(VE)$.
  - Each flow augmentation can be found in $O(E)$ time by BFS.
  - Total running time of the Edmonds-Karp algorithm is $O(VE^2)$.
- Goldberg & Tarjan (1985): $O(EV \lg(V^2/E))$; Ahuja, Orlin, Tarjan (1989): $O(EV \lg(V\sqrt{\lg U}/E + 2))$, U = max. edge capacity.

Maximum Cardinality Matching

- Given an undirected $G=(V, E)$, $M \subseteq E$ is a **matching** iff at most one edge of M is incident on v, $\forall\, v \in V$.
  - $v \in V$ is **matched** if some edge in M is incident on v; otherwise, v is **unmatched**.
- M is a **maximum matching** iff $|M| \geq |M'|$ $\forall$ matching M'.

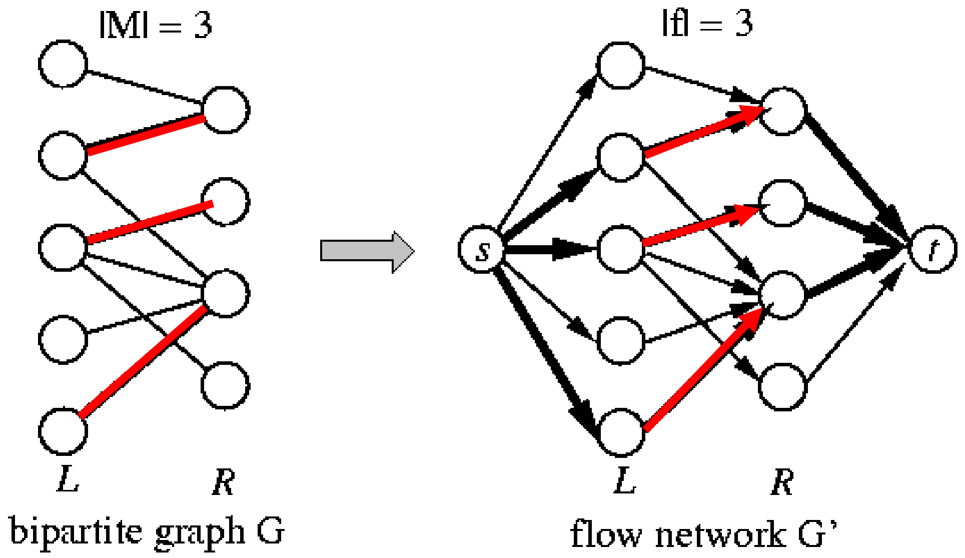


Application: Maximum Cardinality Bipartite Matching

- Given a bipartite graph $G = (V, E)$, $V = L \cup R$, construct a unit-capacity flow network $G' = (V', E')$:

  $V' = V \cup \{s, t\}$

  $E' = \{(s, u): u \in L\} \cup \{(u, v): u \in L, v \in R, (u, v) \in E\} \cup \{(v, t): v \in R\}$.
- The cardinality of a maximum matching in G = the value of a maximum flow in G' (i.e., $|M| = |f|$).
- Time complexity: $O(VE)$ by the Ford-Fulkerson algorithm.
  - Value of maximum flow $\leq \min(L, R) = O(V)$.

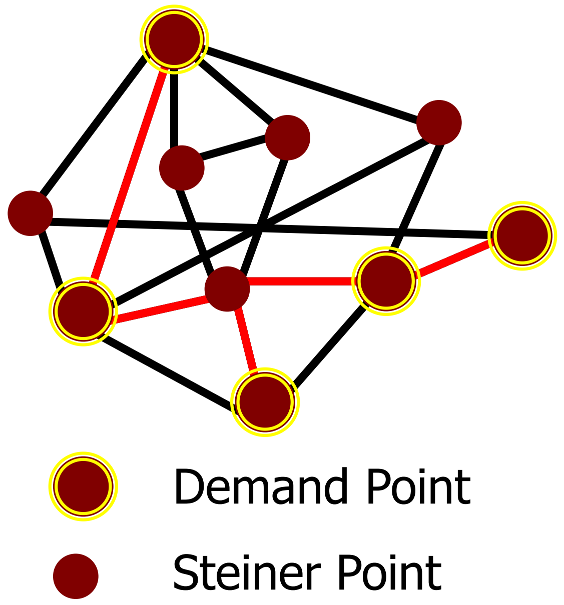


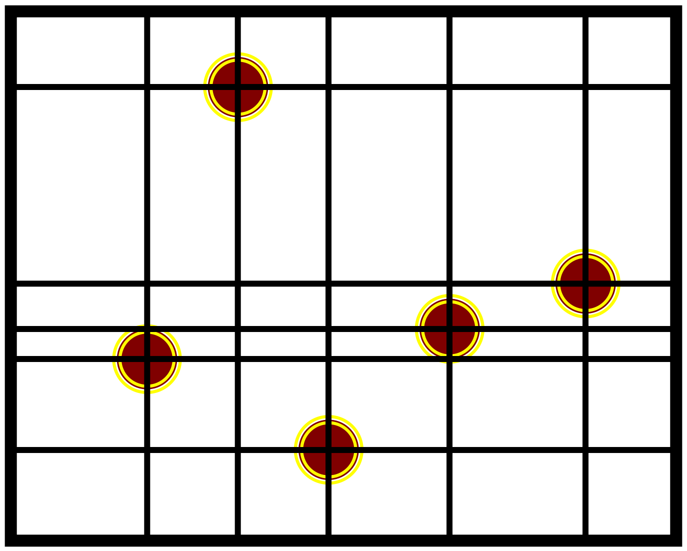
Steiner Tree Algorithms (1/4)

- Steiner Minimum Tree (SMT) – Given $G = (V,E)$ and $D \subseteq V$, select $V' \subseteq V \rightarrow D \subseteq V'$, and V' induces a tree of minimum cost over all such trees
  - D – Demand Points, (V' – D) – Steiner Points
  - Demands point – the net terminal
  - Steiner point – the connection point of two paths
- $D = V \rightarrow$ SMT $\equiv$ MST (minimum spanning tree)
- $|D| = 2 \rightarrow$ SMT $\equiv$ SSSP (single source shortest path)

Steiner Tree Algorithms (2/4)

- The Underlying Grid Graph – defined by the intersections of H-lines and V-lines extending from the demand points

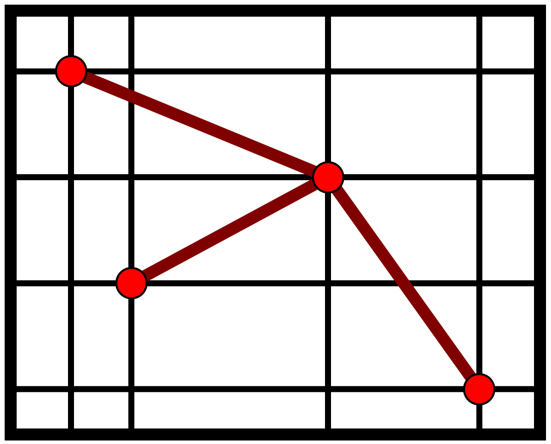


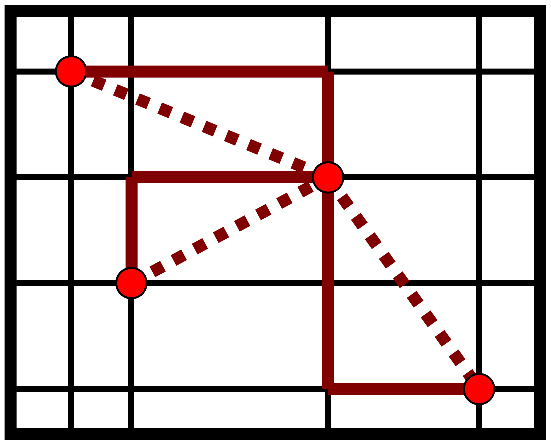


Steiner Tree Algorithms (3/4)

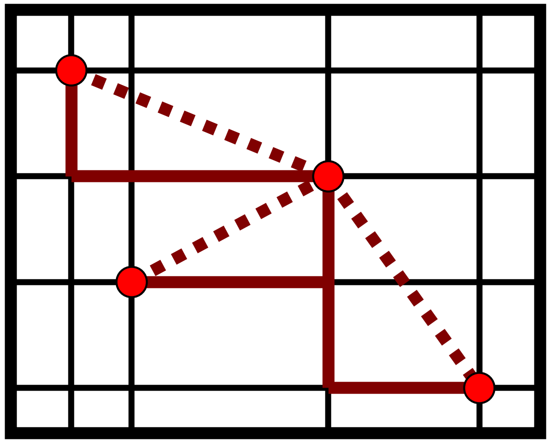
- Rectilinear Steiner Tree (RST) – a steiner tree whose edges are restricted to rectilinear shape
- Rectilinear Steiner Minimum Tree (RSMT)
- Theorem:

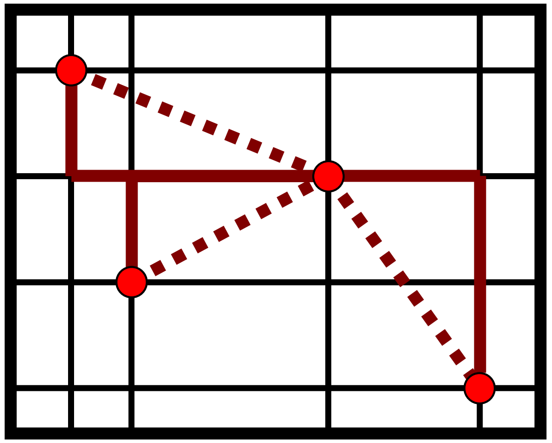
$$\frac{Cost_{MST}}{Cost_{RSMT}} \le \frac{3}{2}$$

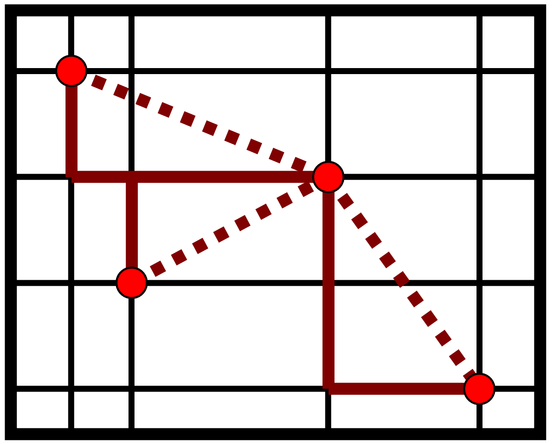




Steiner Tree Algorithms (4/4)



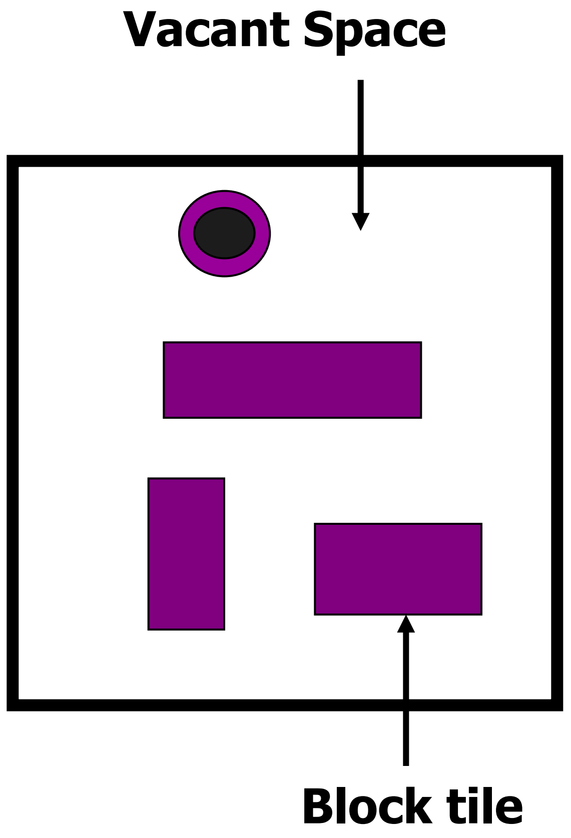




Different Steiner trees constructed from a minimum cost spanning

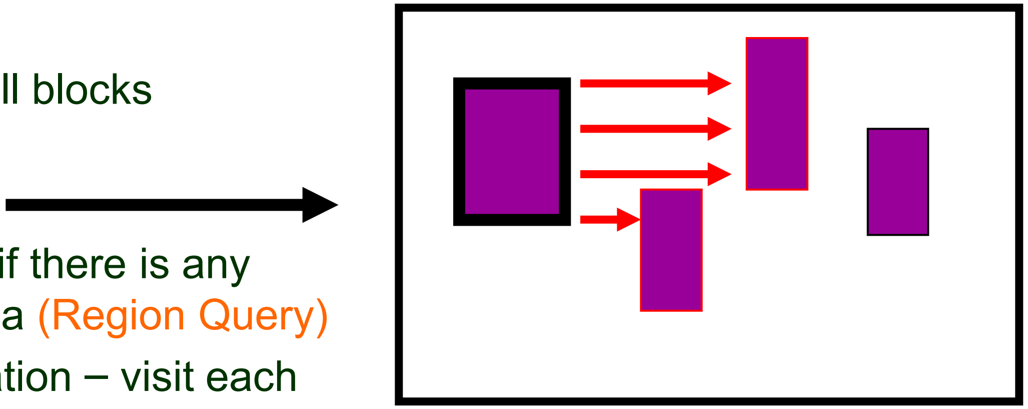
Components of a Layout System (1/2)

- Elements for layout editor
  - Operand – database entry
    - Flat – Dot, line, circle, rectangle, path, polygon, donut, etc.
    - Hierarchical – object, instance, cell, etc.



Components of a Layout System (2/2)

- Operator – atomic operation
  - Point finding – find the block containing the point (Pick)
  - Neighbor finding – find all blocks touching a given block
  - Block visibility
  - Area searching – check if there is any block residing in that area (Region Query)
  - (Directed) area enumeration – visit each tile intersecting with the area exactly once (in sorted order) (Region Query)
  - Block insertion
  - Block deletion
  - Plowing – move an object in one direction and then force associated objects to move
  - Compaction – plow the entire layout
    - One dimensional vs. two dimensional
  - Channel generation



Basic Data Structures for Layout Representation

- Linked list of blocks
- Bin-based method
- Neighbor pointers
- Corner stitching

Linked List of Blocks

- Only suitable for a hierarchical system where each level contains few blocks
- Space complexity *O(n)*, *n* is the number of blocks
- Time complexity *O(n)* for Find, Insert, and Delete

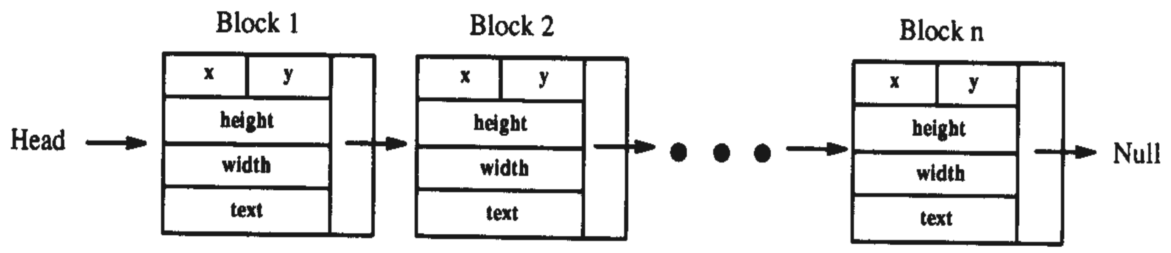


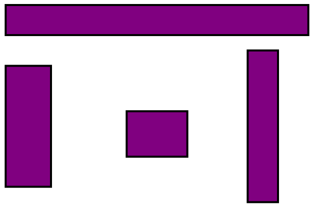
Figure 4.17: Linked list representation.

Bin-Based Method

- Layout is superimposed by virtual grids
- Space complexity $O(bn)$, b is the number of bins
- Easily degenerated to the linked list – all blocks are in a bin
- Worse performance for neighbor finding, area searching, area enumeration
  - Bins containing no blocks must be tested
  - Worst case complexity $O(b + n)$
- Sensitive to time-space tradeoff
  - If the bins are small with respect to the average size of a block, the blocks are likely to intersect with more than one bin, increasing storage requirements
  - If the bins are too large, the average case performance will be reduced since the linked lists used to store blocks in each bin will be very long

Neighbor Pointers

- Keep neighboring info
- Space complexity $O(n^2)$
- Block representation
    - Upper left corner, length, width
- Easy for plowing operation
- Difficult to generate channel (vacant space not explicitly represented)



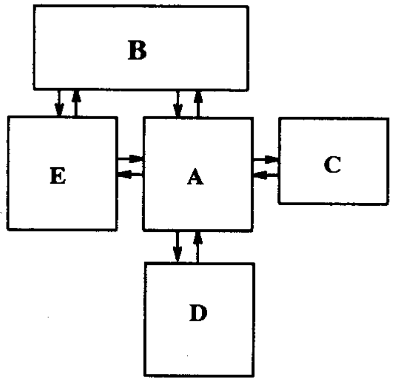


Figure 4.22: Neighbor pointers.

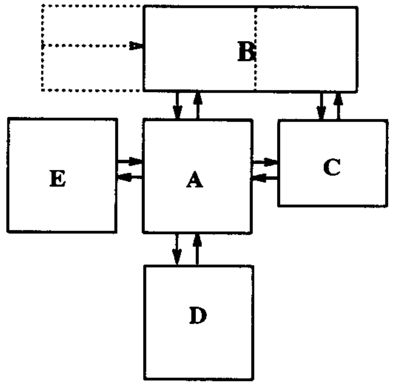
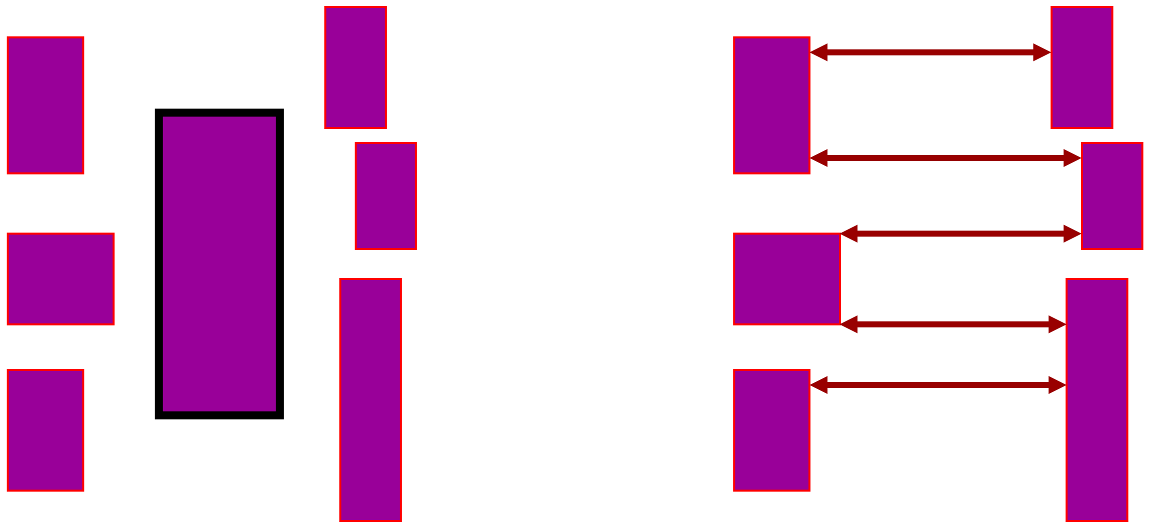


Figure 4.23: Update of neighbor pointers.

Neighbor Pointers (cont)

- Difficult for updating operation
  - $O(n^2)$ for a plowing operation which modifies the neighbors of all blocks
  - $O(n)$ for block insertion and block deletion



Corner Stitching (1/16)

- A data structure to represent both block tiles and vacant (space) tiles, used by Berkeley MAGIC layout editor
- Keep four neighboring pointers
- Canonical representation for "one layer" of stuff on an IC, such as metal1, poly, etc.
- Need more memory space

operation	av. comp. complexity	av.comp. complexity with hint
Insert	$O(\sqrt{n})$	$O(1)$
Delete	$O(\sqrt{n})$	$O(1)$
Neighbor enumeration	$O(\sqrt{n})$	$O(1)$
Area enumeration	$O(\sqrt{n} + n')$	$O(n')$
Point finding	$O(\sqrt{n})$	$O(1)$

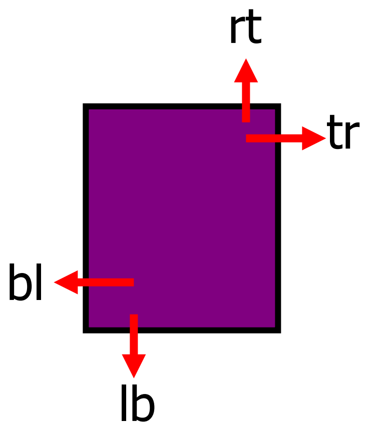
Corner Stitching (2/16)

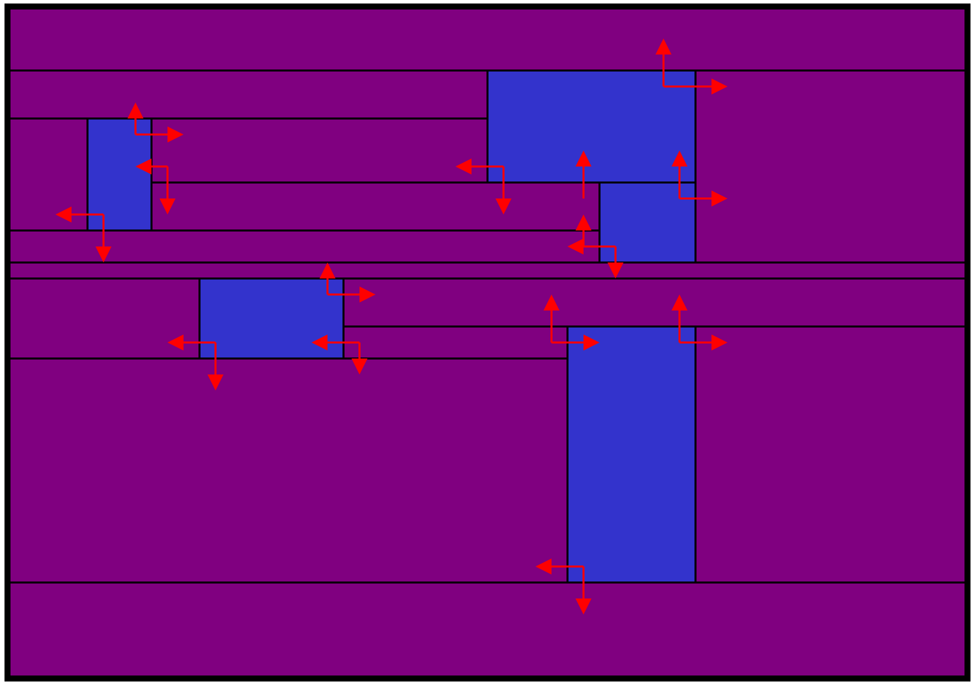
- An example

Canonical representation

Given a layout of objects, there is only one space tile representation

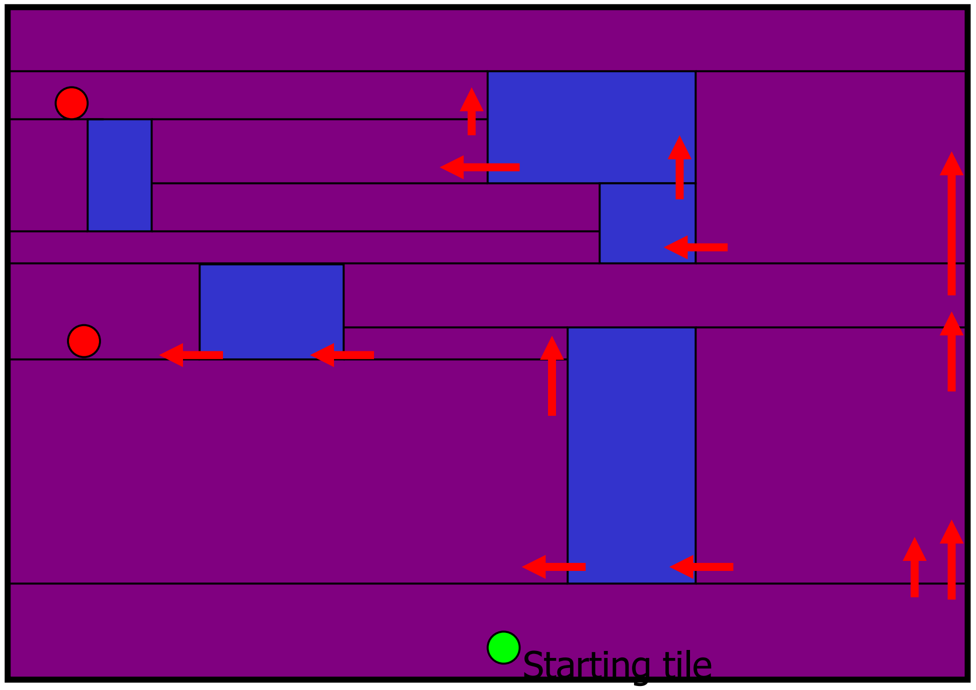
Maximal horizontal strips : every space tile must be as wide as possible





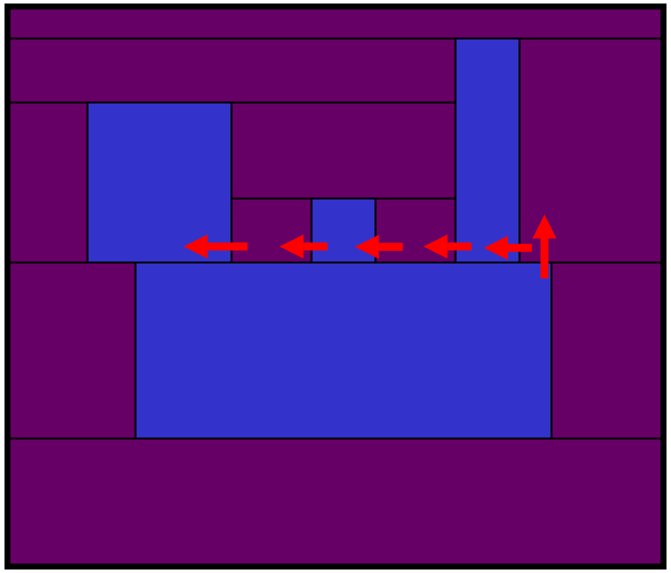
Corner Stitching (3/16)

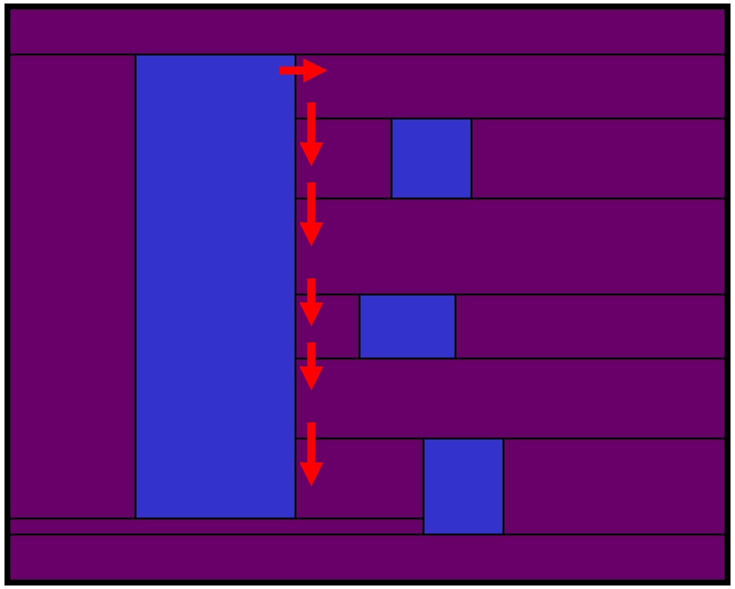
- Point finding
    - First move up/down
    - Then move left/right
    - Move up/down if the tile does not contain the target during horizontal move
    - Worse case: $O(n)$
    - Average case: $O(\sqrt{n})$



Corner Stitching (4/16)

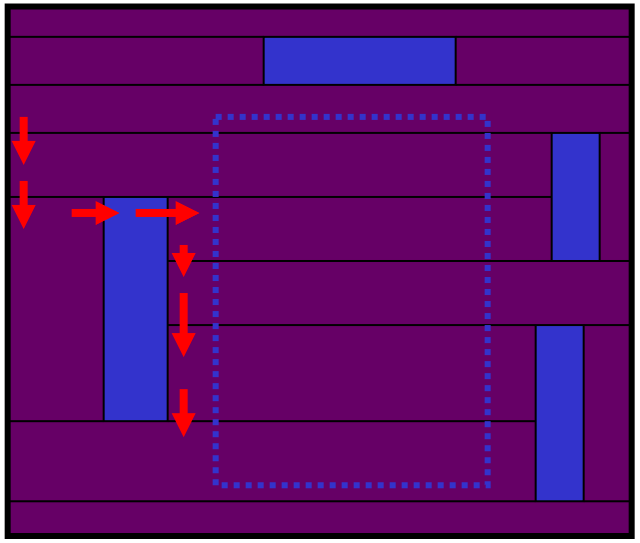
- Neighbor finding

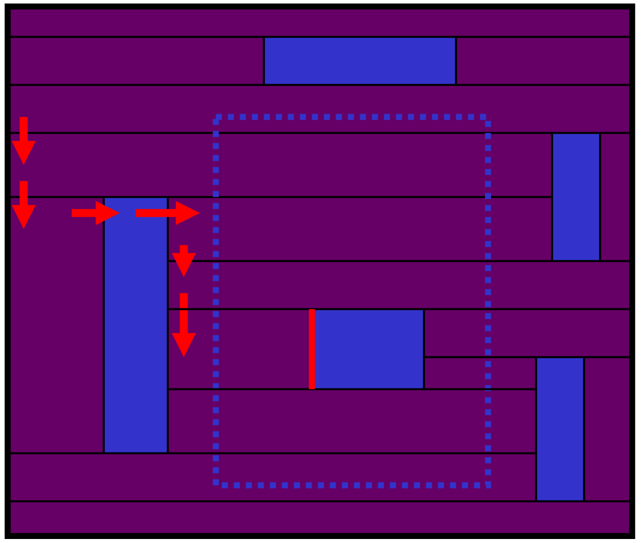




Corner Stitching (5/16)

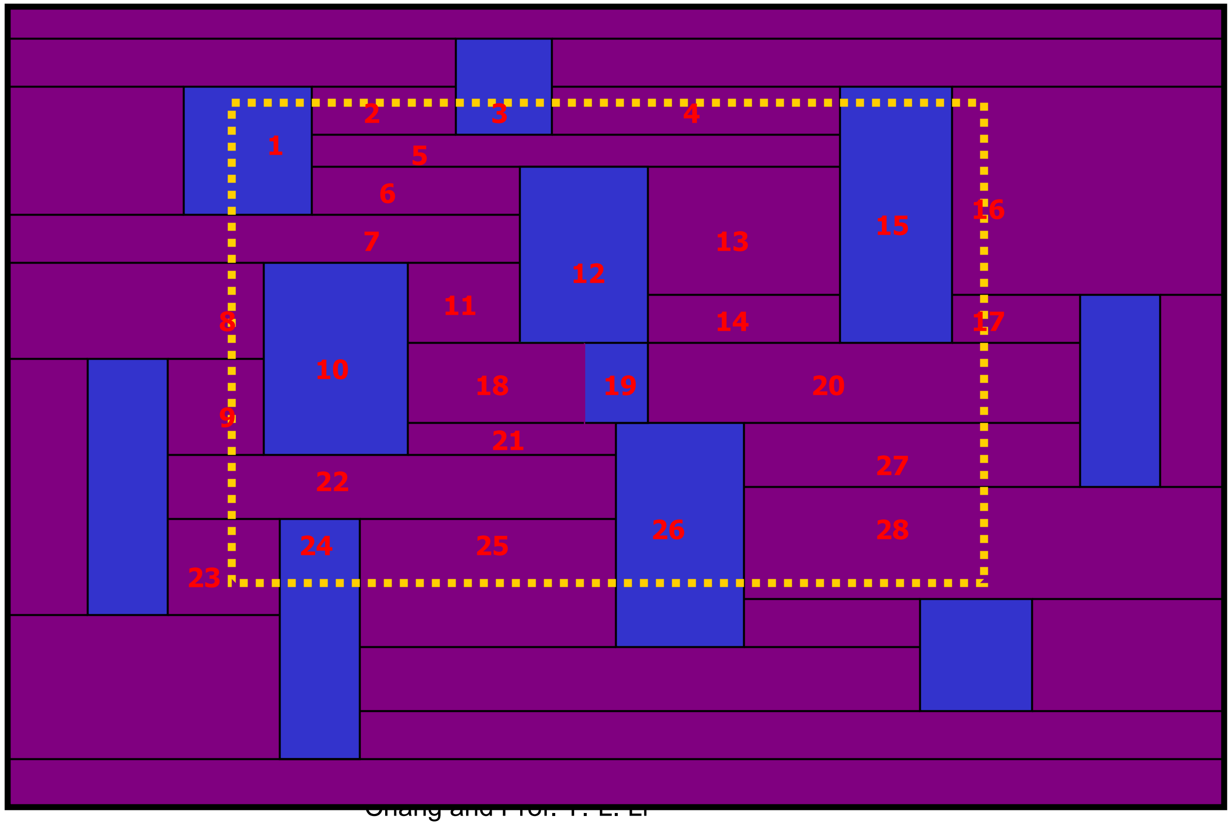
- Area Searching – search if there is block tile inside an area
  - First do point location
  - Check the right side of each tile containing the left edge of the searching area





Corner Stitching (6/16)

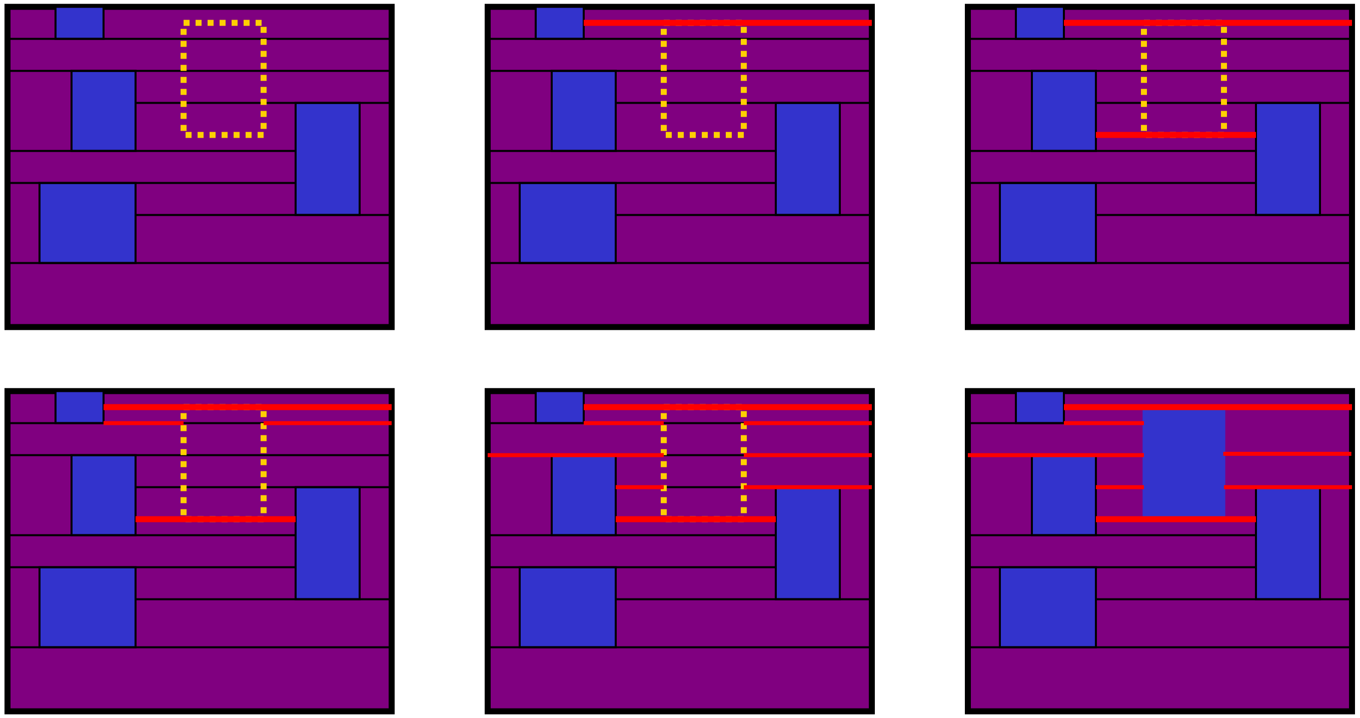
- Enumerate All Tiles – first DFS, and then BFS
  - If the bottom left corner of the neighbor touches the current tile, call recursive enumerate algorithm
  - If the bottom edge of the search area cuts both the current tile and the neighbor, call recursive enumerate algorithm



Corner Stitching (7/16)

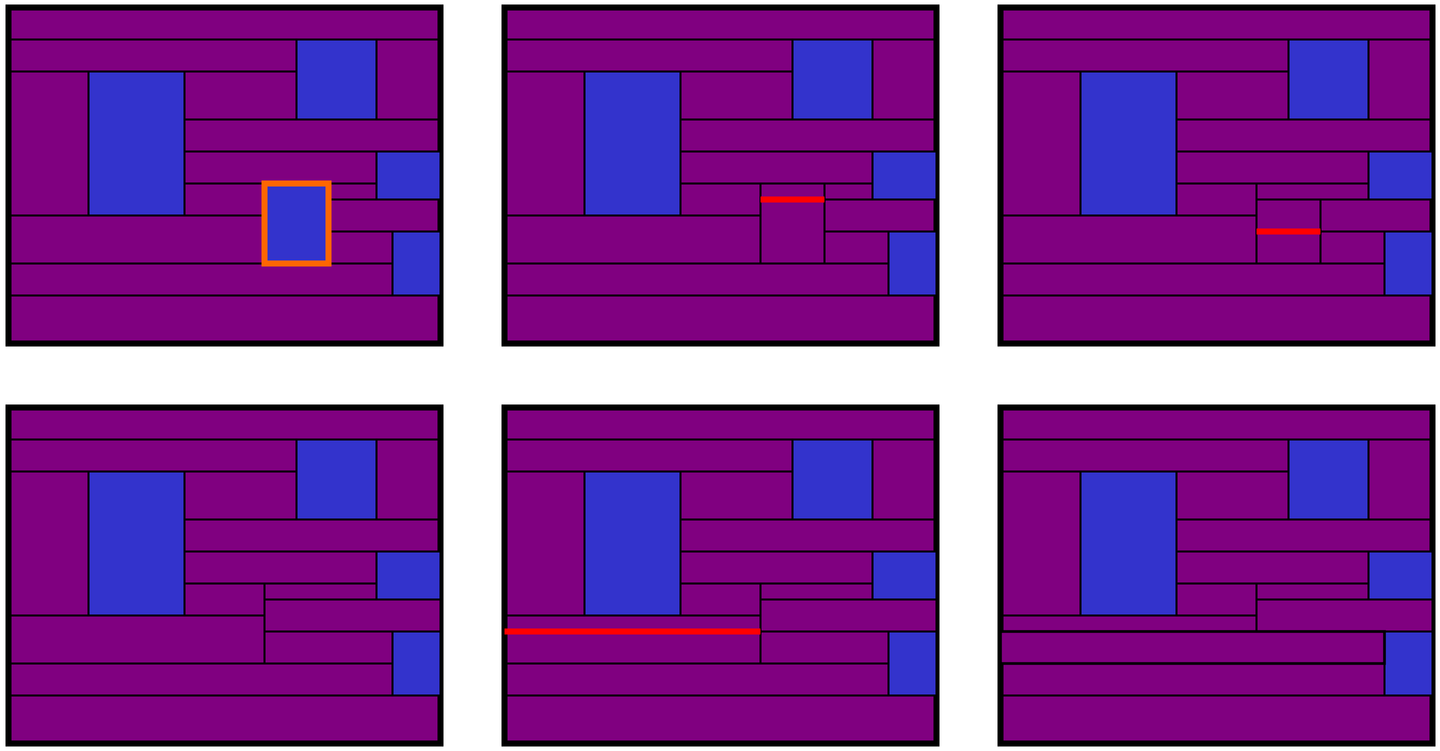
- Block Insertion

1. Split space tiles containing the top and bottom edges of the new tile
2. Walk the left and right edges, then split tiles
3. Merge space-like tiles vertically wherever possible



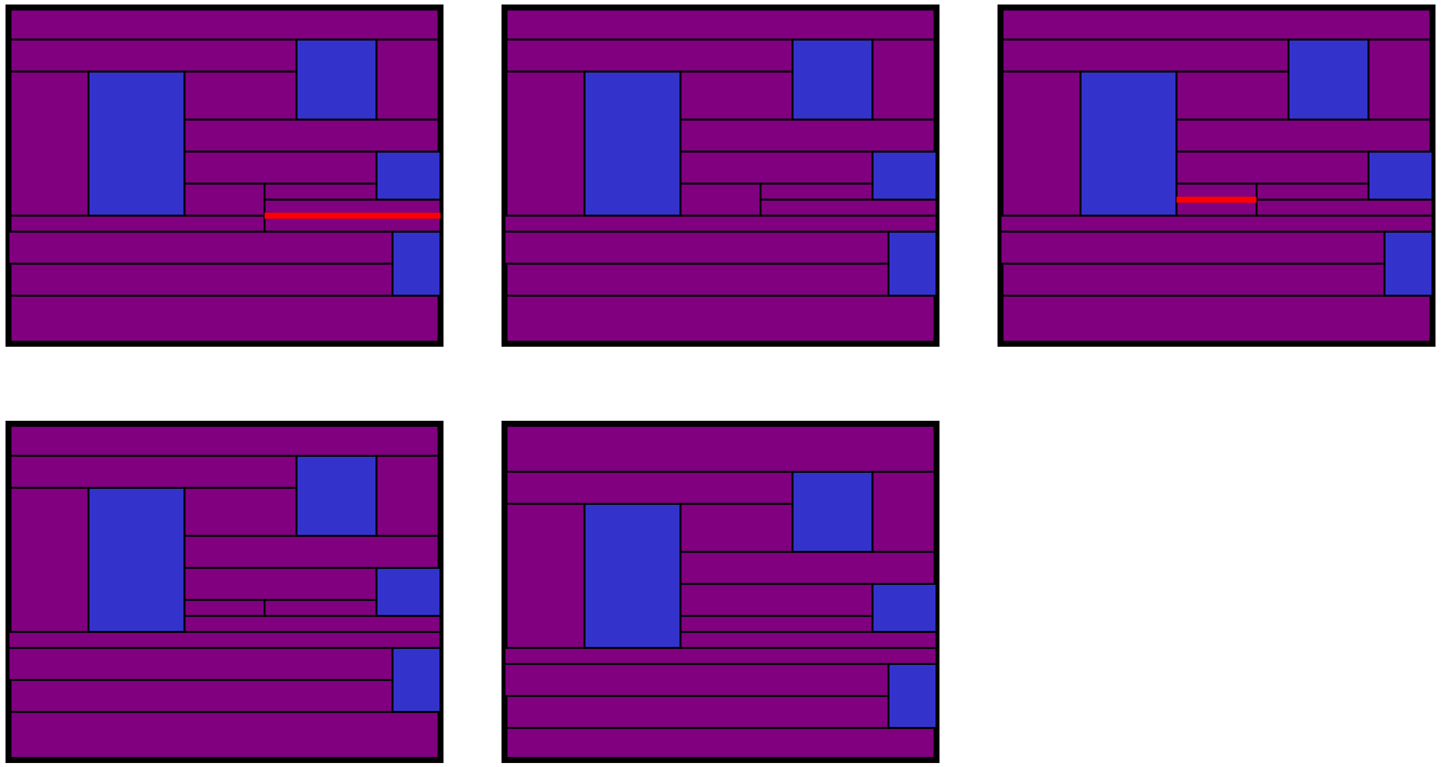
Corner Stitching (8/16)

- Block Deletion



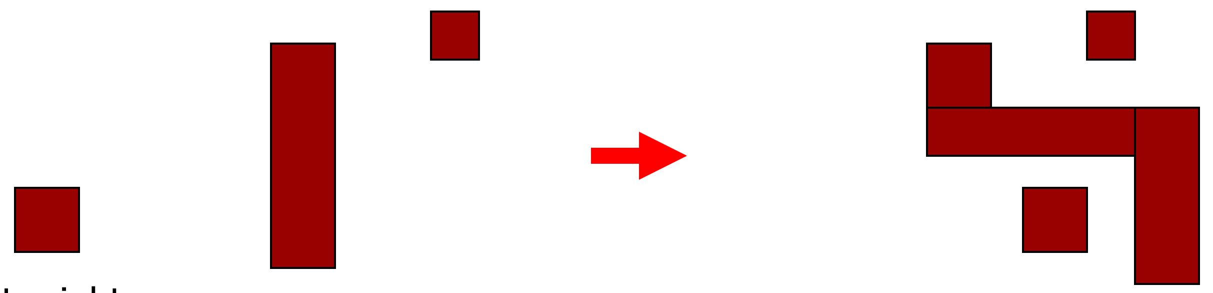
Corner Stitching (9/16)

- Block Deletion (cont)



Corner Stitching (10/16)

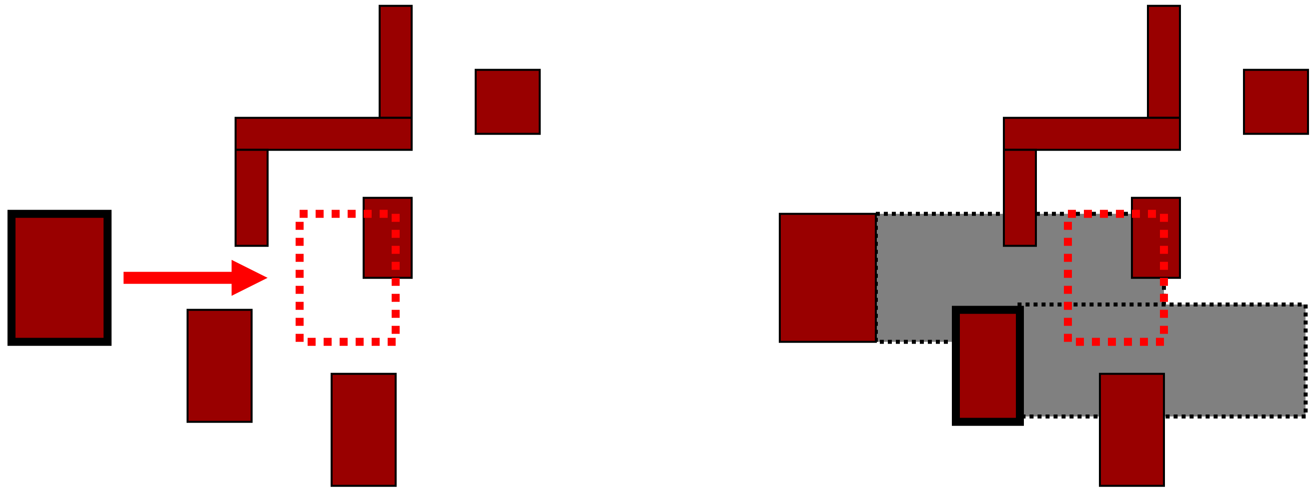
- Plow – preserve ordering between blocks(recursive area search and compress)



Plow to right

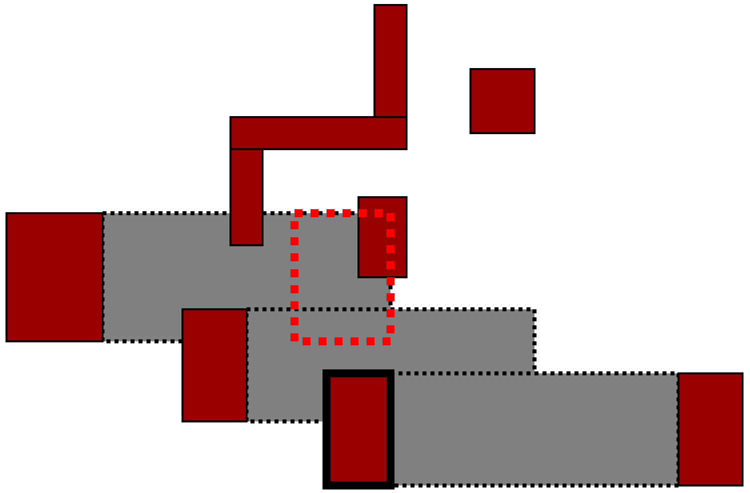
Corner Stitching (11/16)

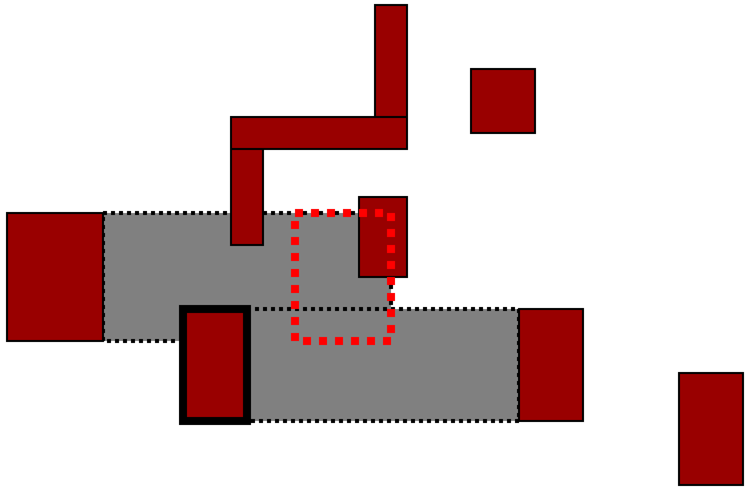
- Plowing example



Corner Stitching (12/16)

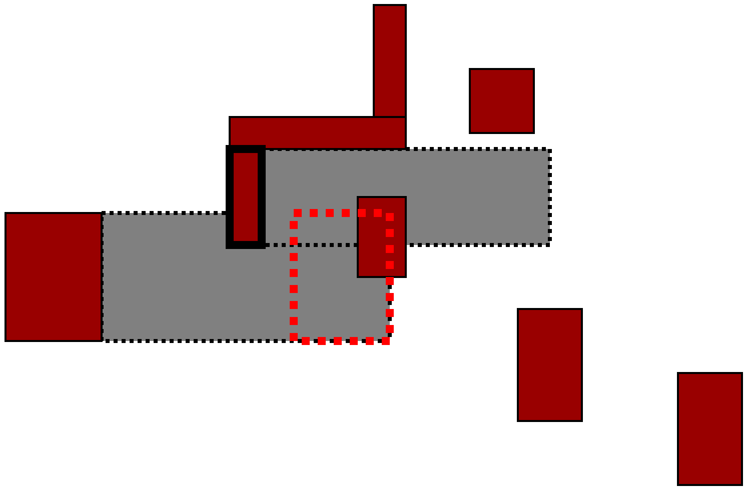
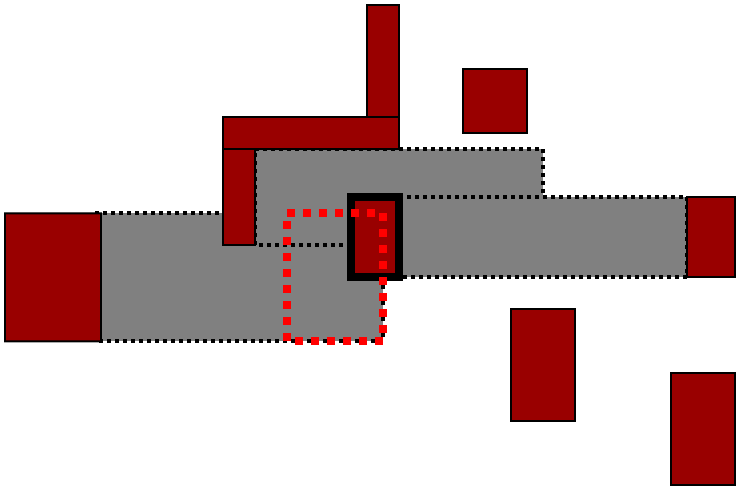
- Plow (cont)





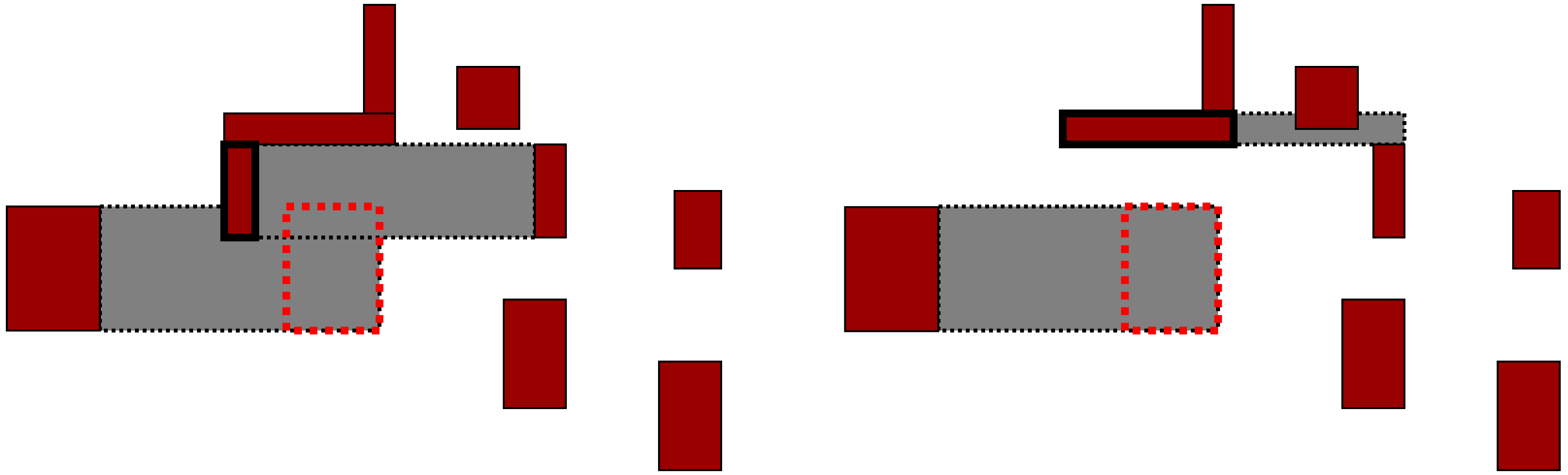
Corner Stitching (13/16)

- Plow (cont)

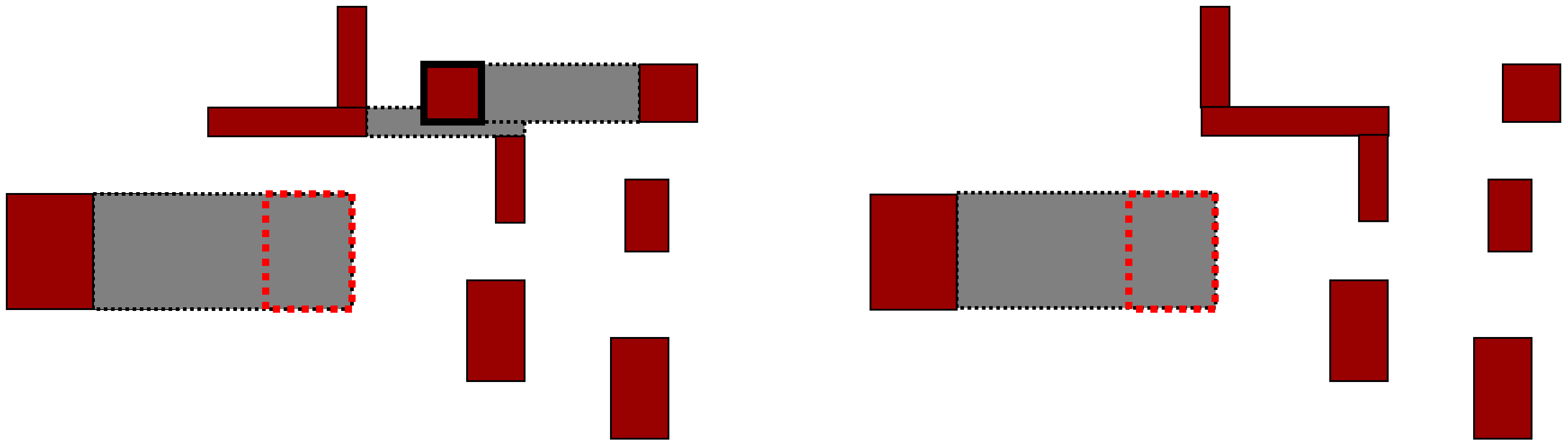
Corner Stitching (14/16)

- Plow (cont)



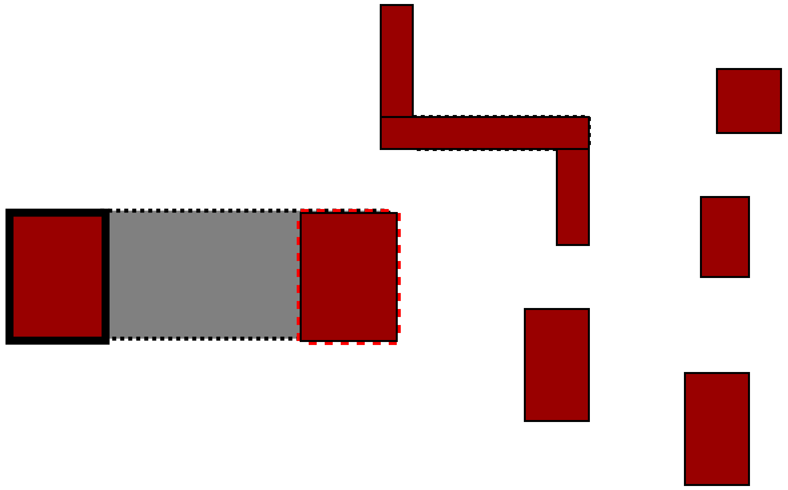
Corner Stitching (15/16)

- Plow (cont)



Corner Stitching (16/16)

- Plow (cont)



Appendix: Physical Design Related Conferences/Journals

- Important Conferences:
  - **ACM/IEEE Design Automation Conference (DAC)**
  - **IEEE/ACM Int'l Conference on Computer-Aided Design (ICCAD)**
  - ACM Int'l Symposium on Physical Design (ISPD)
  - ACM/IEEE Asia and South Pacific Design Automation Conf. (ASP-DAC)
  - ACM/IEEE Design, Automation, and Test in Europe (DATE)
  - IEEE Int'l Conference on Computer Design (ICCD)
  - IEEE Int'l Symposium on Quality Electronic Design (ISQED)
  - IEEE Int'l Symposium on Circuits and Systems (ISCAS)
  - Others: VLSI Design/CAD Symposium (Taiwan)
- Important Journals:
  - **IEEE Transactions on Computer-Aided Design (TCAD)**
  - **ACM Transactions on Design Automation of Electronic Systems (TODAES)**
  - **IEEE Transactions on VLSI Systems (TVLSI)**
  - **IEEE Transactions on Computers (TC)**
  - IEE Proceedings
  - IEICE
  - INTEGRATION: The VLSI Journal