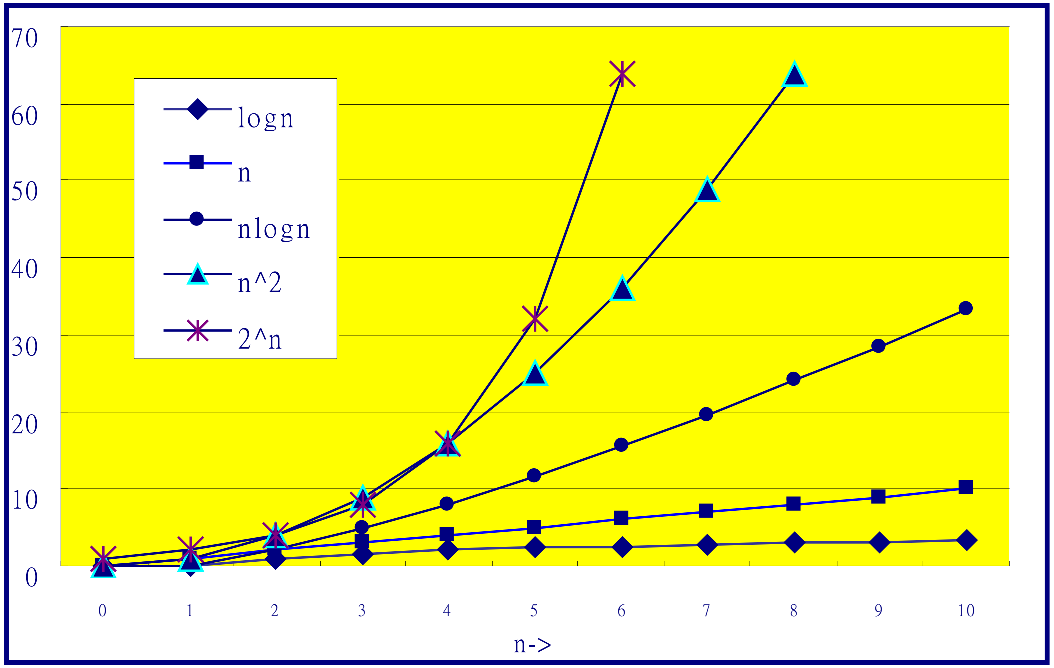
# Reviewing Algorithms for IntroEDA

- Course contents:
  - Computational complexity reviews
  - Often-used graph algorithms and terms



# Asymptotic Notation -Big "oh"

- $f(n)=O(g(n))$ iff
  - $\exists$ positive const. c and $n_0$, $\ni f(n) \le cg(n)\ \forall\ n,\ n \ge n_0$
  - e.g.
    - **$3n+2 = O(n)$**

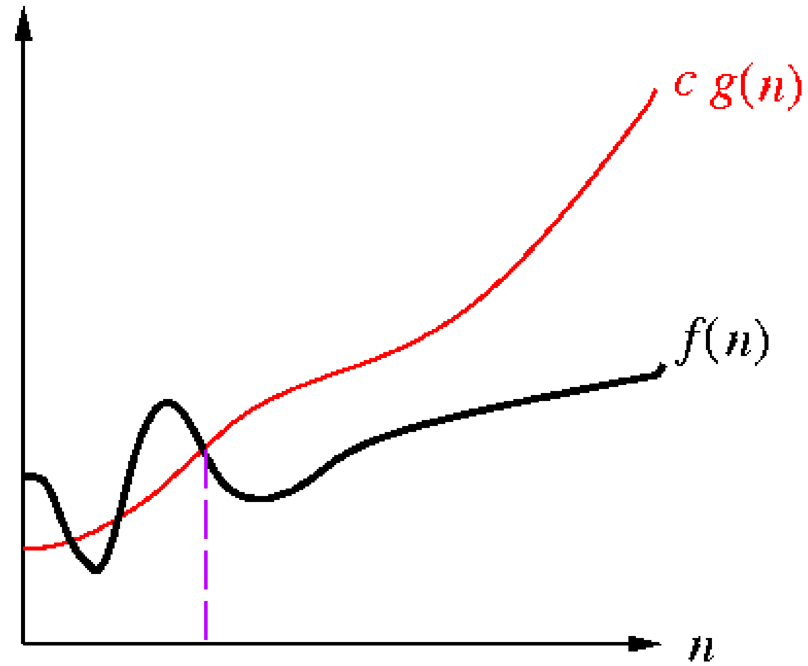
      **$3n+2 \le 4n$ for all $n \ge 2$**
    - **$10n^2+4n+2=O(n^2)$**

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

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

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

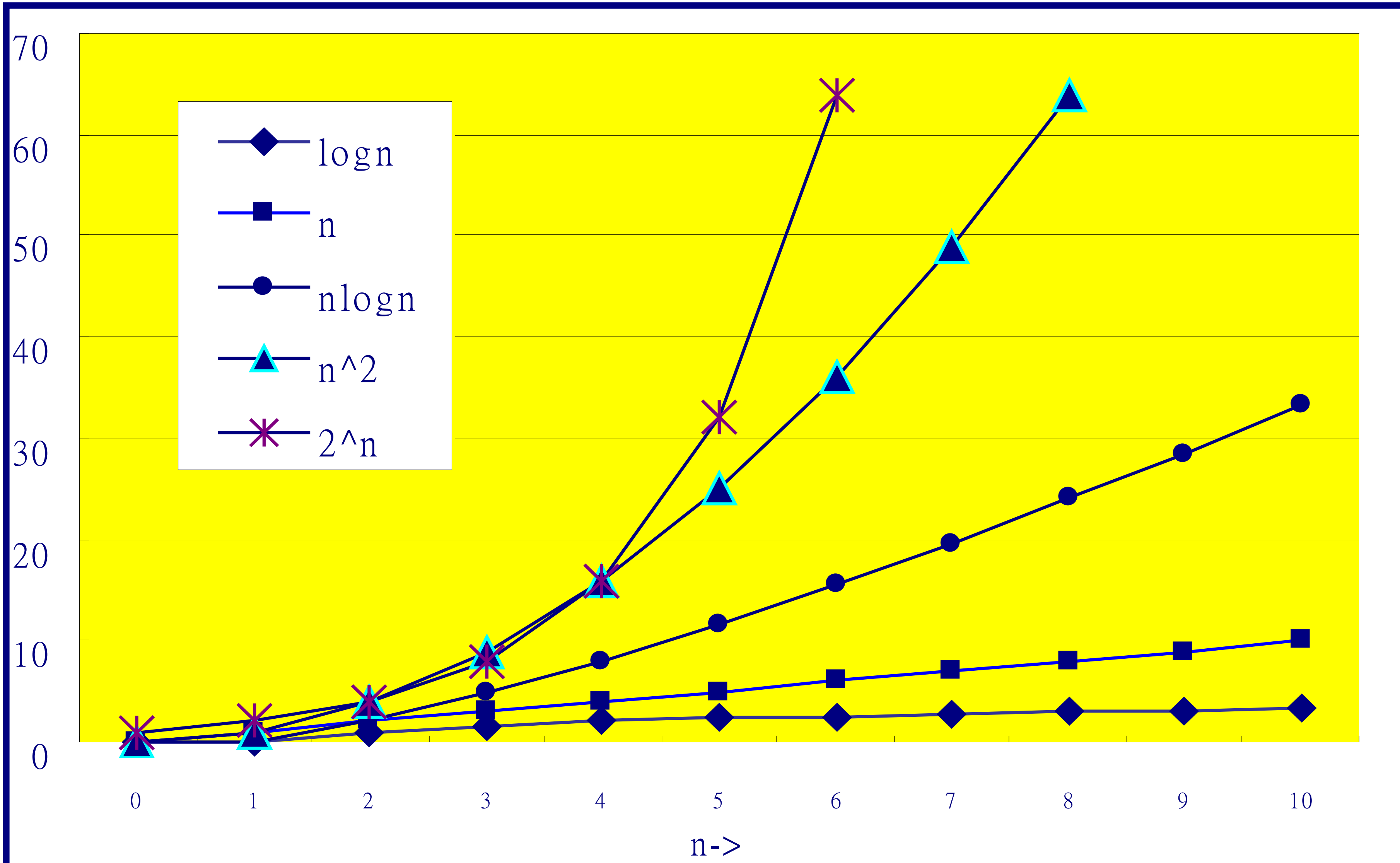
* g(n) should be a ***least upper bound***



# Computational Complexity

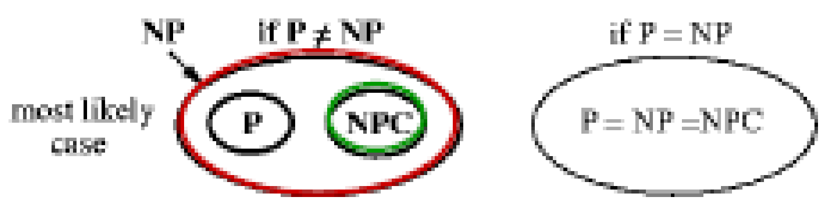
- Computational complexity: an abstract measure of the **time** and **space** necessary to execute an algorithm as function of its “input size”.
- Input size examples:
  - sort $n$ words of bounded length $\Rightarrow n$
  - **the input is the integer $n \Rightarrow \lg n$**
  - the input is the graph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

# Output Growing Curves



# 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,

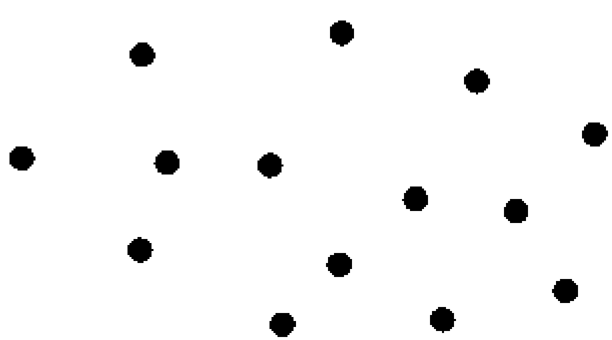
because no such algorithm is possible!"

# 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 $\in$ 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 $\leq B$.
- TSP $\in$ P?
  - Need to solve (find a tour) in polynomial time.
  - Still unknown if TSP $\in$ P.

# Decision Problems and NP-Completeness

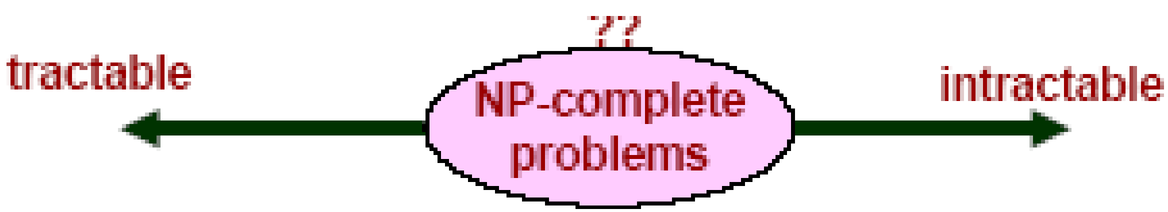
- **Decision problems:** those having yes/no answers.
  - TSP: Given a set of cities, distance between each pair of cities, and a bound *B*, **is there a route** that starts and ends at a given city, visits every city exactly once, and has total distance at most *B*?
- **Optimization problems:** those 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.

# 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.* $L' \leq_P L$ 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).



# Coping with NP-hard Problems

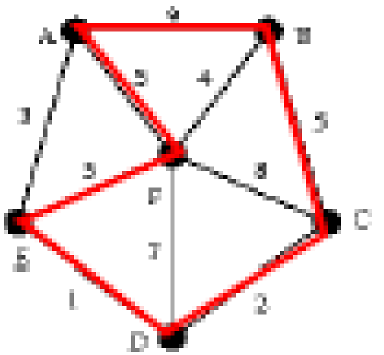
- **Exhaustive search/Branch and bound**
  - Is feasible only when the problem size is small.
- **Approximation algorithms**
  - Guarantee to be a fixed percentage away from the optimum.
  - E.g., MST for the minimum Steiner tree problem.
- **Pseudo-polynomial time algorithms**
  - Has the form of a 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

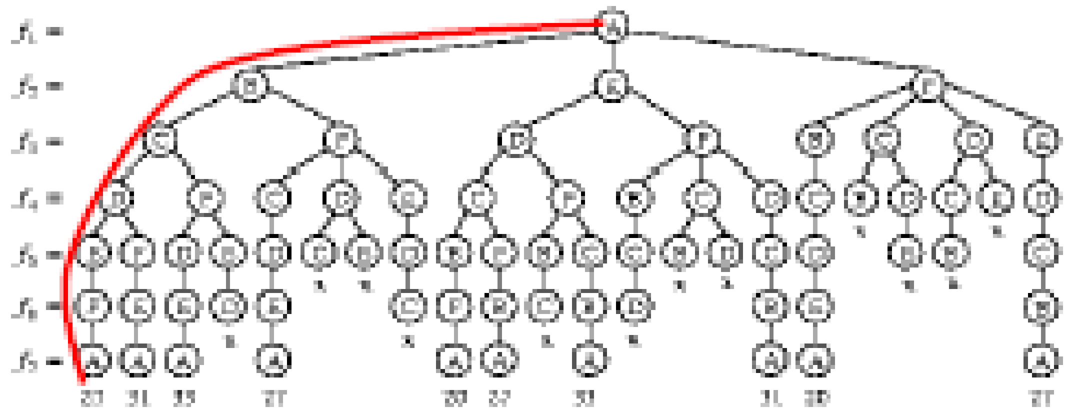
- **Exhaustive search:** Search the entire solution space.
- **Branch and bound:** A search technique with pruning.
- **Greedy method:** Pick a locally optimal solution at each step.
- **Dynamic programming:** Partition a problem into a collection of sub-problems, the sub-problems are solved, and then 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

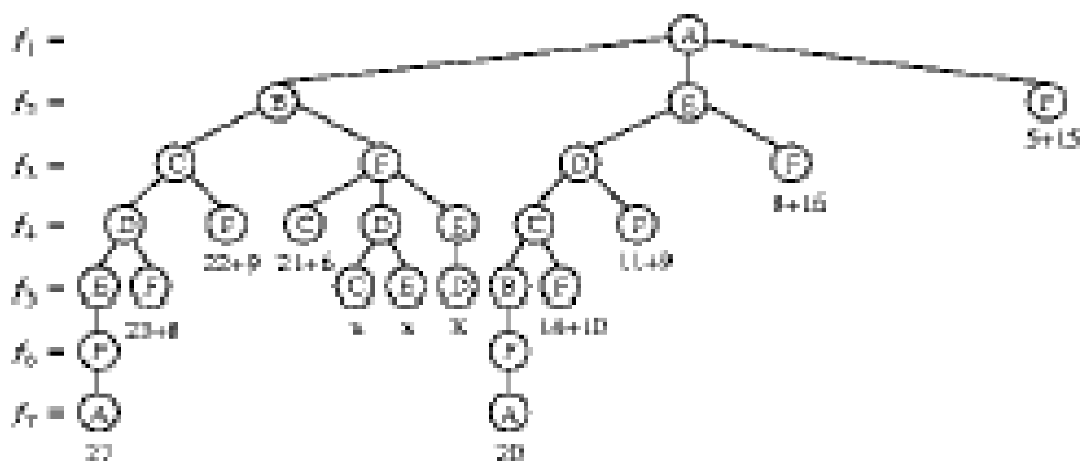
- TSP example




State-space trees



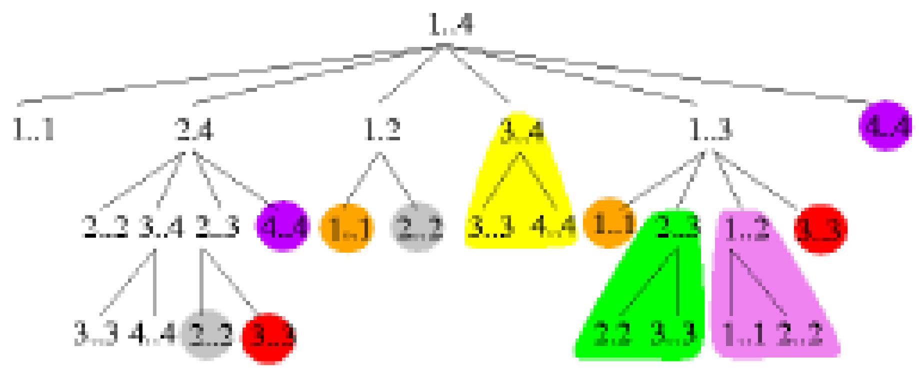
Backtracking/exhaustive search



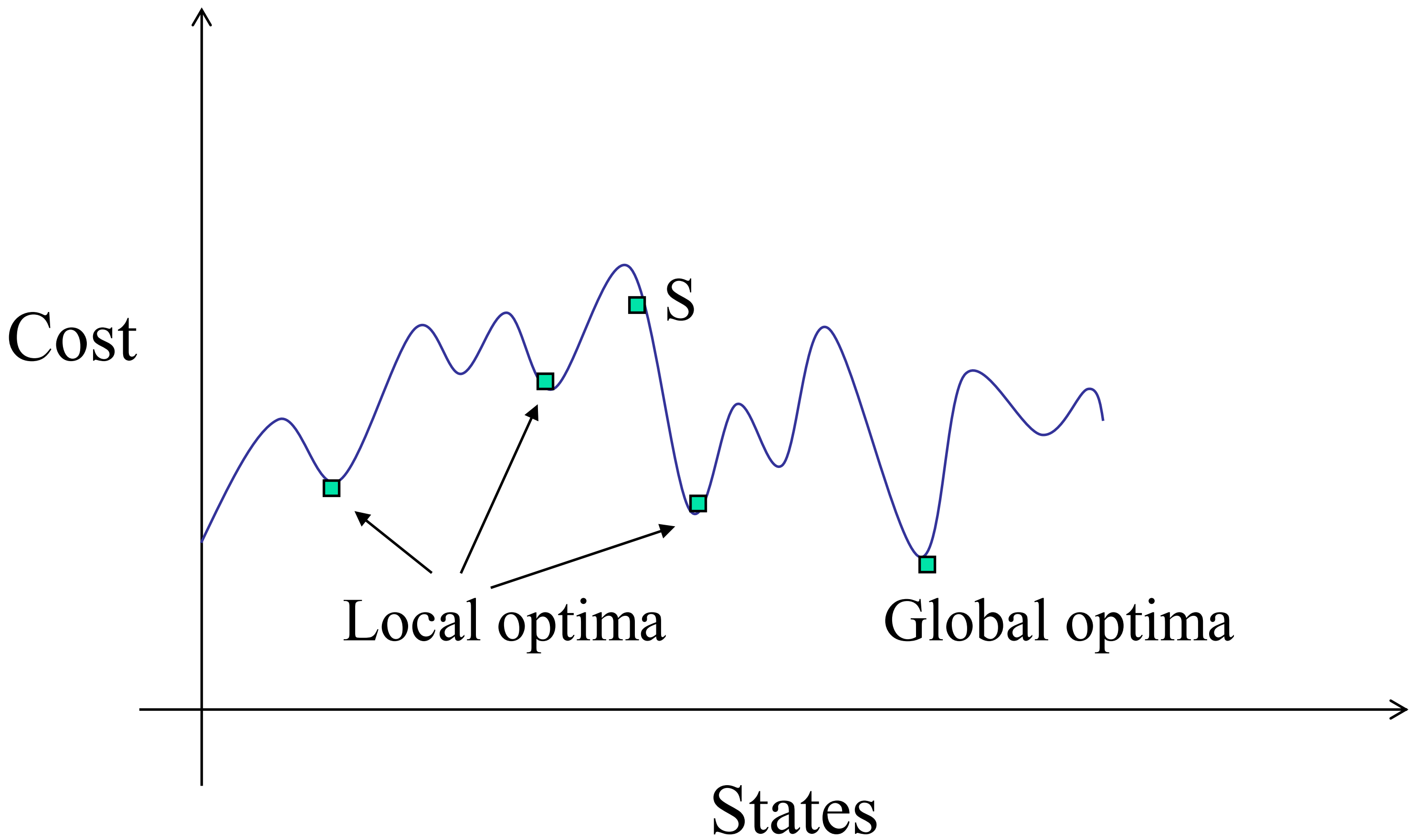

Branch and bound

# 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** $1 \le i \le P$ **do**
      Pick a random neighbor $S'$ of S;
      $\Delta = Cost(S') - Cost(S)$
      **if** $\Delta \le 0$ **then** $S \leftarrow S'$ // down-hill move
      **if** $\Delta > 0$ **then** $S \leftarrow S'$ with probability $e^{-\Delta/T}$ // up-hill
    $T \leftarrow rT$; // reduce temperature
  **return** S
**End**

# 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)

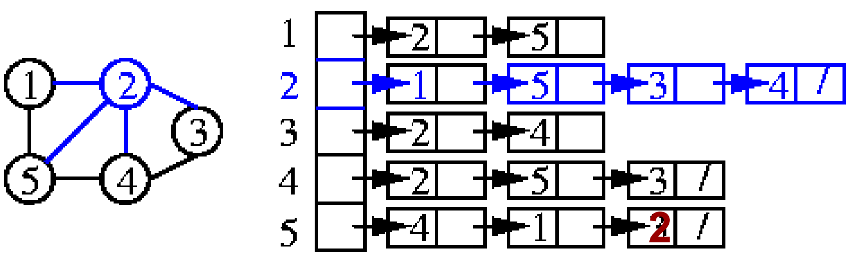
- The ***length*** of a path is the number of edges in it.
  - A path is a (u,v) path if u and v are the terminal vertices.
- A ***cycle*** is a path $(v_0,v_k)$ of length k (k > 2) where $v_0 = v_k$.
  - Odd cycle if k is odd, Even cycle if k is even.
- A ***connected component*** of G is a subgraph 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)

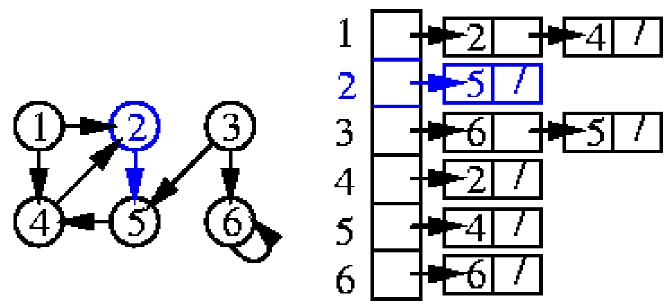
- A ***tree*** is a connected subgraph with no cycles.
- A ***directed graph*** is a pair of sets (V,E) where E is a set of ordered pairs of distinct vertices, called directed edges.
- A ***directed acyclic graph*** (DAG) is a directed graph with no cycles.
- ***Hypergraph*** is a pair (V,E) where V 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.



undirected graph



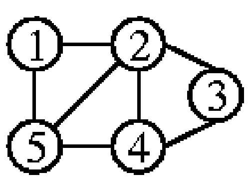
directed graph

# Representations of Graphs: Adjacency Matrix

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

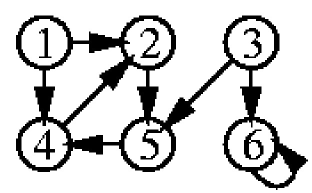
$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

- 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?



| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 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 |

undirected graph



| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 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 |

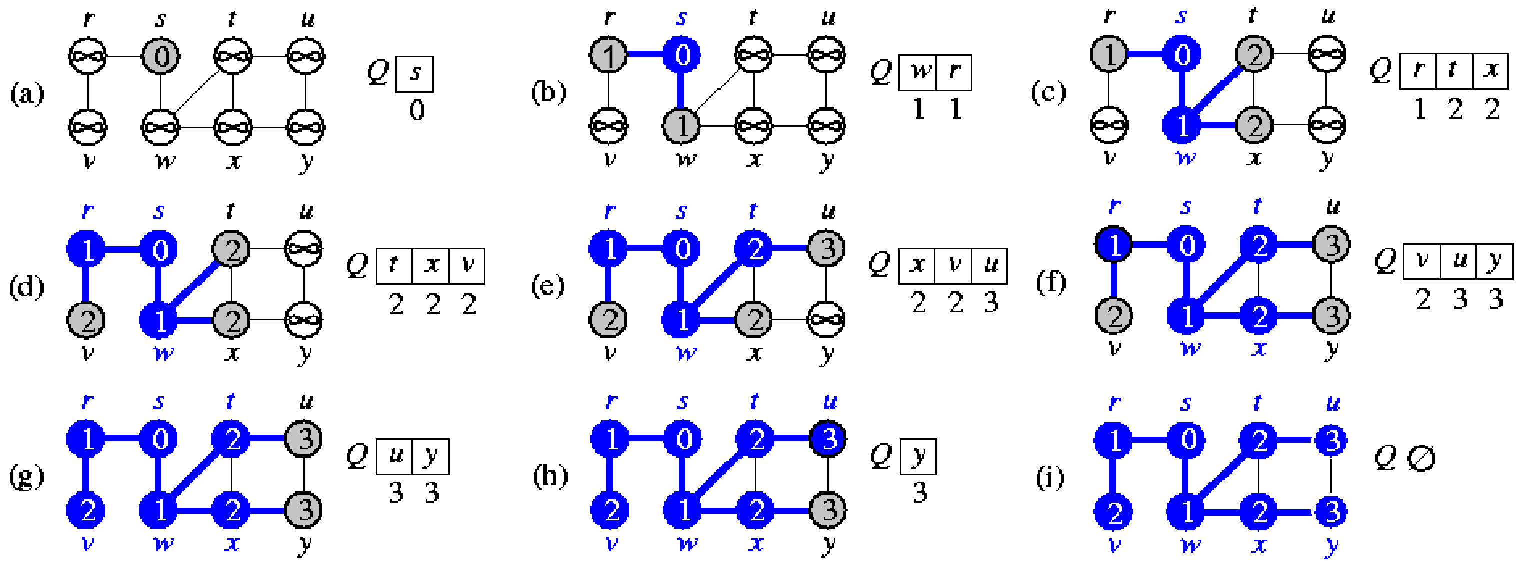
directed graph

# Breadth-First Search (BFS)

```
BFS(G, s)
for each vertex u ∈ V[G]-{s} do
    color[u] ← WHITE
    d[u] ← ∞
    π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← ∅
Enqueue(Q, s)
while Q ≠ ∅ do
  u ← Dequeue[Q]
  for each v ∈ Adj[u] do
      if color[v] = WHITE then
          color[v] ← GRAY
          d[v] ← d[u]+1
          π[v] ← u
          Enqueue(Q, v)
  color[u] ← BLACK
```

- *color*[*u*]: white (undiscovered) → gray (discovered) → 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: $G_\pi = (V_\pi, E_\pi)$, $V_\pi = \{v \in V \mid \pi[v] \neq \text{NIL}\} \cup \{s\}$, $E_\pi = \{(\pi[v], v) \in E \mid v \in V_\pi - \{s\}\}$.

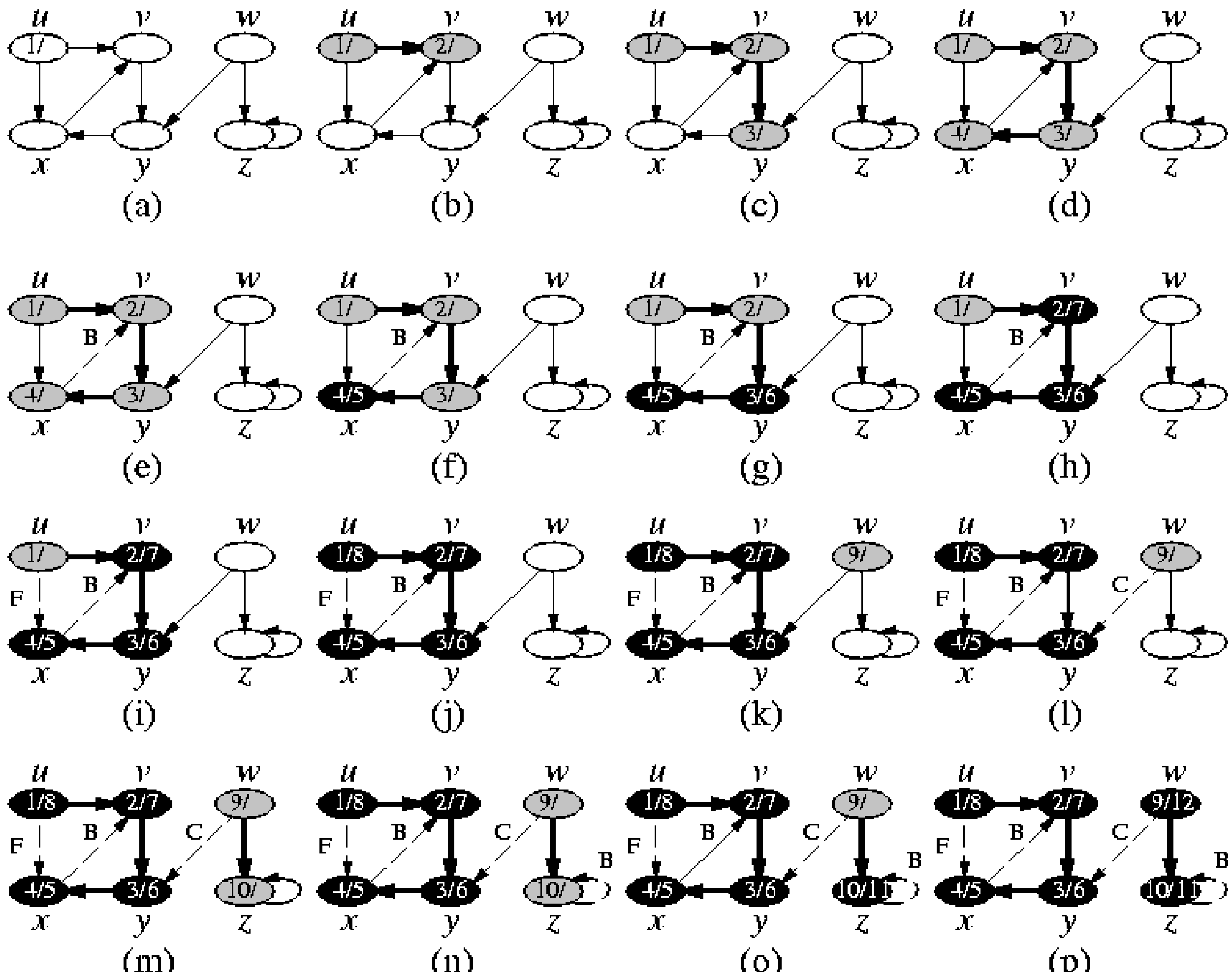
# Depth-First Search (DFS)

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



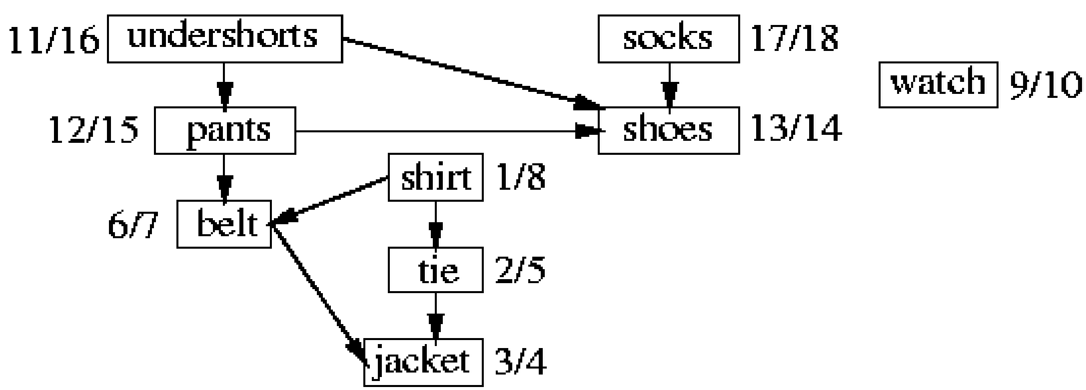

- *color*[*u*]: white → gray → black.
- Depth-first **forest**: $G_\pi = (V, E_\pi)$, $E_\pi = \{(\pi[v], v) \in E \mid v \in V, \pi[v] \neq \text{NIL}\}$.

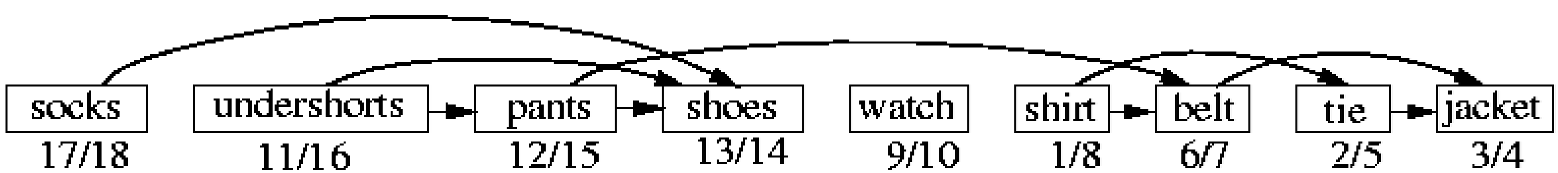
# Topological Sort

- 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

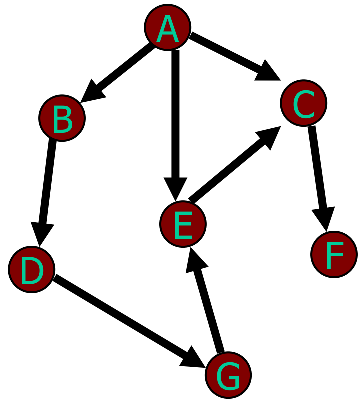
- A directed acyclic graph always contains a vertex with indegree 0.

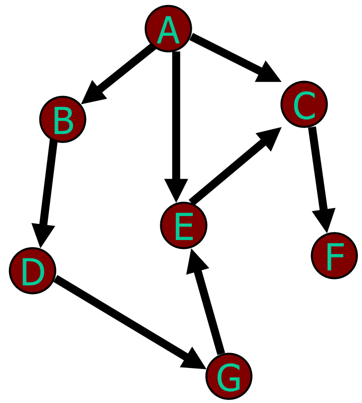
```
Topological-Sort2(G)
1. Call DFS(G) to compute indegree[v] for each vertex v ∈ V[G]
2. Q ← ∅
3. label ← 0
4. for each vertex v ∈ V[G] do
5.      if indegree[v] = 0 then
6.          Enqueue(Q, v)
7. while Q ≠ ∅ do
8.    u ← Dequeue(Q)
9.    label[u] ← label ← label+1
10.   for each v ∈ Adj[u] do
11.       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.



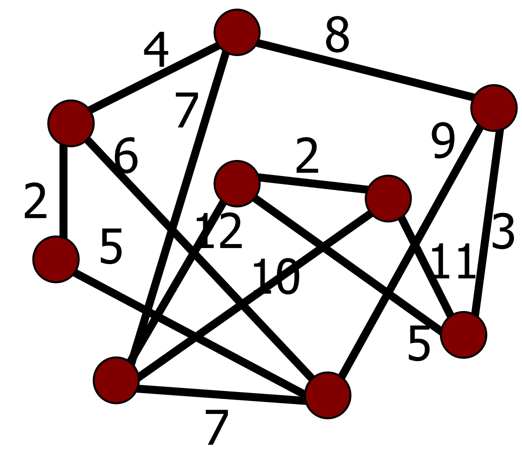



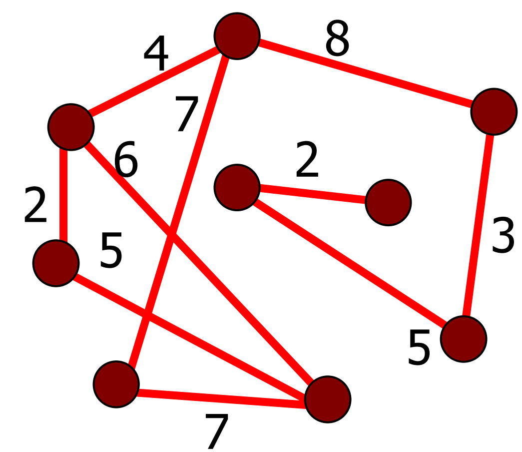

# Spanning Tree Algorithms

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



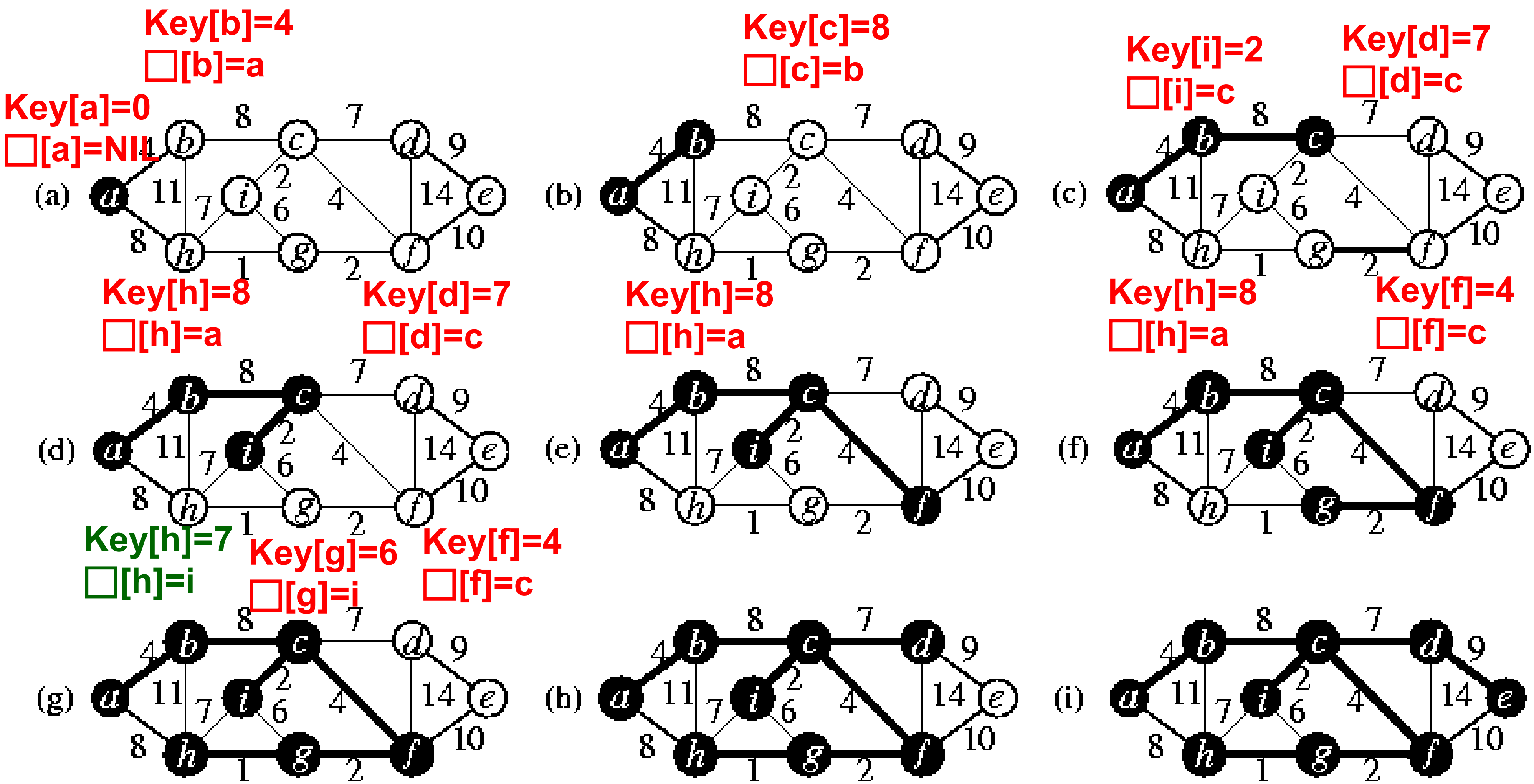



# Prim's (Prim-Dijkstra's?) MST Algorithm

```
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 ∈ V[G] do
2.    key[u] ← ∞
3.    π[u] ←  NIL
4. key[r] ← 0
5. Q ← V[G]
6. while Q ≠ ∅ do
7.    u ← Extract-Min(Q)
8.    for each vertex v ∈ Adj[u] do
9.       if v ∈ Q and w(u,v) < key[v] then
10.           π[v] ← u
11.           key[v] ← 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



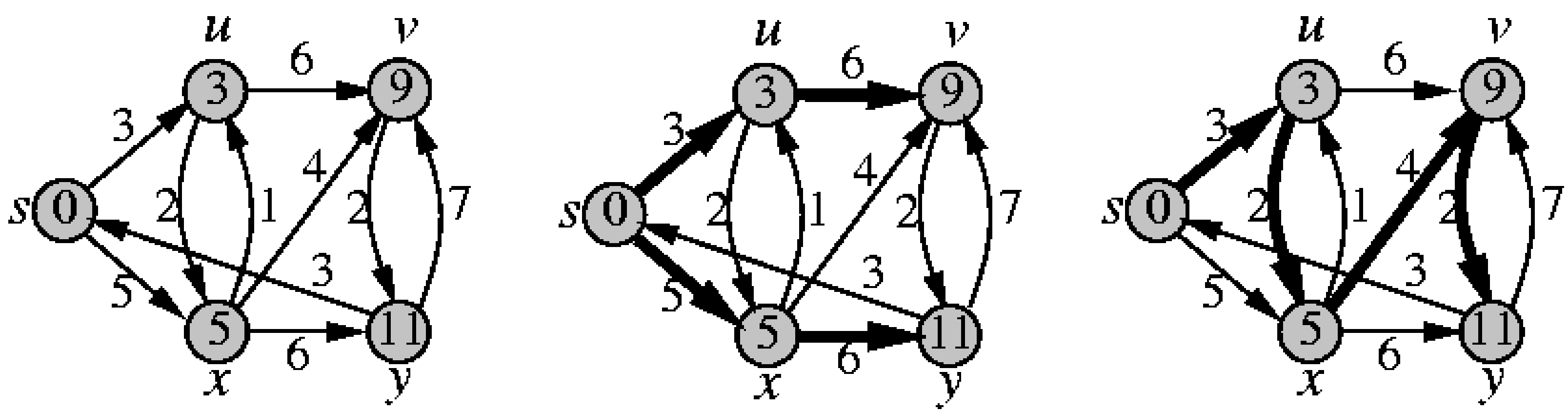
# Time Complexity of Prim's MST Algorithm

```
MST-Prim(G,w,r)
1. for each vertex u ∈ V[G] do
2.     key[u] ← ∞
3.     π[u] ← NIL
4. key[r] ← 0
5. Q ← V[G]
6. while Q ≠ ∅ do
7.     u ← Extract-Min(Q)
8.     for each vertex v ∈ Adj[u] do
9.         if v ∈ Q and w(u,v) < key[v] then
10.            π[v] ← u
11.            key[v] ← 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).

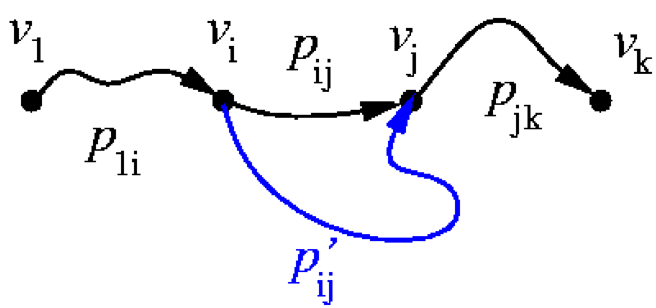


# Variants on Shortest-Paths Problem

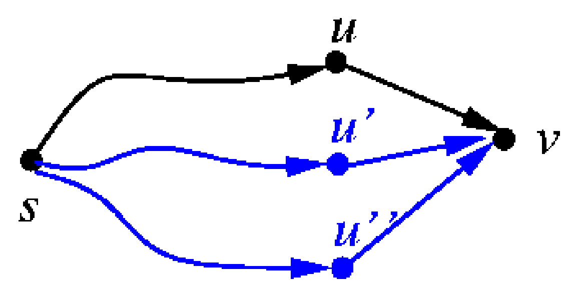
- Single-source shortest-paths problem
- Single-destination shortest-paths problem
- Single-pair shortest-path problem
- All-pairs shortest-paths problem

# Optimal Substructure of a Shortest Path

- Subpaths of shortest paths are shortest paths.
  - Let $p = \langle v_1, v_2, \ldots, v_k \rangle$ be a shortest path from vertex $v_1$ to vertex $v_k$, and let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ 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 $s \overset{p'}{\rightsquigarrow} u \rightarrow v$. 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

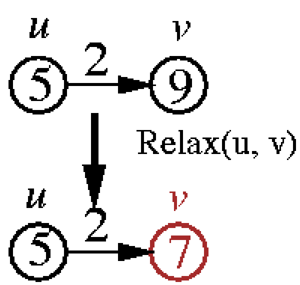


# Relaxation

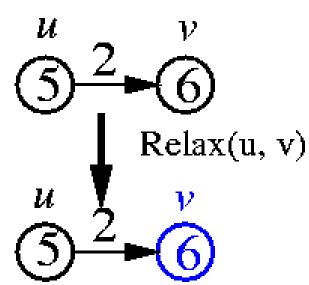
Initialize-Single-Source($G$, $s$)
1. **for** each vertex $v \in V[G]$ **do**
2. $d[v] \leftarrow \infty$
   /* shortest-path estimate, upper bound on the weight of a shortest path from $s$ to $v$ */
3. $\pi[v] \leftarrow$ NIL /* predecessor of $v$ */
4. $d[s] \leftarrow 0$

*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.




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

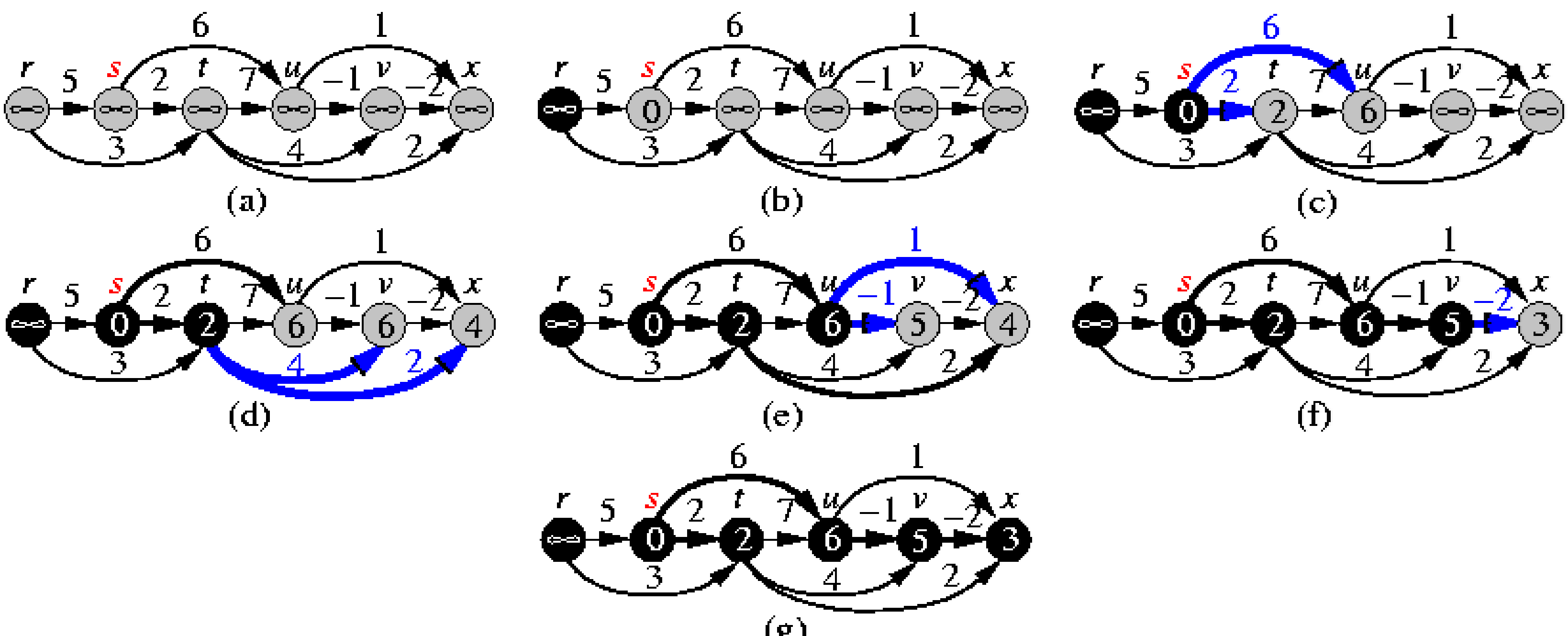



$d[v] <= d[u] + w(u, v)$

# SSSPs in Directed Acyclic Graphs (DAGs)

```
DAG-Shortest-Paths(G, w, s)
topologically sort the vertices of G
Initialize-Single-Source(G, s)
for each vertex u taken in topologically sorted order do
    for each vertex v ∈ Adj[u] do
        Relax(u, v, w)
```

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

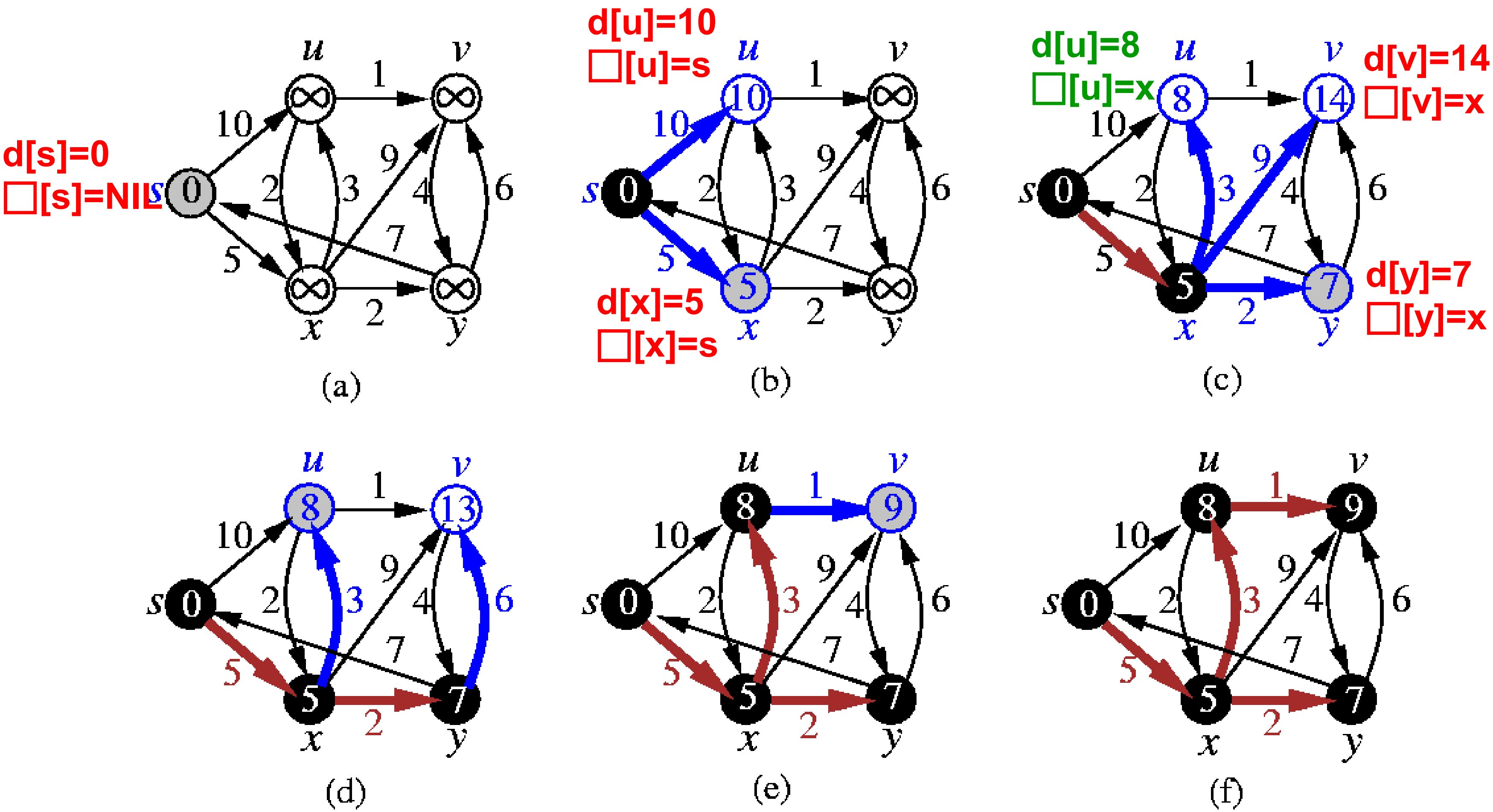


# Dijkstra's Shortest-Path Algorithm

```
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 ← ∅
3. Q ← V[G]
4. while Q ≠ ∅ do
5.    u ← Extract-Min(Q)
6.    S ← S ∪ {u}
7.    for  each vertex v ∈ 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



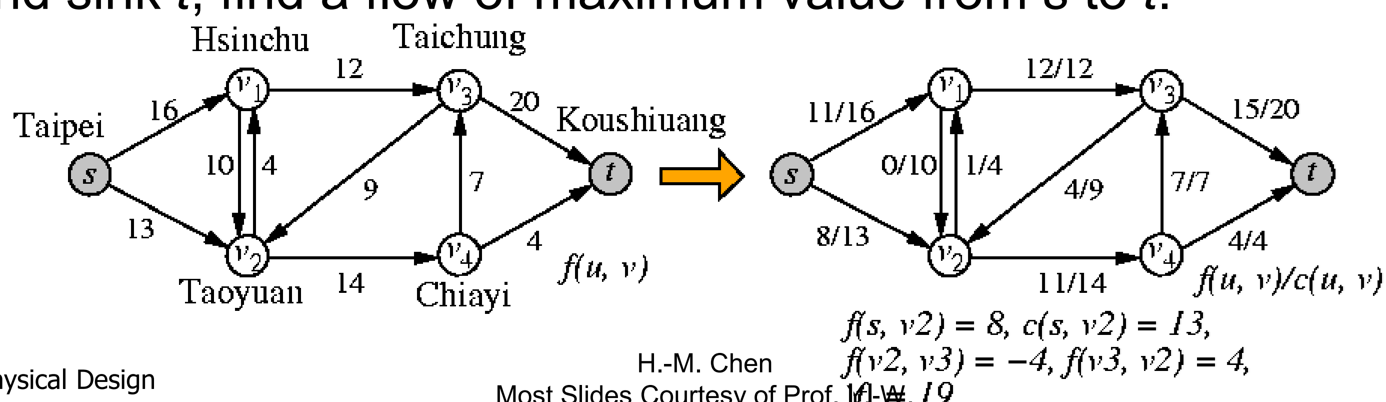
# Runtime Analysis of Dijkstra's Algorithm

```
Dijkstra(G, w, s)
1. Initialize-Single-Source(G, s)
2. S ← ∅
3. Q ← V[G]
4. while Q ≠ ∅ do
5.     u ← Extract-Min(Q)
6.     S ← S ∪ {u}
7.     for each vertex v ∈ 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: $O(E + V \lg V)$.

# 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
```

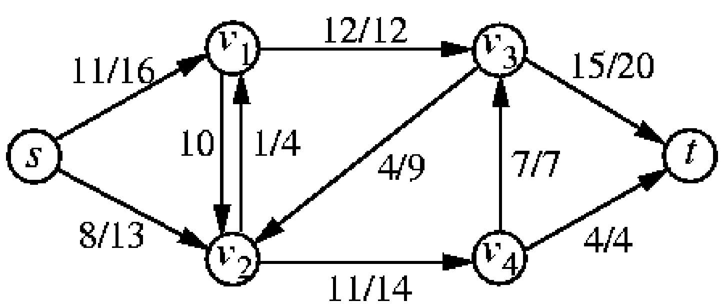
- Ford & Fulkerson, 1956
- **Augmenting path:** A path from *s* to *t* along which we can push more flow.
- Need to construct a **residual network** to find an augmenting path.

# Residual Network

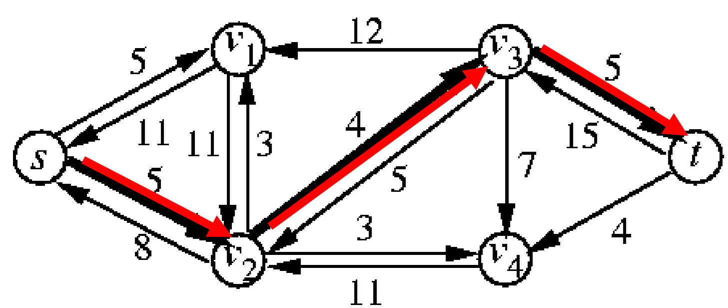
- Construct a **residual network** to find an augmenting path.
- **Residual capacity of edge ($u$, $v$), $c_f(u, v)$:** Amount of **additional** net flow that can be pushed from $u$ to $v$ before exceeding $c(u, v)$,

  $$c_f(u, v) = c(u, v) - f(u, v).$$
- $G_f = (V, E_f)$: **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** $f + f'$: $V \times V \to \mathbb{R}$: $(f + f')(u, v) = f(u, v) + f'(u, v)$ 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

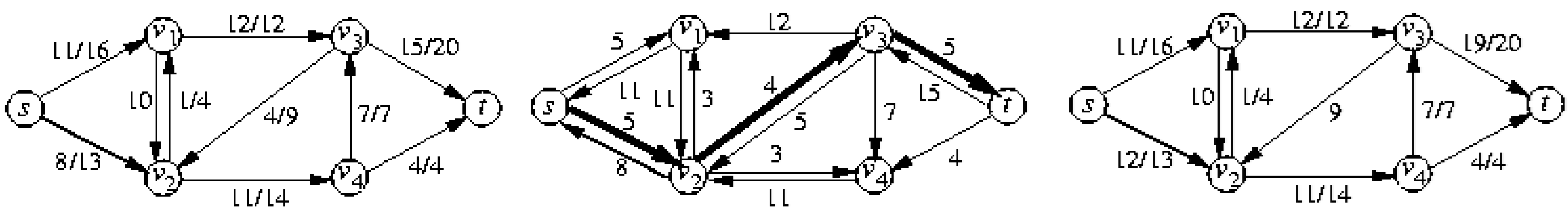
- An **augmenting path** $p$ is a simple path from $s$ to $t$ in the residual network $G_f$.
  - $(u, v) \in E$ on $p$ in the **forward** direction (a **forward edge**), $f(u, v) < c(u, v)$.
  - $(u, v) \in E$ on $p$ in the **reverse** direction (a **backward edge**), $f(u, v) > 0$.
- **Residual capacity** 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 $f_p$: $V \times V \to \mathbf{R}$ 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 $|f_p| = c_f(p) > 0$.



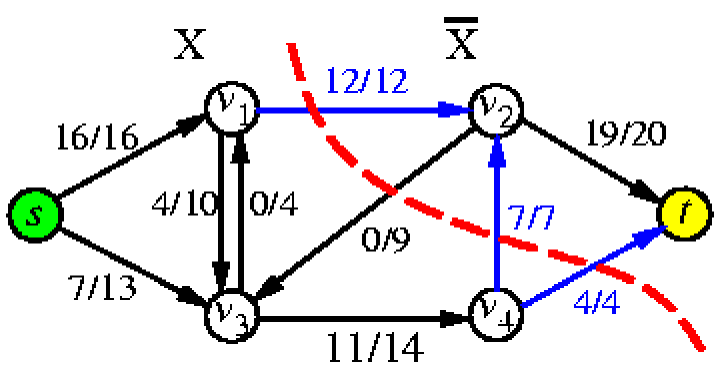

(a)

(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 $(S, T)$.




flow/capacity

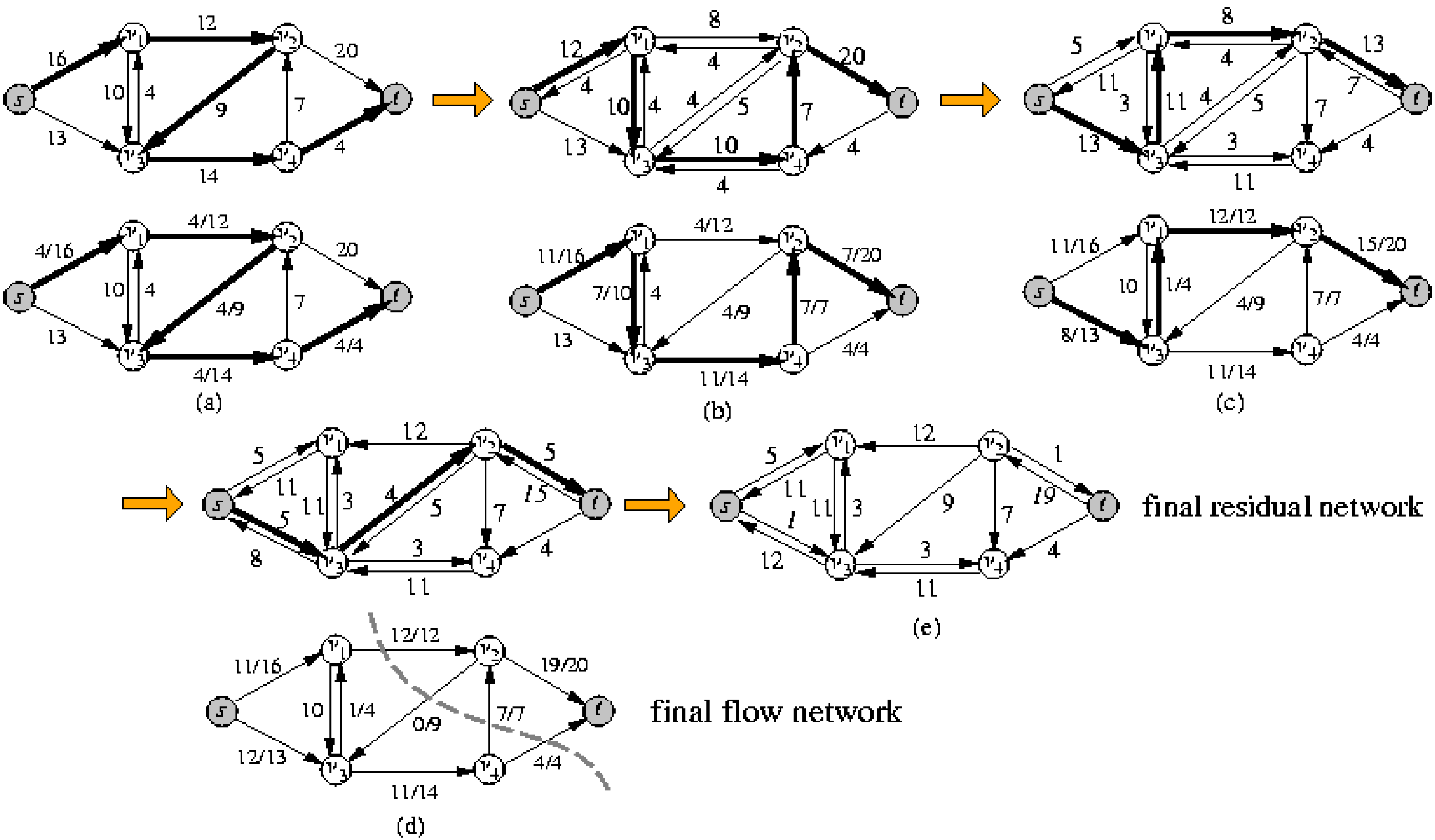
max flow $|f| = 16 + 7 = 23$
$cap(X, \bar{X}) = 12 + 7 + 4 = 23$

# 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

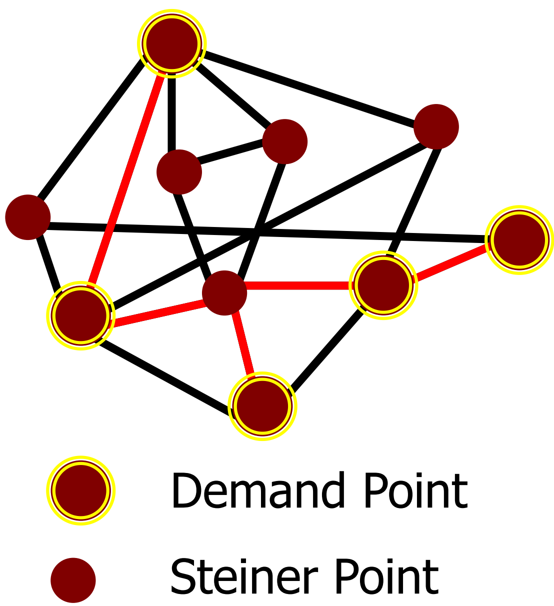


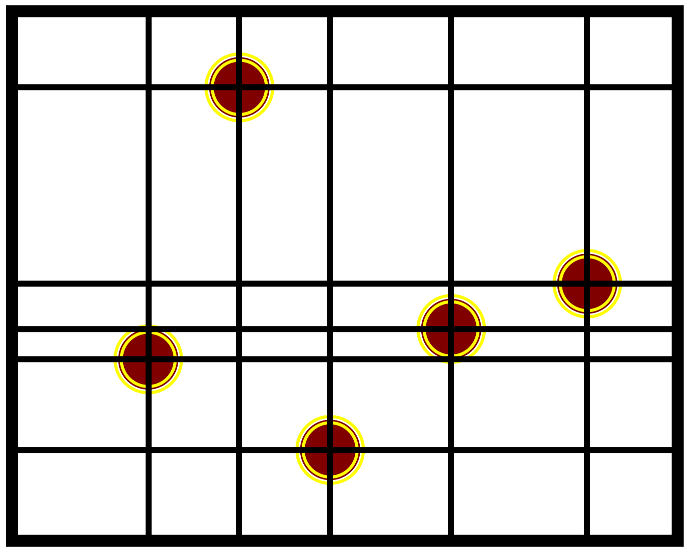
# Steiner Tree Algorithms (1/4)

- 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

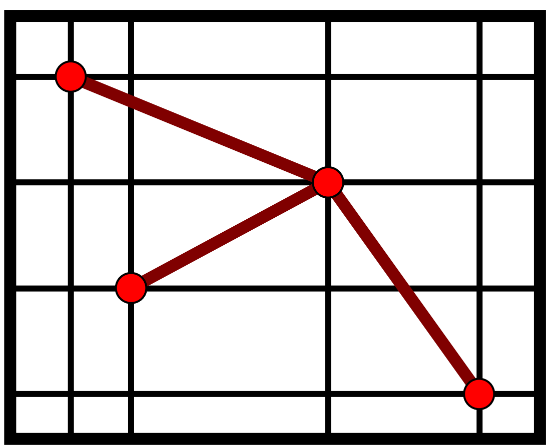


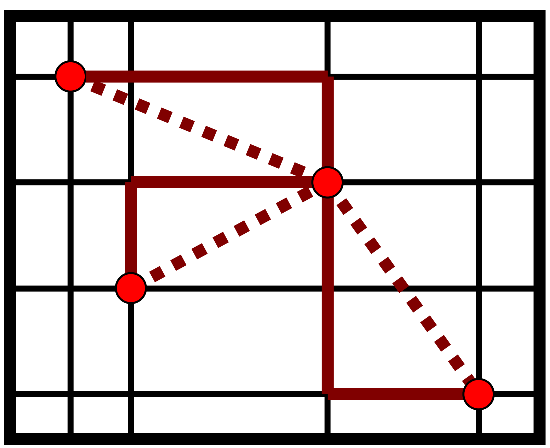



# Steiner Tree Algorithms (3/4)

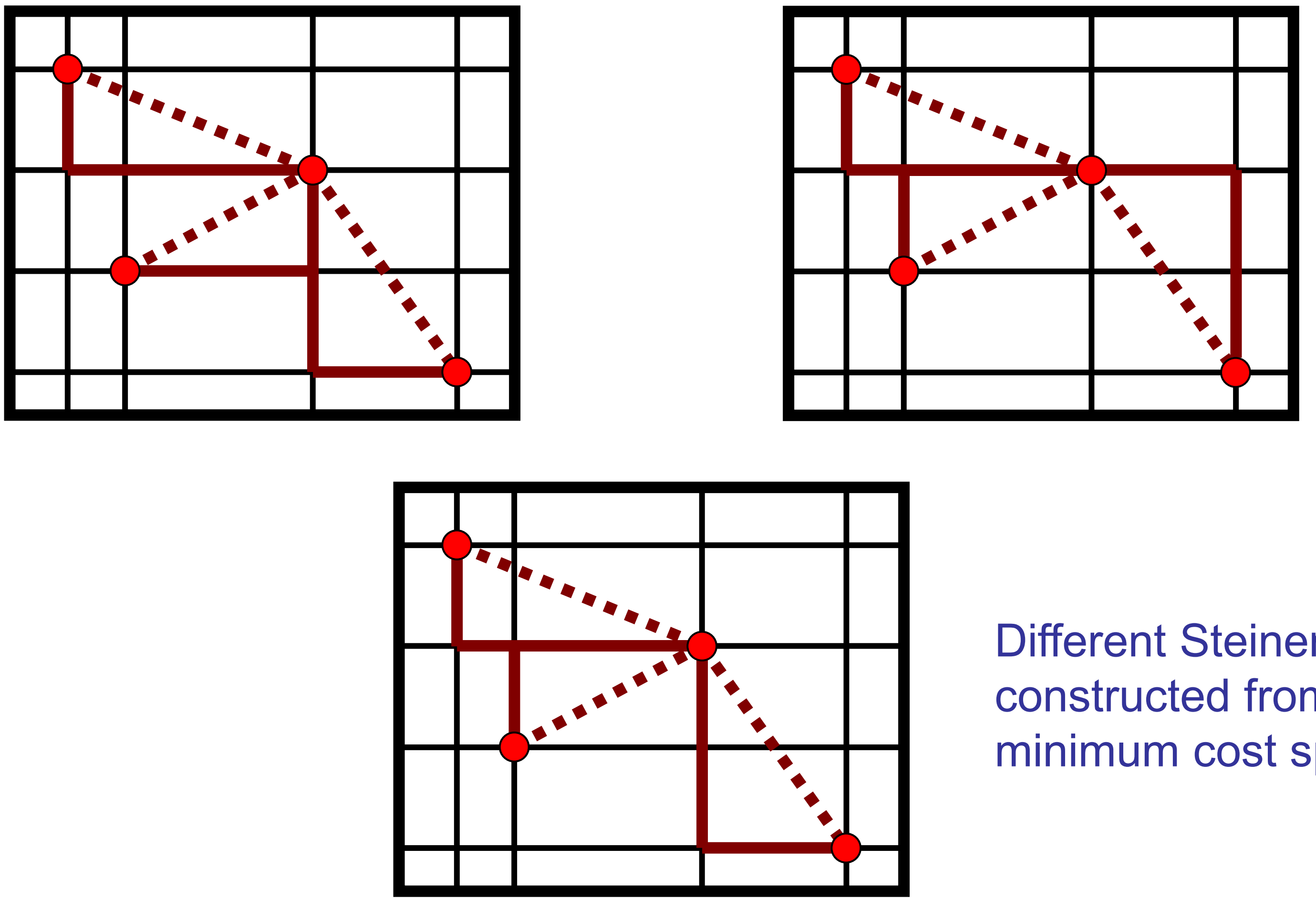
- Rectilinear Steiner Tree (RST) – a steiner tree whose edges are restricted to rectilinear shape
- Rectilinear Steiner Minimum Tree (RSMT)
- Theorem:

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





# Steiner Tree Algorithms (4/4)



Different Steiner trees constructed from a minimum cost spanning

# Appendix: EDA 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