Topic 4: Graph Algorithms

Agenda

- Searching: Breadth-first, Depth-first
- Minimum Spanning trees
- Single-Source Shortest Paths
- Maximum Flow

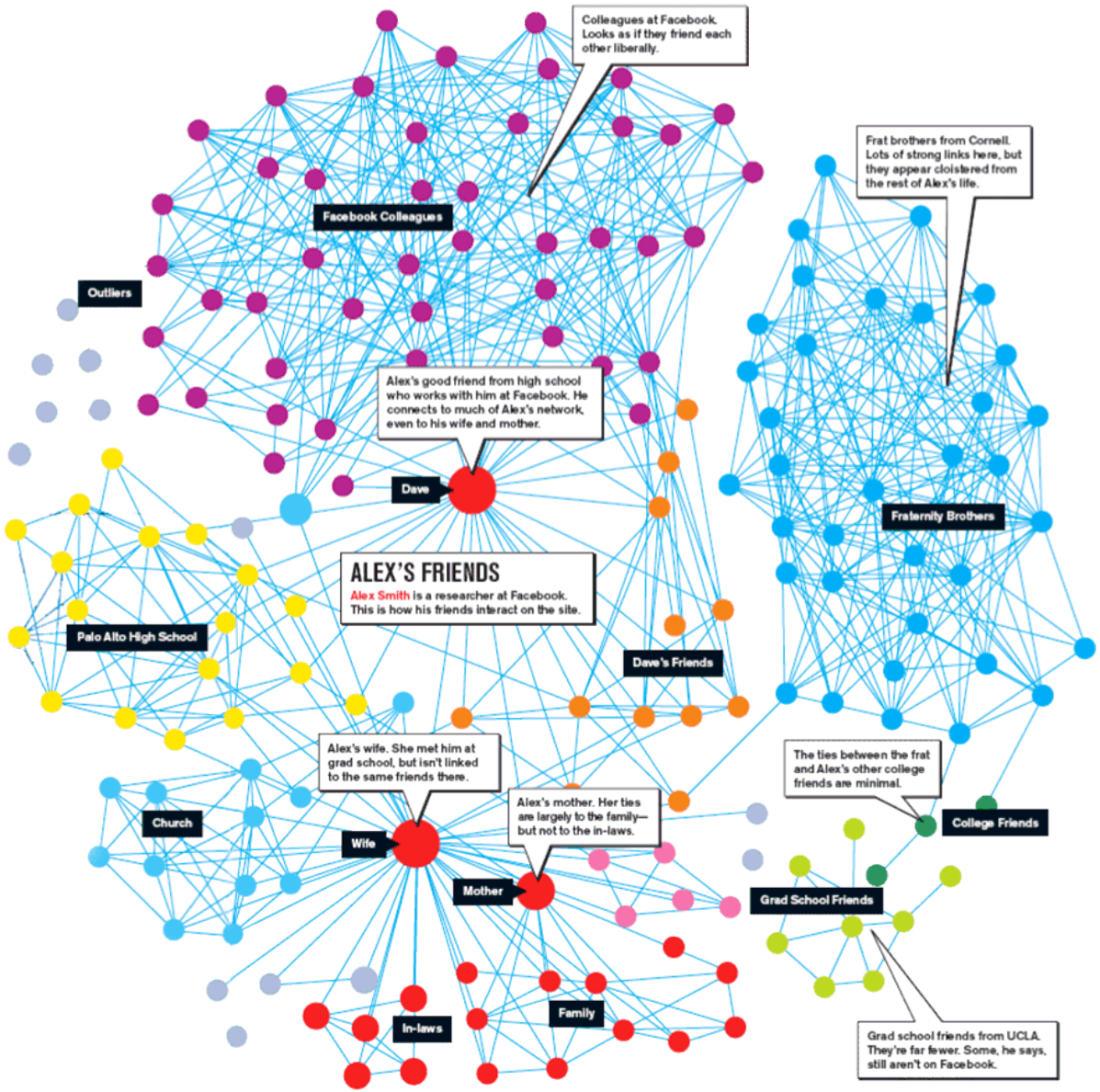
Graphs and their Applications

A graph is an ordered pair $G = (V, E)$ comprising a set V of vertices (nodes or points) together with a set E of edges (arcs or lines), which are 2-element subsets of V.

Applications:

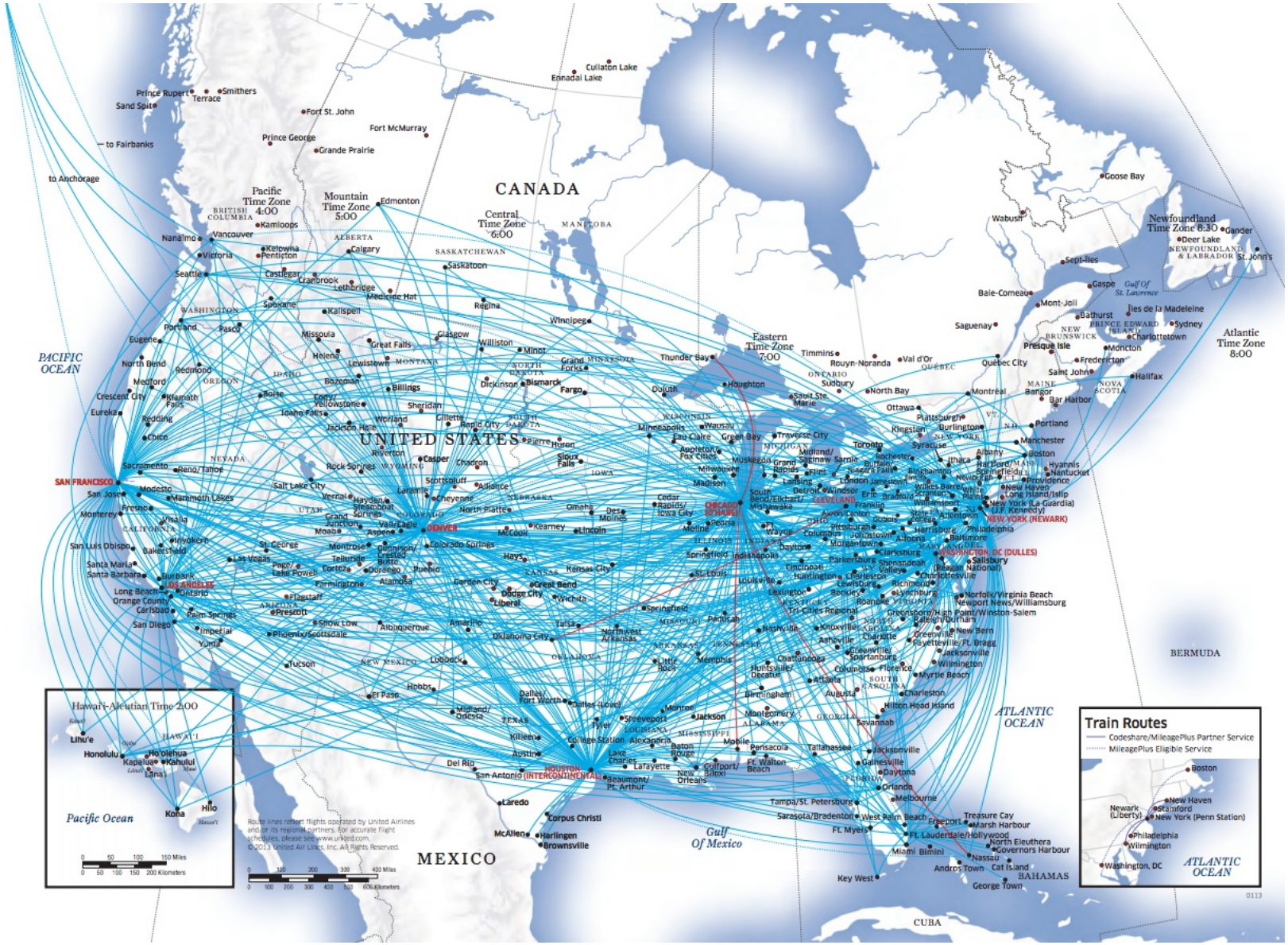
- Social Networks
- Transportation Networks
- Communication Networks
- Semantic Networks
- State Transitions in Multi-Stage Decision Problems
- Biology: Gene Regulatory Networks
- Circuits: Kirchoff's Laws
- and so on...

Example 1: Social Networks



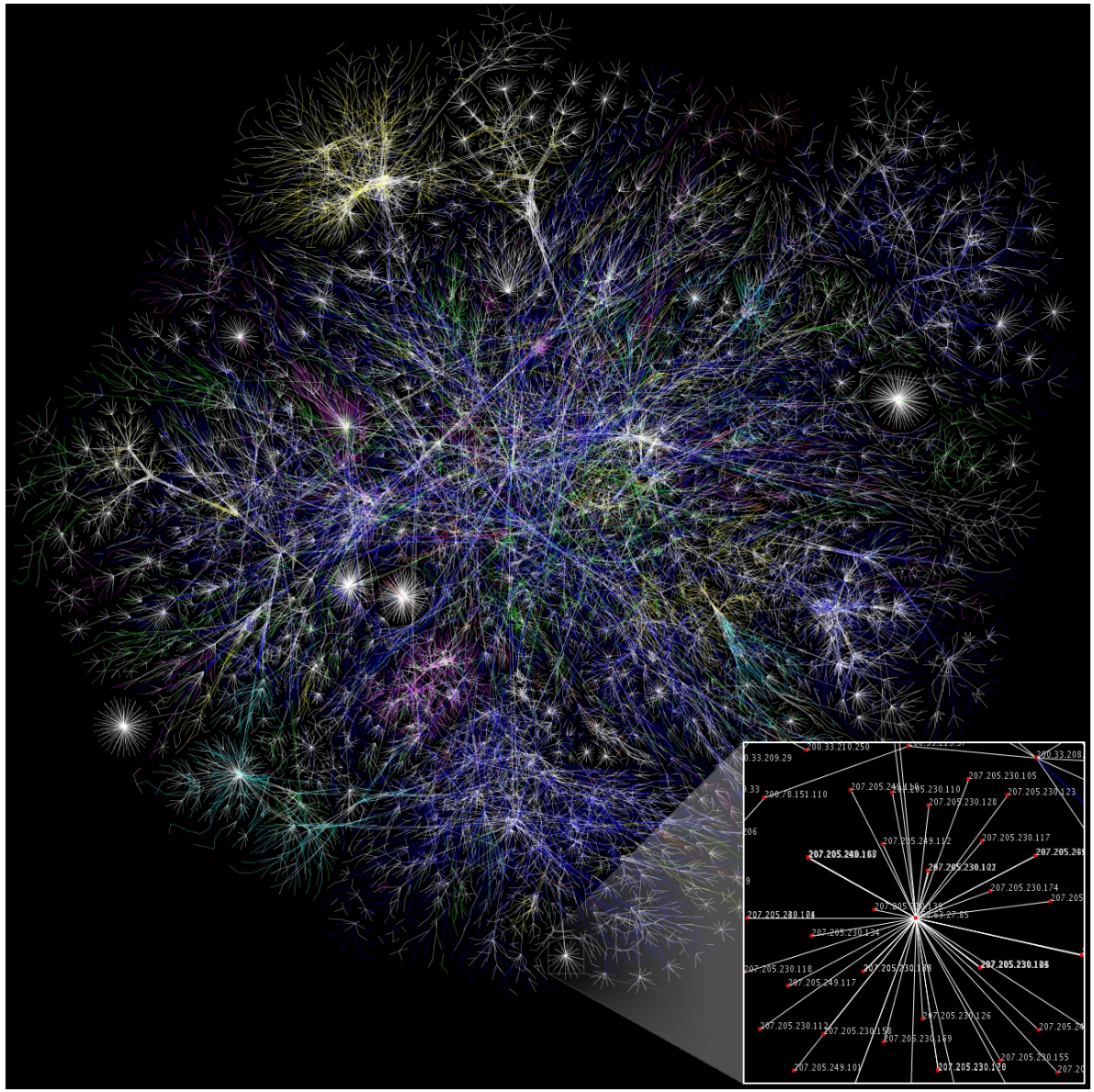
Source: https://i.pinimg.com/originals/81/3b/67/813b67df338d91bc9e4cef103d8e297b.jpg

Example 2: Transportation Networks



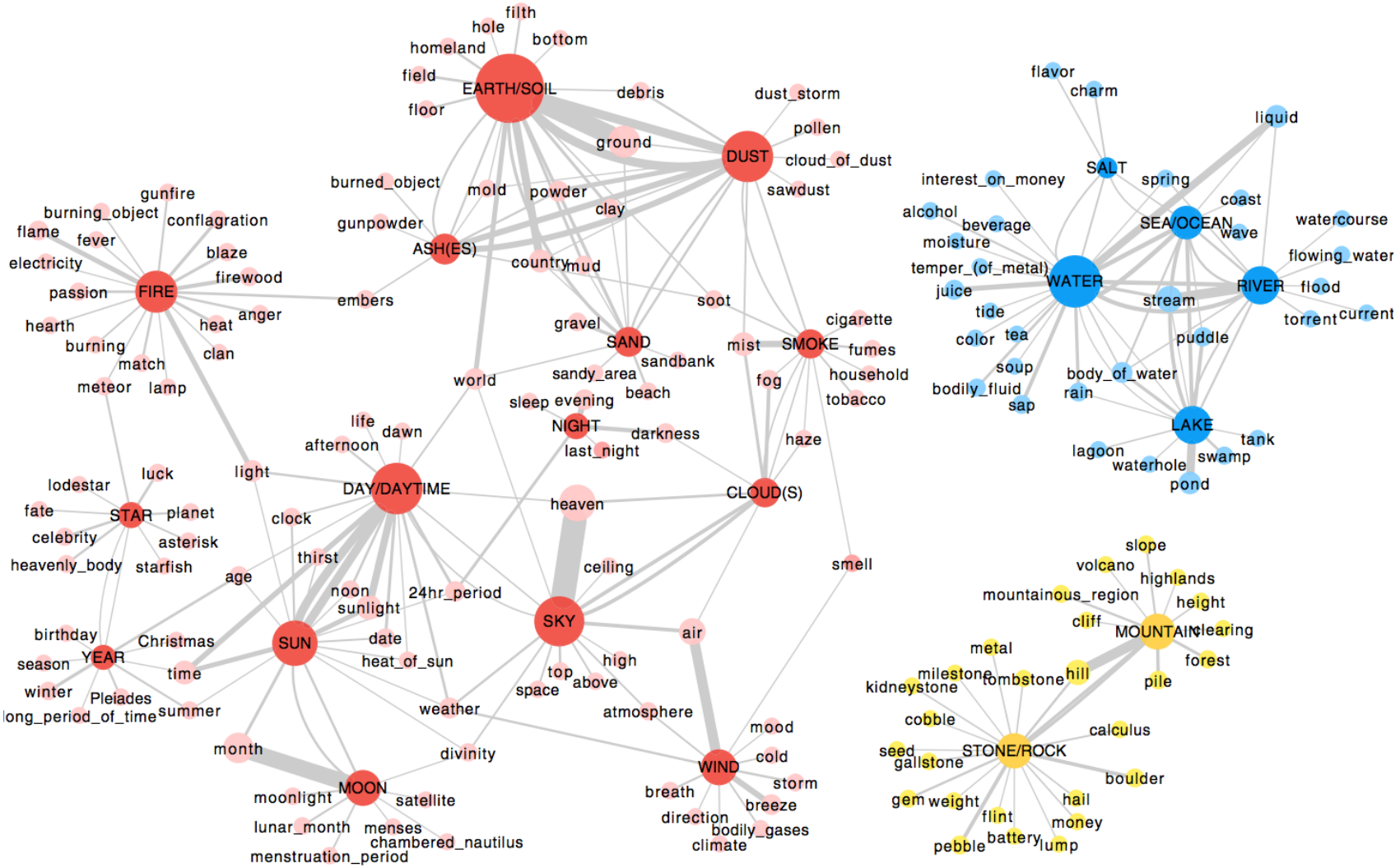
Source: https://worldairlinenews.files.wordpress.com/2013/01/united-domestic-12013-route-map.jpg

Example 3: Communication Networks



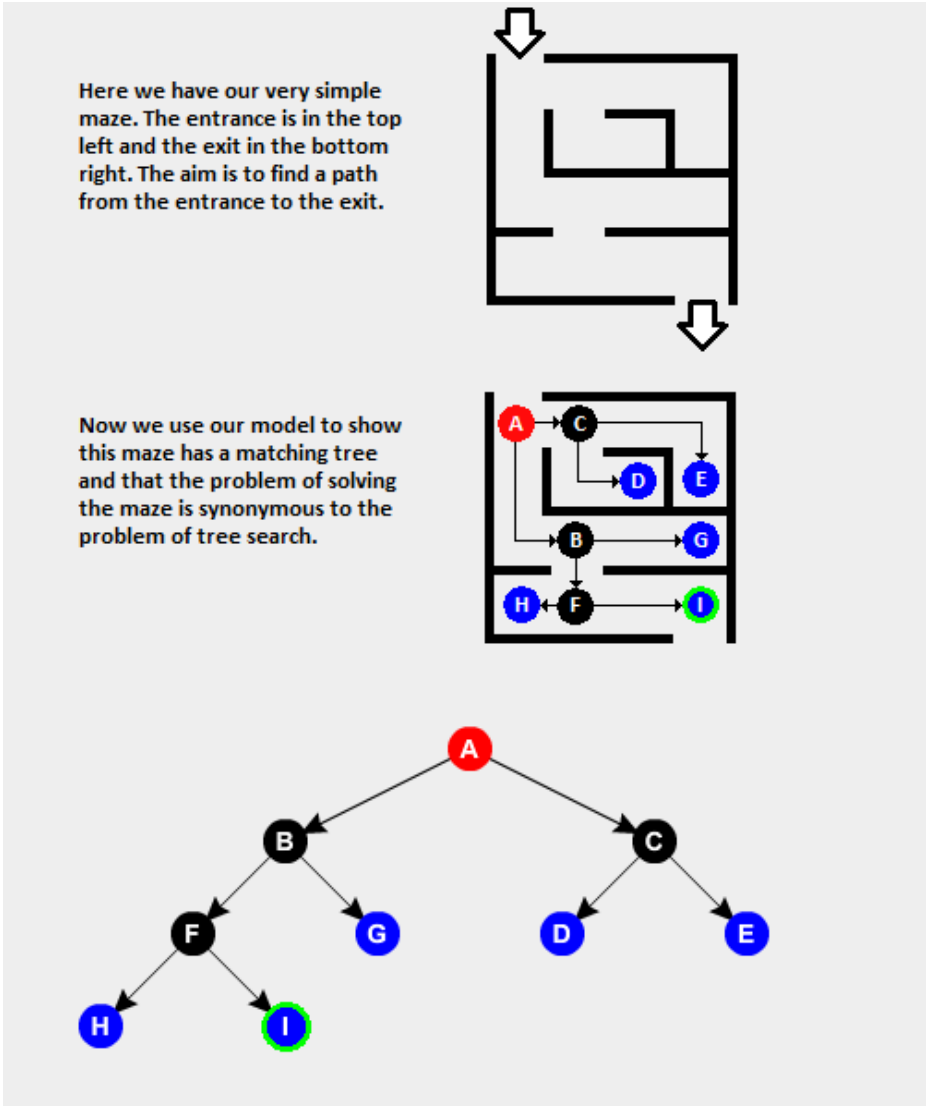
Source: https://en.wikipedia.org/wiki/Computer_network/media/File:Internet_map_1024.jpg

Example 4: Semantic Networks



Source: http://hyoun.me/language/fig_semanticnetwork.png

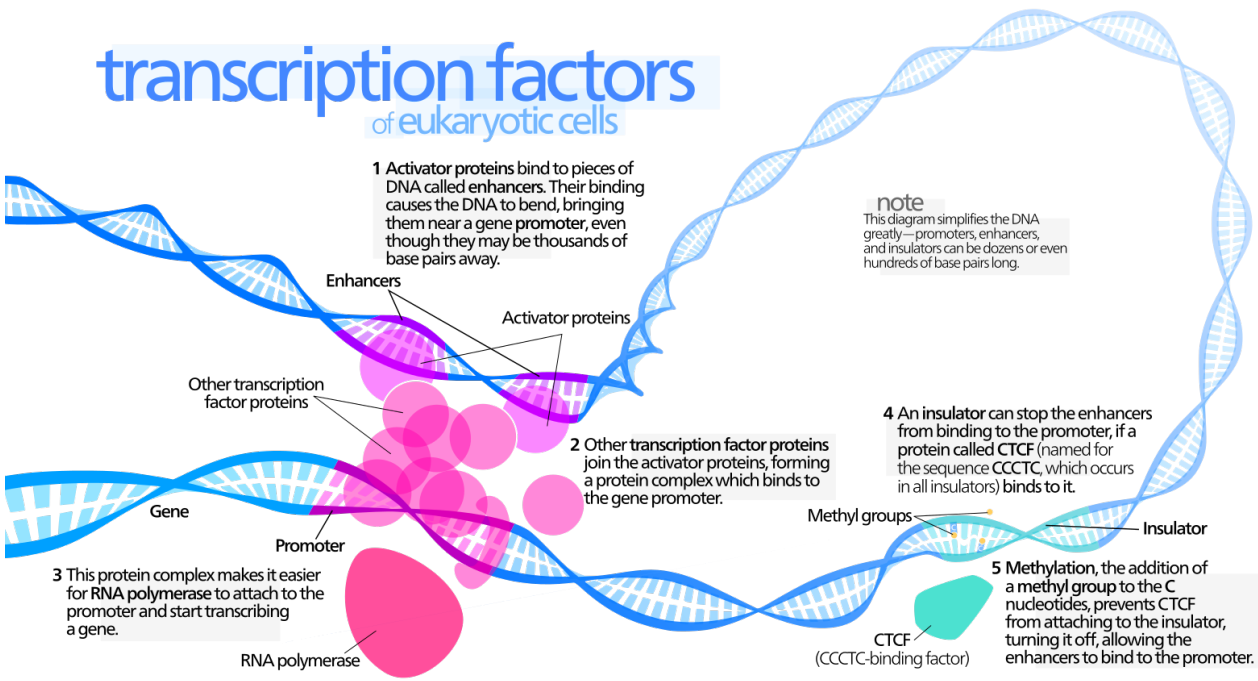
Example 5: State Transition Graphs



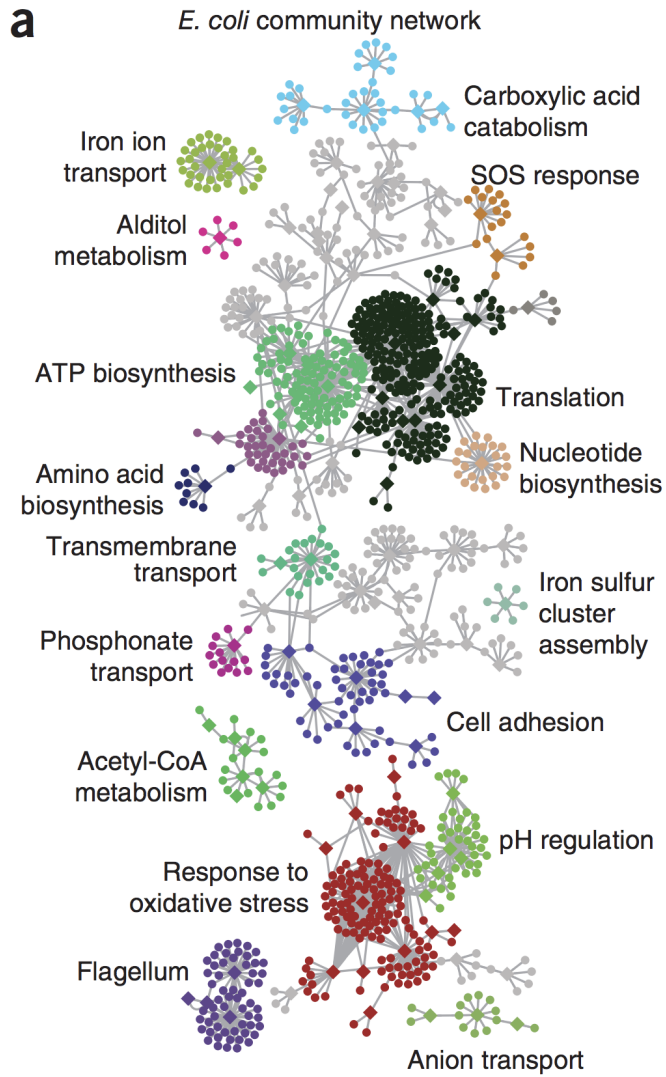
Source: https://computationalgamedesign.files.wordpress.com/2016/09/puzzle-example4.png

Example 6: Gene Regulatory Networks (GRNs)

- DNA comprises of a sequence of genes, each having a specific functionality.
- A gene is activated, when a protein interacts with DNA during the transcription process.
- Some genes, when activated, can lead to diseases.



Example 6: GRNs in e coli



Source: D. Marbach *et al.* "Wisdom of crowds for robust gene network inference," *Nature methods*, vol. 9, no. 8, pp. 796-804. 15 Jul. 2012.

Representation of Graphs

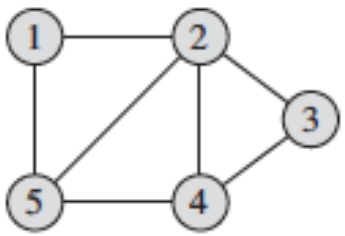
Two styles of representation:

- **Adjacency list:** Dictionary of lists of adjacent vertex to each vertex.
  - Compact way to represent sparse graphs ($|E| \ll |V|^2$)
- **Adjacency Matrix:** Matrix with $\{0, 1\}$ entries, where edge (i, j) is present if the $(i, j)^{th}$ entry is 1.
  - More convenient to represent dense graphs ($|E| \approx |V|^2$)
  - Also, easy to represent weighted graphs.

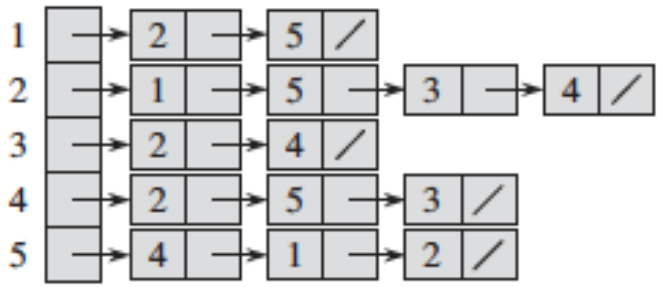
Either method can be used for directed/undirected graphs.

Representation of Graphs

Example 1:



(a)

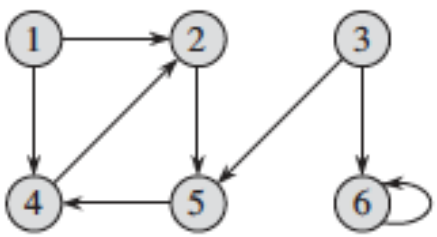


(b)

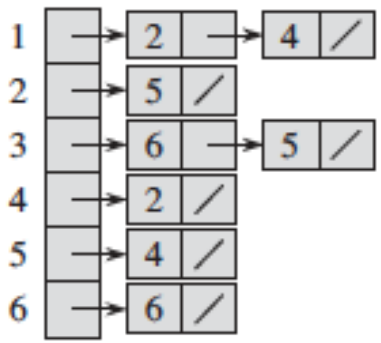
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Example 2:



(a)



(b)

	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

(c)

Graphs: Attributes/Properties

- **Degree/Valency:** Number of edges incident to a vertex.
  - Note: $\sum_{v \in V} d(v) = 2|E|$.
- **Path:** A sequence of edges that trails along a sequence of vertices from a start node to a destination node.
- **Cycle/Loop:** A path wherein the start and destination nodes are identical.
- **Distance:** Length of shortest path between two nodes.
- **Diameter:** Largest distance between any pair of vertices.
- **Connected Components:** A subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.

Graphs: Types

- **Connected Graph:** A graph with at least one path between any pair of vertices.
- **Acyclic Graph:** A graph with no cycles (loops)
- **Simple Graph:** An undirected graph without multiple edges or loops.
- **Trees:** A connected/acyclic graph. (Unique path between any pair of vertices).
- **Forests:** A collection of trees.
- **Bipartite Graph:** A graph in which the vertex set can be partitioned into two sets, W and X, so that no two vertices in W share a common edge and no two vertices in X share a common edge.
- **Complete Graph:** A graph where every pair of nodes lies in E.

Some Interesting Problems on Graphs

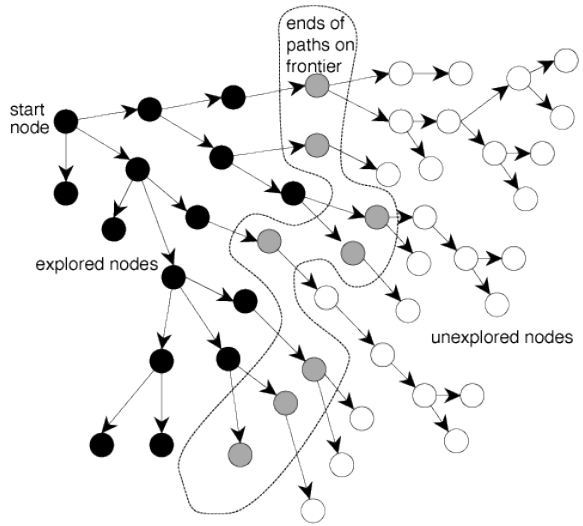
- **Searching over Graphs**
- **Minimum Spanning Tree**
- **Shortest Path**
- **Maximum Flow**
- Hamiltonian Path
- Graph Planarity
- Graph Coloring
- Rigidity of Graphs
- Graph Centrality
- and so on...

Graph Search/Traversal

How to traverse on a graph, given a start node?

- Specify an order to search (traverse) through the nodes.
  - Depth-First Search (DFS)
  - Breadth-First Search (BFS)
- If a target node is specified, keep searching until it is found.
- *Frontier* is the boundary between the set of explored and unexplored nodes.

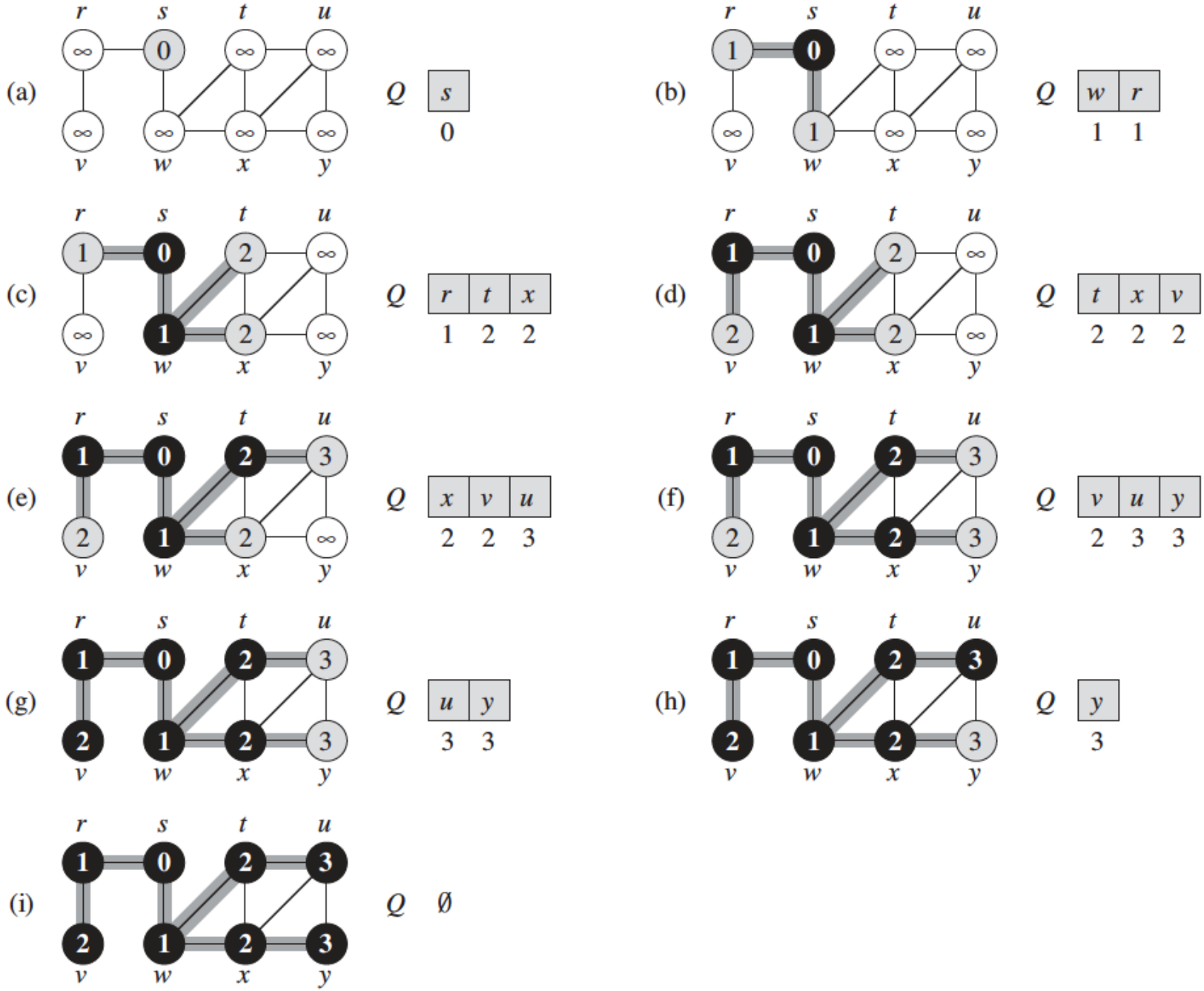
$$visited(u) = \begin{cases} -1, & \text{if } u \text{ is not explored} \\ 0, & \text{if } u \text{ is in the frontier} \\ 1, & \text{if } u \text{ has been explored} \end{cases} .$$



Breadth-First Search

$\text{BFS}(s, G)$

```
 1  for each u ∈ G.V-{s}
 2      visited(u) = −1 // Unexplored, Colored white in picture
 3      level(u) = ∞
 4      parent(u) = ∅
 5  level(s) = 0
 6  parent(s) = ∅
 7  visited(s) = 0 // Frontier, Colored grey in picture
 8  frontier = ∅ // Frontier is a queue.
 9  ENQUEUE(frontier, s)
10  while frontier ≠ ∅
11      u = DEQUEUE(frontier)
12      for each v ∈ G.Adj(u)
13          if visited(v) == −1
14              visited(v) = 0
15              level(v) = level(u) + 1
16              parent(v) = u
17              ENQUEUE(frontier, v)
18      visited(u) = 1 // Explored, Colored black in picture
```


(a)
r s t u
∞ 0 ∞ ∞
∞ ∞ ∞ ∞
v w x y
Q
s
0
(b)
r s t u
1 0 ∞ ∞
∞ 1 ∞ ∞
v w x y
Q
w r
1 1
(c)
r s t u
1 0 2 ∞
∞ 1 2 ∞
v w x y
Q
r t x
1 2 2
(d)
r s t u
1 0 2 ∞
2 1 2 ∞
v w x y
Q
t x v
2 2 2
(e)
r s t u
1 0 2 3
2 1 2 ∞
v w x y
Q
x v u
2 2 3
(f)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q
v u y
2 3 3
(g)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q
u y
3 3
(h)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q
y
3
(i)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q
∅

Analysis of BFS

Run-time: $O(|V| + |E|)$

- Enqueuing and Dequeuing: $O(|V|)$ – Each node is only considered once.
- Time spent to scan adjacency lists: $O(|E|)$
- Initialization: $O(|V|)$

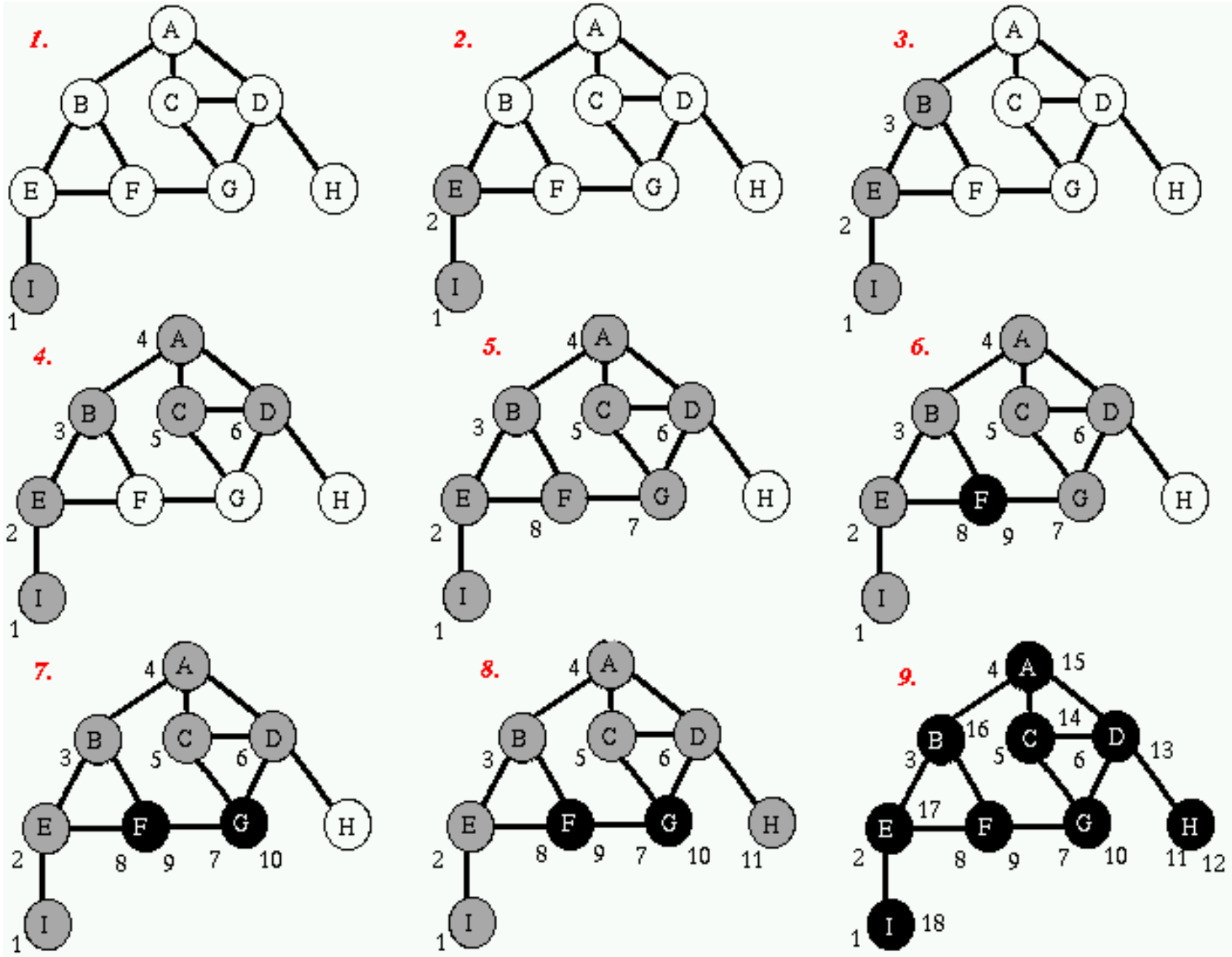
level(u): Distance between s and u.

Applications:

- Shortest paths on unweighted graphs.
- Tracing Garbage: Cheney's algorithm.
- Message Broadcasting
- Web Crawlers (Spiders)

Depth-First Search

```
DFS(G, s)
 1  for each u ∈ G.V-{s}
 2        visited(u) = −1 // Unexplored, Colored white in picture
 3        parent(u) = ∅, adj_count(u) = 0
 4  Frontier = ∅ // Frontier is a stack.
 5  parent(s) = ∅, adj_count(s) = 0
 6  visited(s) = 0 // Frontier, Colored grey in picture
 7  PUSH(Frontier, s)
 8  while Frontier ≠ ∅
 9        u = TOP(Frontier)
10        for each v ∈ G.Adj(u)
11              if visited(v) == −1
12                    visited(v) = 0, parent(v) = u
13                    PUSH(Frontier, v), BREAK
14              else adj_count(u) = adj_count(u) + 1
15        if adj_count(u) == G.Adj(u).size
16              visited(u) = 1
17              POP(Frontier)
```


1.
A
B
C
D
E
F
G
H
I
1
2.
A
B
C
D
E
F
G
H
I
2
1
3.
A
B
C
D
3
E
F
G
H
2
I
1
4.
4
A
B
C
D
3
5
6
E
F
G
H
2
I
1
5.
4
A
B
C
D
3
5
6
E
F
G
H
8
7
2
I
1
6.
4
A
B
C
D
3
5
6
E
F
G
H
8
9
7
2
I
1
7.
4
A
B
C
D
3
5
6
E
F
G
H
8
9
7
10
2
I
1
8.
4
A
B
C
D
3
5
6
E
F
G
H
8
9
7
10
11
2
I
1
9.
4
A
15
B
16
C
14
D
13
3
5
6
E
17
F
G
H
8
9
7
10
11
12
2
I
18
1

Analysis of DFS

Run-time: $O(|V| + |E|)$

- Pushing and Popping: $O(|V|)$ – Each node is only considered once.
- Time spent to scan adjacency lists: $O(|E|)$
- Initialization: $O(|V|)$

Applications:

- Loop Detection in Directed Graphs.
- Topological Sorting
- Strongly connected components in directed graphs: Tarjan's algorithm, Kosaraju's algorithm.

Minimum Spanning Trees (MST)

Problem Statement:
Given a weighted graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$, find a spanning tree (a subset $\tilde{E} \subseteq E$ which forms a tree $T = (V, \tilde{E})$) that

$$\underset{\tilde{E} \subseteq E}{\text{minimize}} \sum_{e \in \tilde{E}} w(e).$$

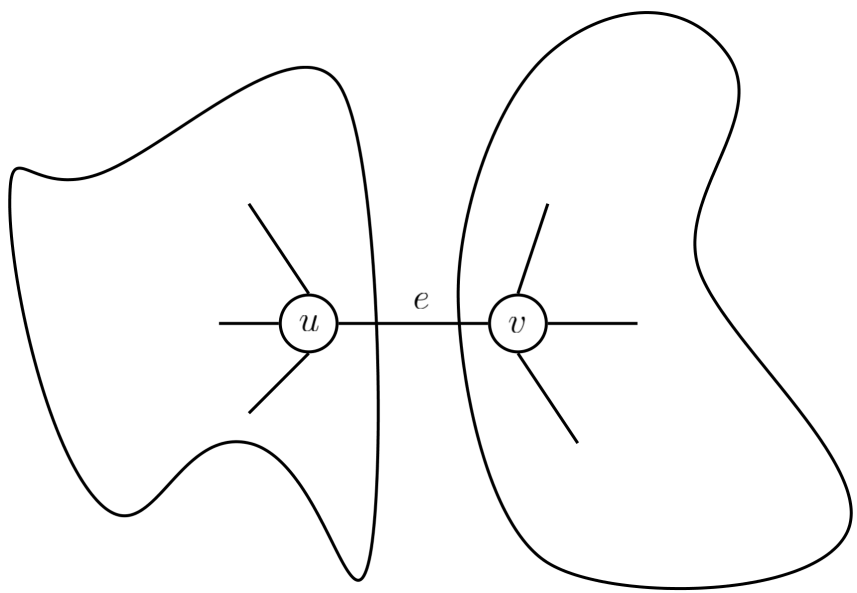
Note 1: Consider a complete graph K_n. Then, the total number of spanning trees possible are n^{n-2}.
Note 2: There could be multiple solutions to this problem.

Can dynamic programming solve this problem efficiently?

MST: Dynamic Programing

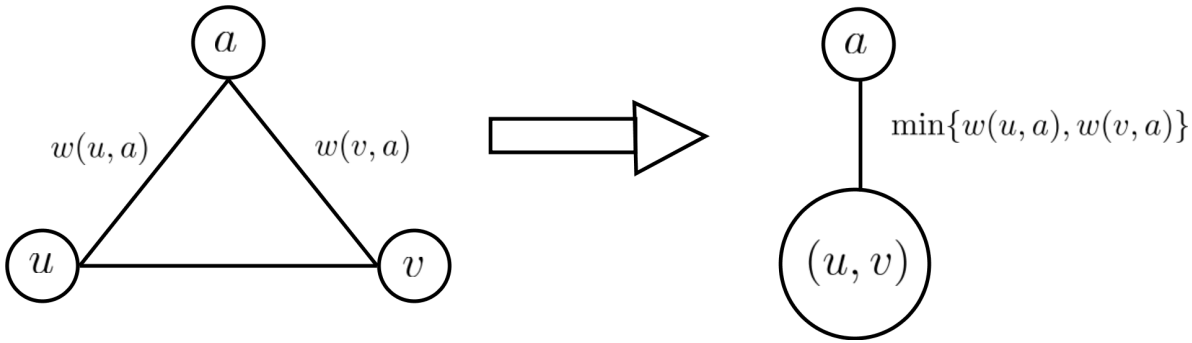
- Let T denote the minimum spanning tree to the graph G.
- Let an edge $e = (u, v)$ be in T.
- Then, contracting e into a single node uv results in a new tree T'.

Then, T' is the minimum spanning tree on $G \setminus e$.



MST: Dynamic Programing

Contraction of an edge e means:



$\text{MST-DP}(G, w)$

1 $T = \emptyset$
2 Guess edge e in MST
3 Contract e
4 $T' = \text{MST-DP}(G \setminus e, w)$
5 Decontract e
6 $T = T' \cup e$

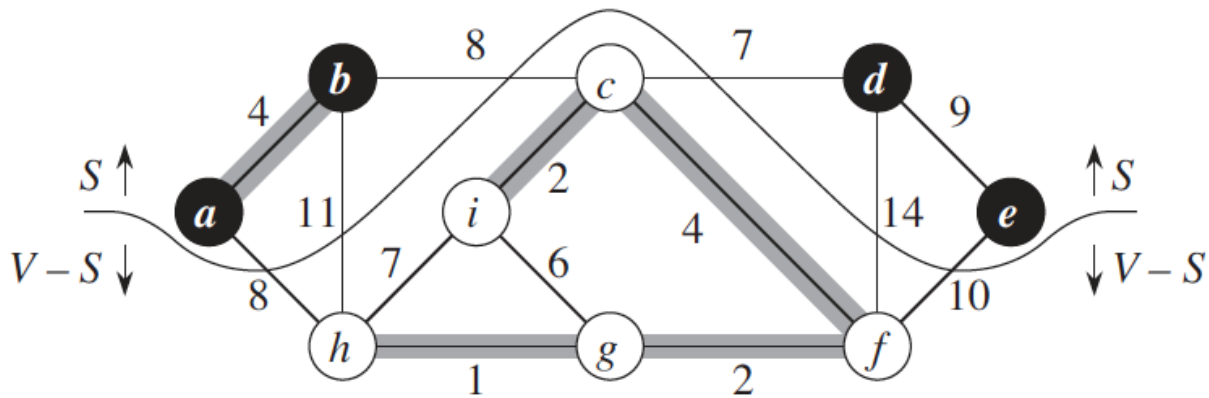
Run-time: Depends on the edge guesses, i.e. $O(2^{|E|})$.

MST: Greedy Approach

MST-GENERIC(G, w)

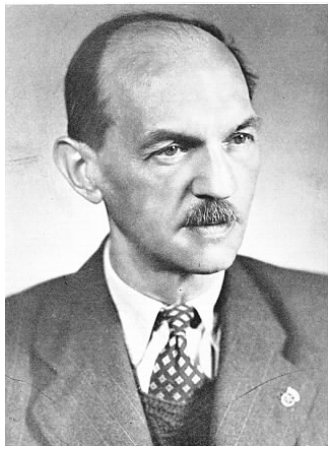
1 $A = \emptyset$
2 **while** A does not form a spanning tree
3 Find an edge (u, v) that is safe for A.
4 $A = A \cup \{(u, v)\}$
5 **return** A

Example: For this *cut*, a *safe* edge could be (c, d), since it has the minimum weight.



MST: History of Greedy Solutions

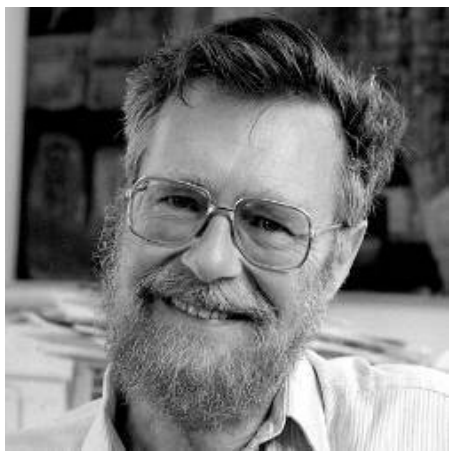
Boruvka



Jarnìk

Kruskal

Prim



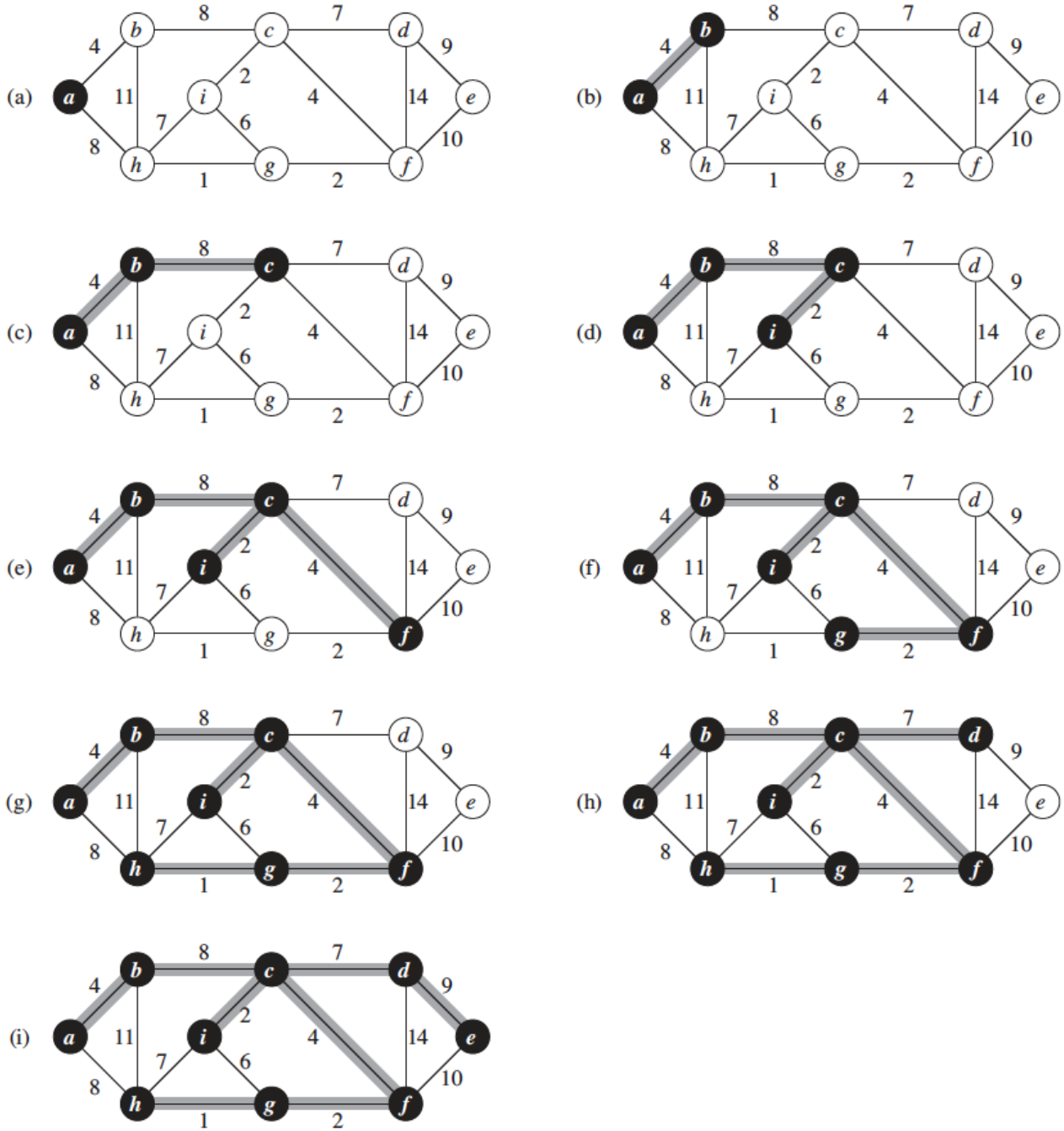
Dijkstra

Prim's Algorithm

```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V
 2      u.key = ∞
 3      u.parent = ∅
 4  r.key = 0
 5  Q = G.V // Implement Q as a min-priority queue using min-heaps.
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.parent = u
11              v.key = w(u, v)
```

Run-time Analysis: $O(|E| \log |V|)$

- Each $\textsc{Extract-Min}(Q)$ call takes $O(\log |V|)$ time – run $|V|$ times.
- The inside for-loop takes $O(|E|)$ iterations.
- Finding the min. weight key takes $O(\log |V|)$ steps.


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
1
2

Kruskal's Algorithm - Prerequisites

Idea: Instead of starting from one node, let us build MST dynamically from a forest comprising a subtrees in MST.

Safe edge: Identify the min. weight edge that does not form a cycle.

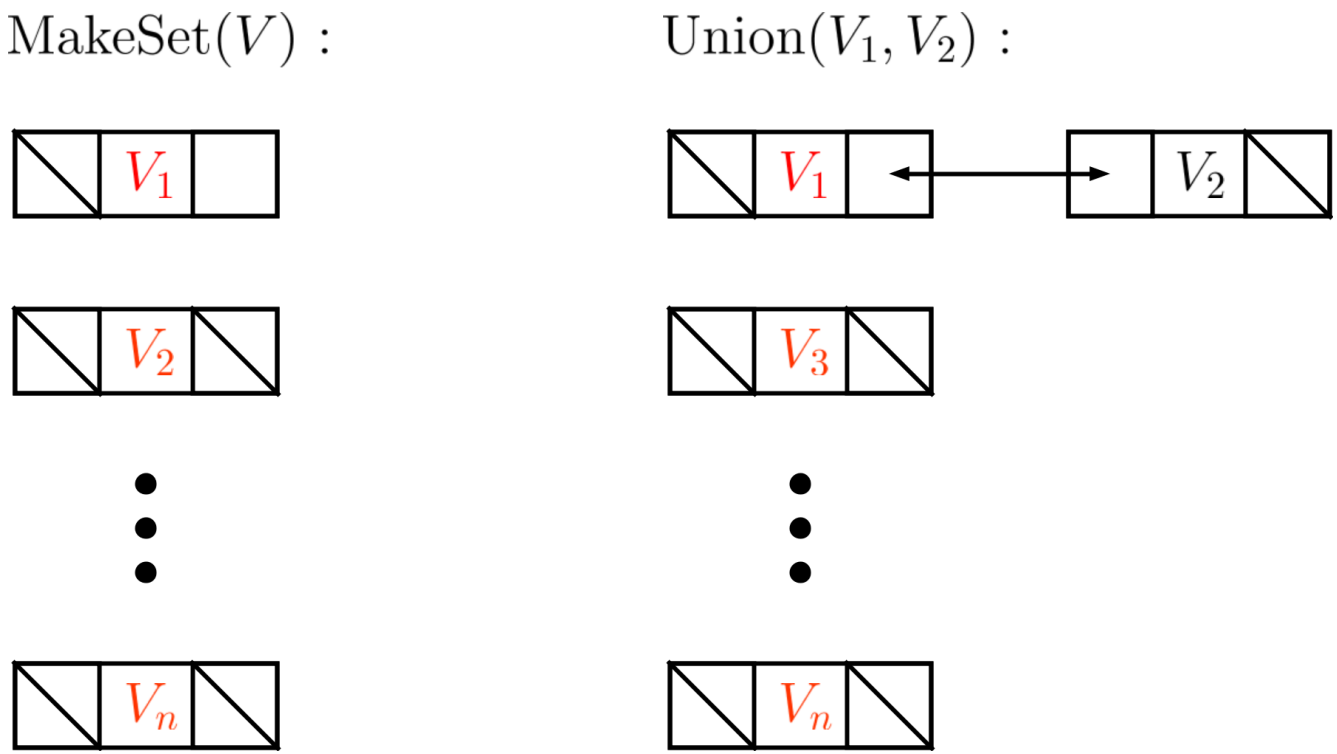
Disjoint-Set data structure:

- Maintains a collection of disjoint dynamic sets $S = \{S_1, \cdots, S_k\}$.
- Identify each set by a *representative* member x in that set.
- Operations allowed:
  - $\textsc{MakeSet}(S)$: Create a new set for each $x \in S$
  - $\textsc{Union}(x, y)$: Returns $S_x \cup S_y$
  - $\textsc{FindSet}(x)$: Returns the representative element of the set containing x

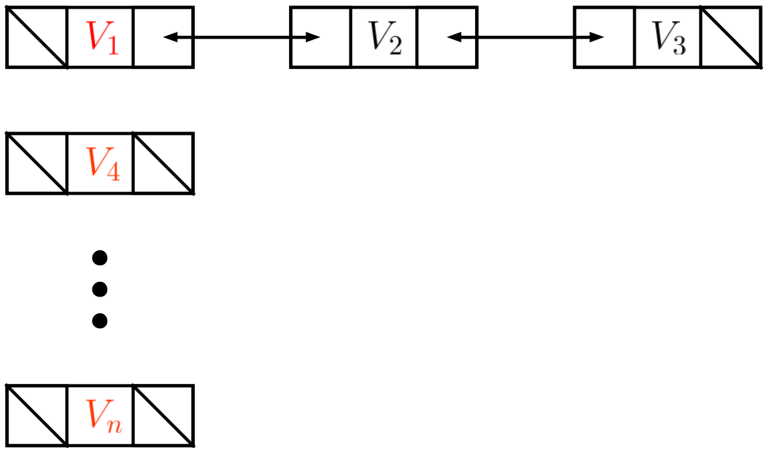
Simple Implementation of Disjoint-Set Data Structure

Idea: Implement each disjoint set as a doubly-linked list with the left-head as its representative.

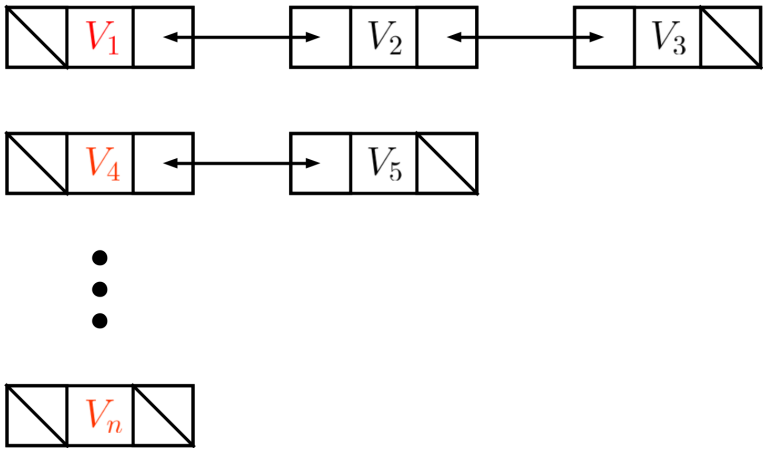
- $\textsc{FindSet}(x)$ always returns the representative of the set in which x lies.



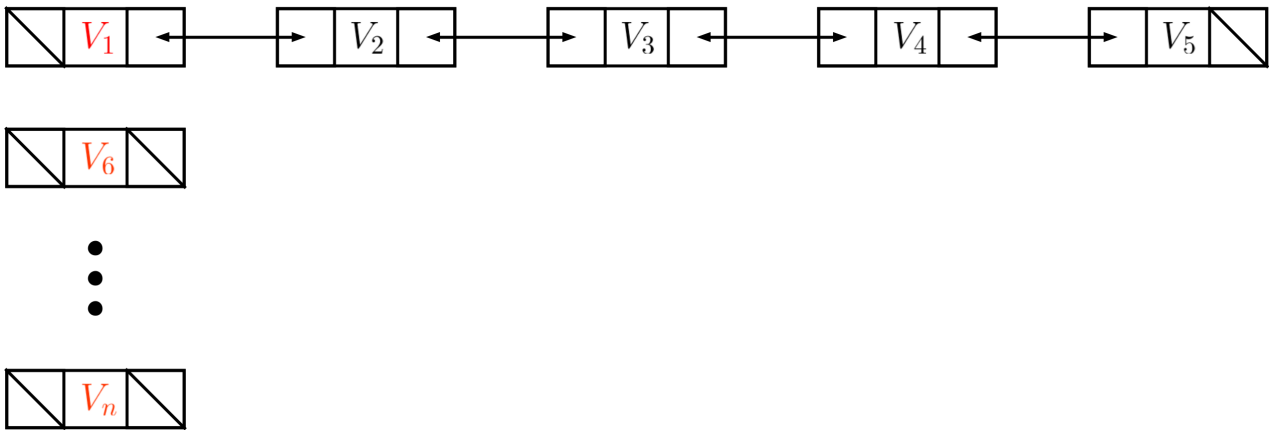
Union(V_1, V_3) or Union(V_2, V_3) :



Union(V_4, V_5) :



Union(V_2, V_5) :



Note: If $\textsc{FindSet}(u) = \textsc{FindSet}(v)$ for any two vertices, then we have a cycle.

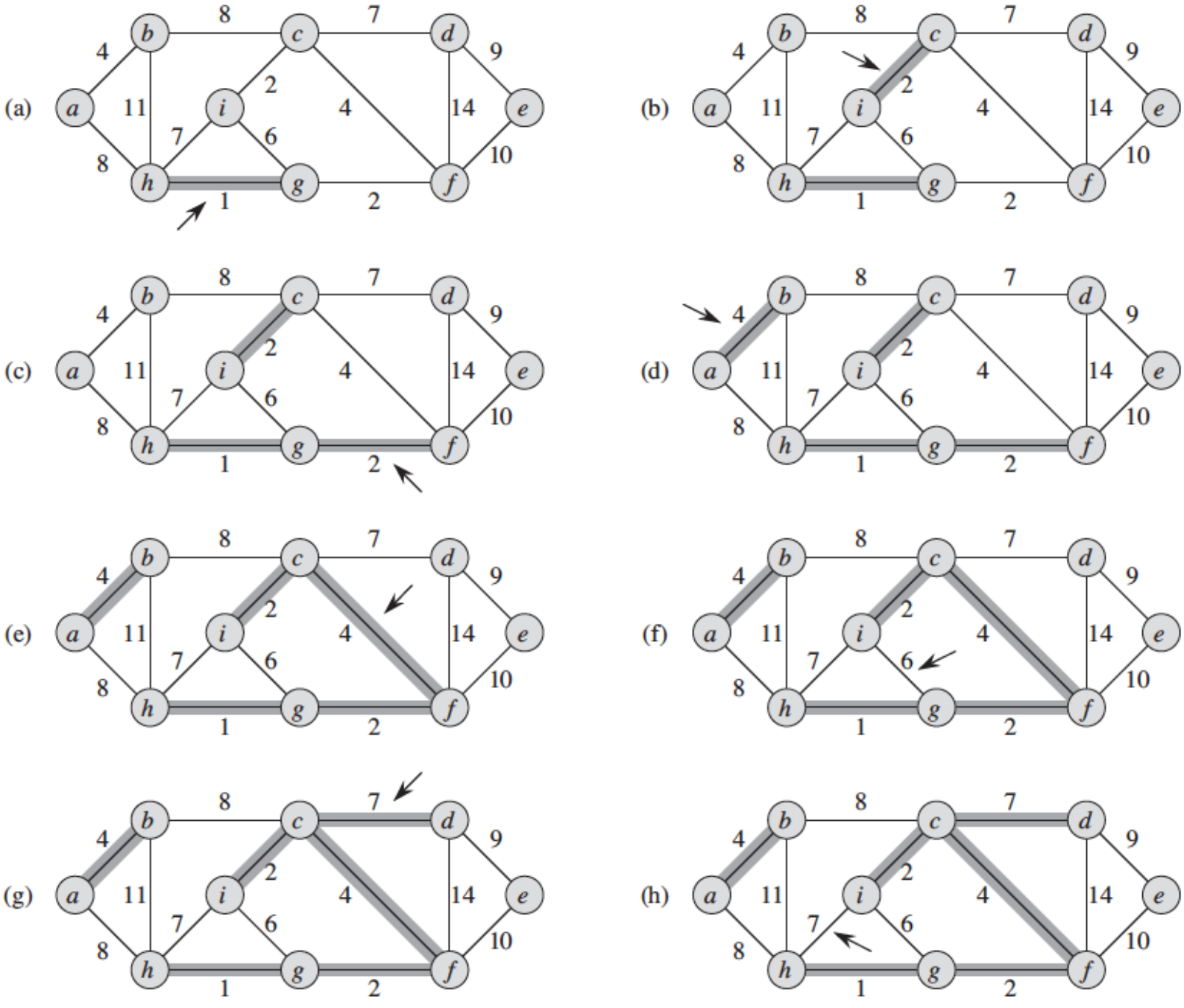
Kruskal's Algorithm

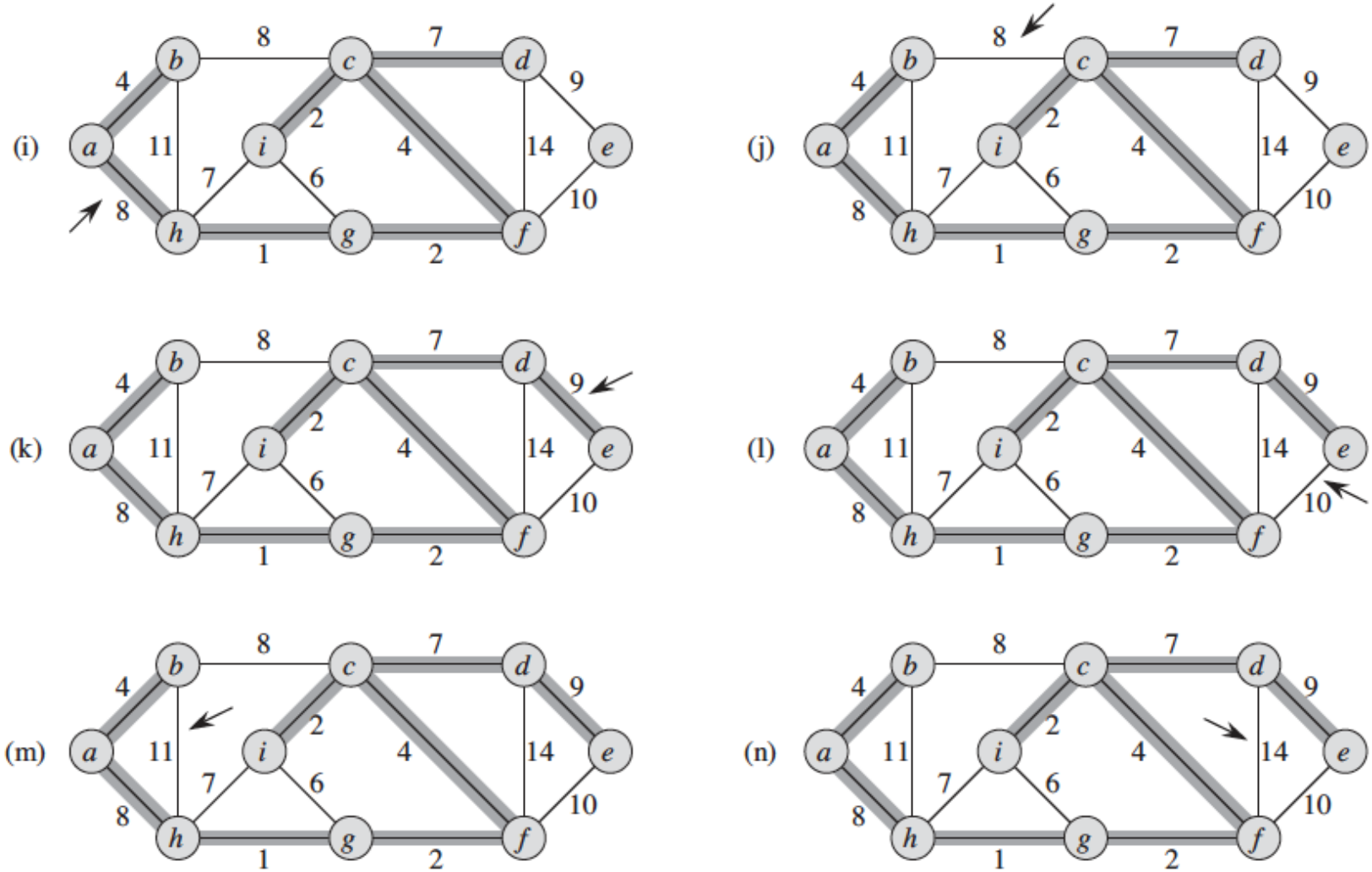
MST-KRUSKAL(G, w)

```
1  A = ∅
2  for each vertex v ∈ G.V
3        MAKESET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each (u, v) ∈ G.E, taken in nondecreasing order by weight
6        if FINDSET(u) ≠ FINDSET(v)
7              A = A ∪ (u, v)
8              UNION(u, v)
9  return A
```

Run-time Analysis: $O(|E| \log |V| + |E||V|)$

- Sorting edges – $O(|E| \log |E|)$
- If G is connected, $|V|^2 \geq |E| \geq |V| - 1$.
- Number of FIND-SET and UNION operations: $O(|E|)$
- Each operation takes $\alpha(|V|) = O(|V|)$ steps.
- All the MAKE-SET operations together: $O(|V|)$


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
a
b
c
d
e
f
g
h
i
4
8
11
7
8
2
4
14
9
6
10
1
2



Note: Run-time of this algorithm can be improved to $O(|E| \log |V|)$ by implementing Disjoint-Set data structure differently (Union-by-rank and path compression).

Single-Source Shortest Path

Problem Statement:
Given a weighted directed graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$, find a shortest path between a source vertex s and all the other vertices $v \in V$ with

$$\delta(s, v) = \begin{cases} \underset{p \in \mathcal{P}(s,v)}{\text{minimize}}\ w(p), & \text{if } \mathcal{P}(s, v) \neq \emptyset \\ \infty, & \text{otherwise.} \end{cases}$$

where

- $w(p) = \sum_{e \in p} w(e)$ is the sum of all edge weights in a path p, and
- $\mathcal{P}(s, v)$ is the set of all possible paths from s to v in the graph G.

Optimal Substructure in Shortest Path

Lemma

Given a weighted directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $p = (v_0, v_1, \cdots, v_k)$ denote the shortest path from vertex v_0 to v_k. Furthermore, for any i, j such that $0 \leq i \leq j \leq k$, let $p_{ij} = (v_i, \cdots, v_j)$ denote any subpath in p between the vertices v_i and v_j. Then, p_{ij} is the shortest path from v_i to v_j.

Proof.

Suppose p can be decomposed as $v_0 \overset{p_{0.i}}{\rightsquigarrow} v_i \overset{p_{i,j}}{\rightsquigarrow} v_j \overset{p_{j,k}}{\rightsquigarrow} v_k$. This means, $w(p) = w(p_{0,i}) + w(p_{i,j}) + w(p_{j,k})$.

If $p'_{i,j}$ is the shortest path between v_i and v_j, then we have $w(p'_{i,j}) < w(p_{i,j})$. Therefore, the weight of the path $v_0 \overset{p_{0.i}}{\rightsquigarrow} v_i \overset{p'_{i,j}}{\rightsquigarrow} v_j \overset{p_{j,k}}{\rightsquigarrow} v_k$ is $w(p_{0,i}) + w(p'_{i,j}) + w(p_{j,k}) < w(p)$. This is a contradiction to our assumption that p is the shortest path. □

Some Degenerate Cases

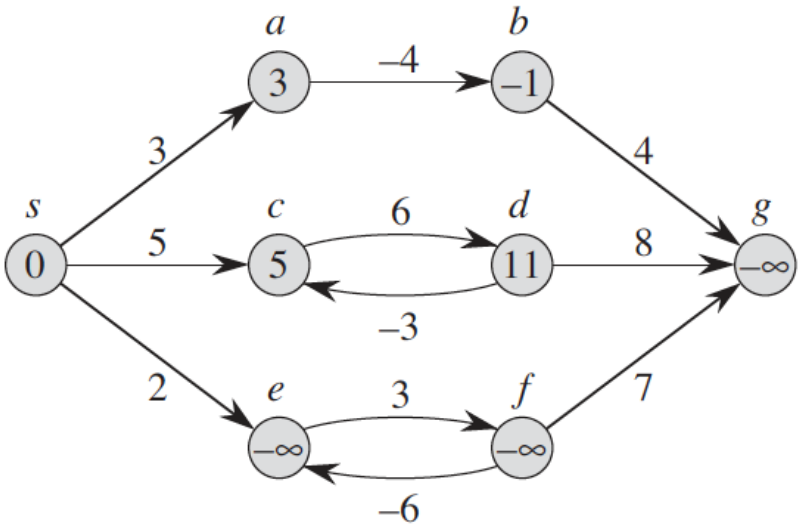
- If a path between vertices u and v has a negative-weight cycle, then

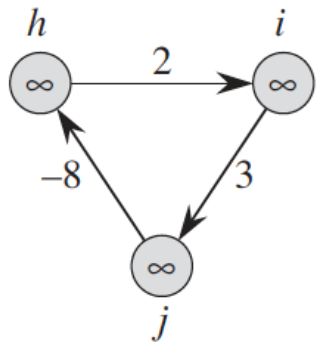
$$\delta(u, v) = \underset{p \in \mathcal{P}(u,v)}{\text{minimize}}\ w(p) = -\infty.$$

  Note: Shortest path does not have any cycle $\Rightarrow$ *simple path*.

- If there is no path between vertices u and v, then

$$\delta(u, v) = \underset{p \in \mathcal{P}(u,v)}{\text{minimize}}\ w(p) = \infty.$$





Length of a Simple Path

Lemma

Consider a directed graph $G = (V, E)$ with weights $w : E \to \mathbb{R}$, but without negative weight cycles. Then, if a node v is reachable from s, the length (no. of hops) of the path is less than $|V|$.

Proof.

- Assume there is a shortest path of length $\geq |V|$ between s and v_k.
- If $k \geq |V|$, then a cycle exists.
- Positive weight cycle $\Rightarrow$ not the shortest path.
- Negative weight cycle $\Rightarrow$ violates our assumption.

We have a contradiction! $\square$

Classic Bellman-Ford Algorithm

- Solution Approach: Dynamic programming
- Let $M[k, v]$ be weight of shortest path between s and v, of length atmost k.

BELLMAN-FORD-CLASSIC(G, w, s)

```
1  // M[k,v] = min{M[k-1,v], min{M[k-1,u] + w(u,v) s.t. (u,v) ∈ E}}
2  for v ∈ G.V
3        M[0,v] = ∞
4        v.parent = ∅
5  M[0,s] = 0
6  for k = 1 to |G.V|
7        for v ∈ G.V
8        M[k,v] = M[k-1,v]
9        for each edge (u,v) ∈ G.E
10             if M[k-1,u] + w(u,v) < M[k,v]
11                  M[k,v] = M[k-1,u] + w(u,v)
12                  v.parent = u
13       if k == n & M[k,v] ≠ M[k-1,v]
14            return "G has a negative-weight cycle"
15 return M, parent
```

Edge Relaxation

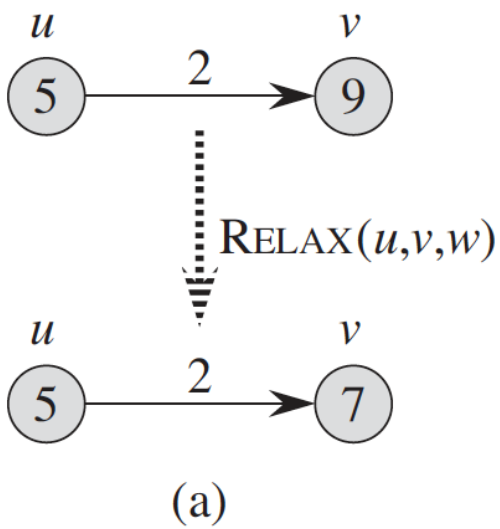
- Let $v.d$ denote the estimate of shortest path's weight.
- Update this estimate everytime a shorter path is found.
- Repeat this until no estimate can be updated with any edge.

INITIALIZE-SINGLE-SOURCE(G, s)

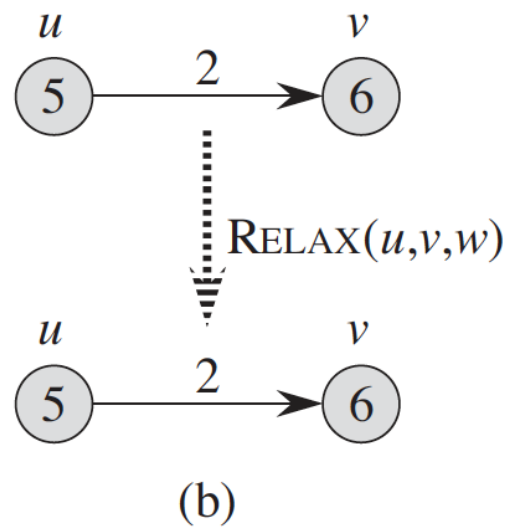
```
1  for each vertex v ∈ G.V
2        v.d = ∞
3        v.parent = ∅
4  s.d = 0
```

RELAX(u, v, w)

```
1  if v.d > u.d + w(u, v)
2        v.d = u.d + w(u, v)
3        v.parent = u
```



(a)



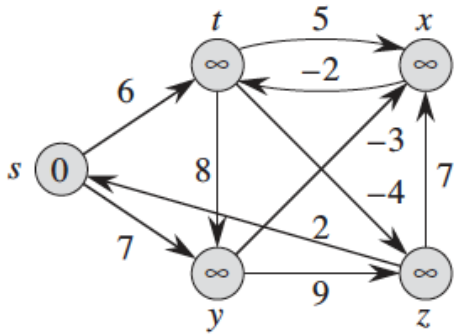
(b)

Bellman-Ford Algorithm with Edge Relaxation

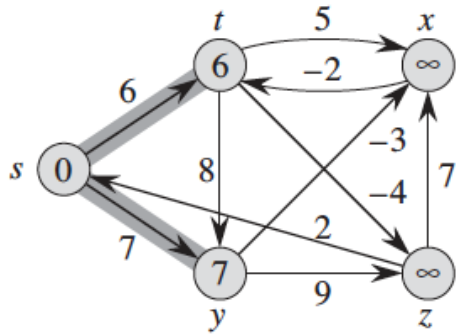
```
BELLMAN-FORD-RELAX(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3        for each edge (u, v) ∈ G.E
4              RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6        if v.d > u.d + w(u, v)
7              return FALSE
8  return TRUE
```

- Run-time: $O(|V| \cdot |E|)$

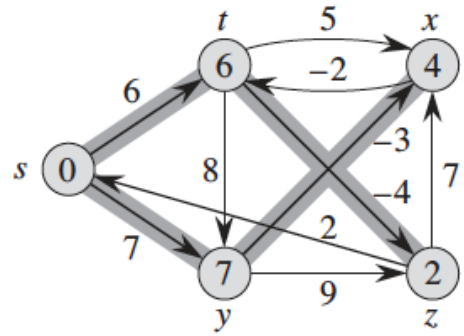
Example



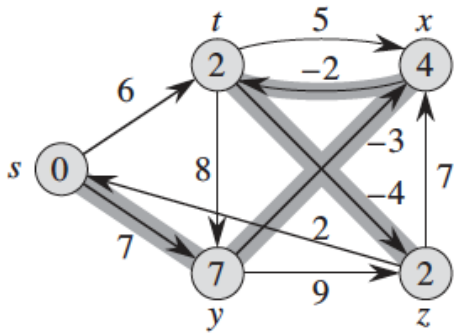
(a)



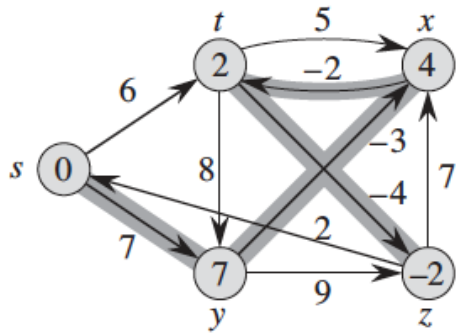
(b)



(c)



(d)



(e)

Convergence to Shortest Path

Lemma

Let $G = (V, E)$ be a weighted, directed graph with source s and weight function $w : E \to \mathbb{R}$, and assume G contains no negative-weight cycles that are reachable from s. Then, after $|V| - 1$ iterations of edge relaxations, BELLMAN-FORD returns $v.d = \delta(s, v)$ for all $v \in V$.

Sketch of Proof: Proof by Induction.
Let $p = \{s, v_1, \cdots, v_k\}$ denote the shortest path between s and v_k. Since shortest paths are simple, $k \leq |V| - 1$. In the worst-case ordering of edges,

- *First iteration:* Returns $v_1.d = \delta(s, v_1)$

  $\vdots$

- i^{th} *iteration:* Returns $v_i.d = \delta(s, v_i)$.

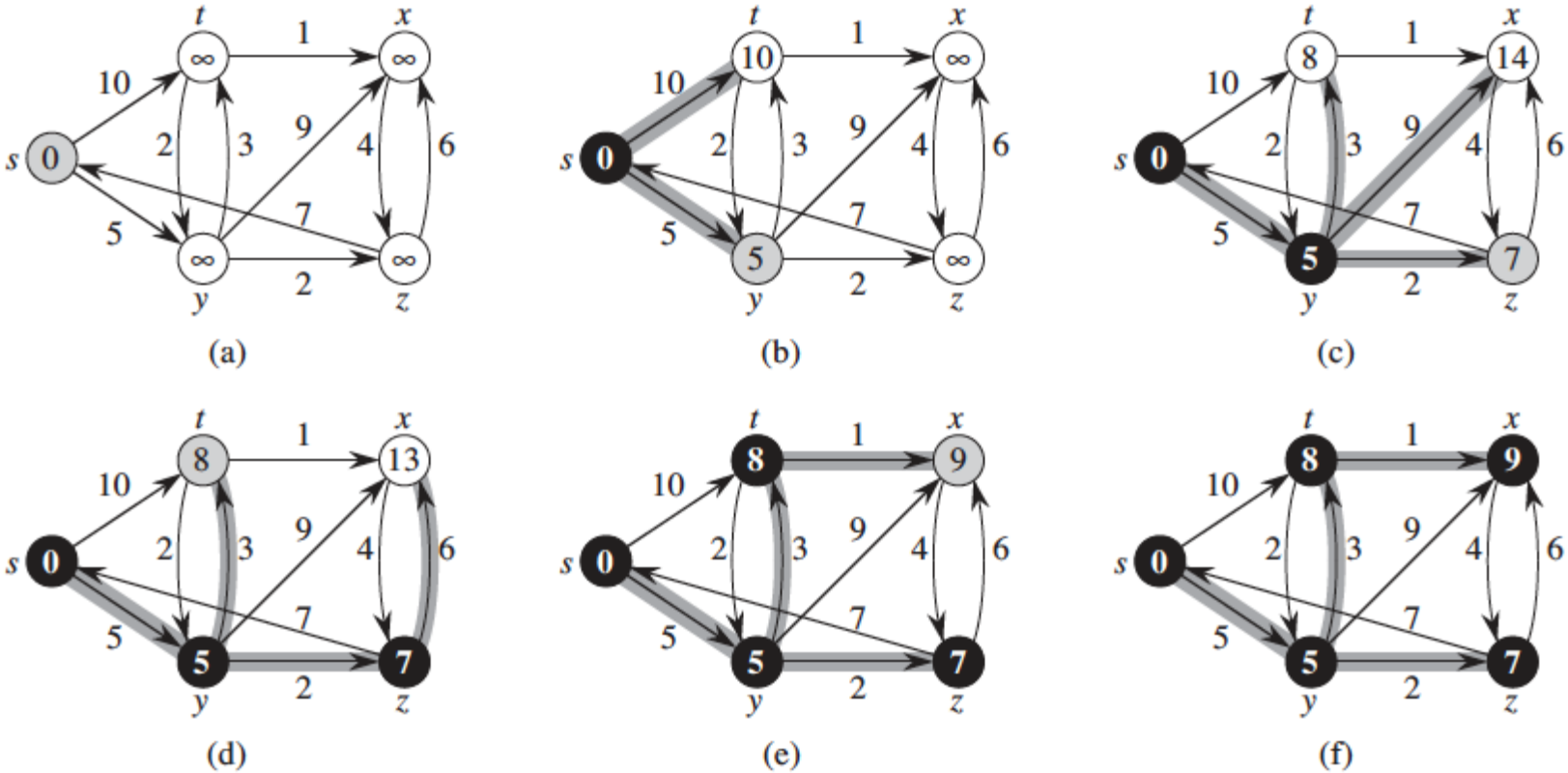
Dijkstra's Algorithm

- Solution Approach: Greedy Algorithm
- Produces an optimal solution when all the graph edges have non-negative weights.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5        u = EXTRACT-MIN(Q)
6        S = S ∪ {u}
7        for each vertex v ∈ G.Adj[u]
8              RELAX(u, v, w)
```

- Run-time: $O(|V| \log |V| + |E| \log |V|)$, assuming the priority queue is implemented using min-heaps.
- Can be further improved to $O(|V| \log |V| + |E|)$ using Fibonacci heaps.

Dijkstra's Algorithm (cont.)

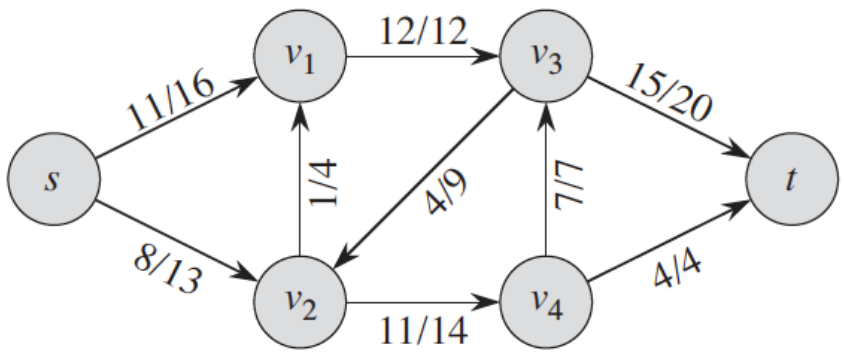


Flow Networks

- Directed graph $G = (V, E)$ with a non-negative capacity $c(u, v),\ \forall\ (u, v) \in E$.
- **Goal:** Maximize flow from source s to sink t.
- Flow $f : V \times V \to \mathbb{R}$ such that
  - **Capacity constraint:** $0 \leq f(u, v) \leq c(u, v)$, for all $(u, v) \in E$.
  - **Flow conversation:** For all $u \in V - \{s, t\}$,

$$\sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u).$$

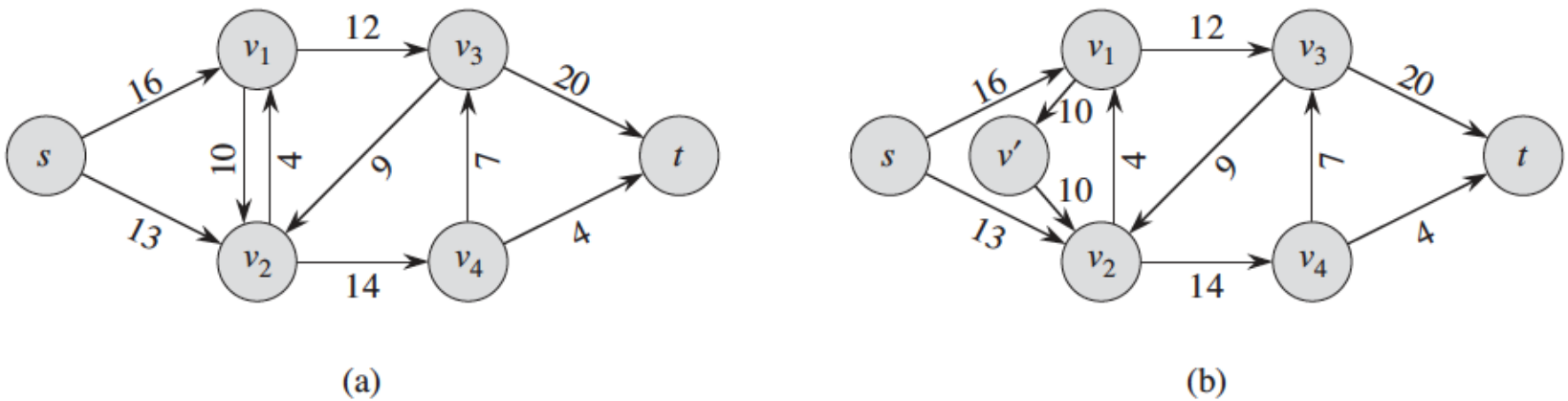
- Edge labels: $f(u, v)/c(u, v)$ for each edge $(u, v) \in E$.



Flow Networks: Assumptions

Assumptions:

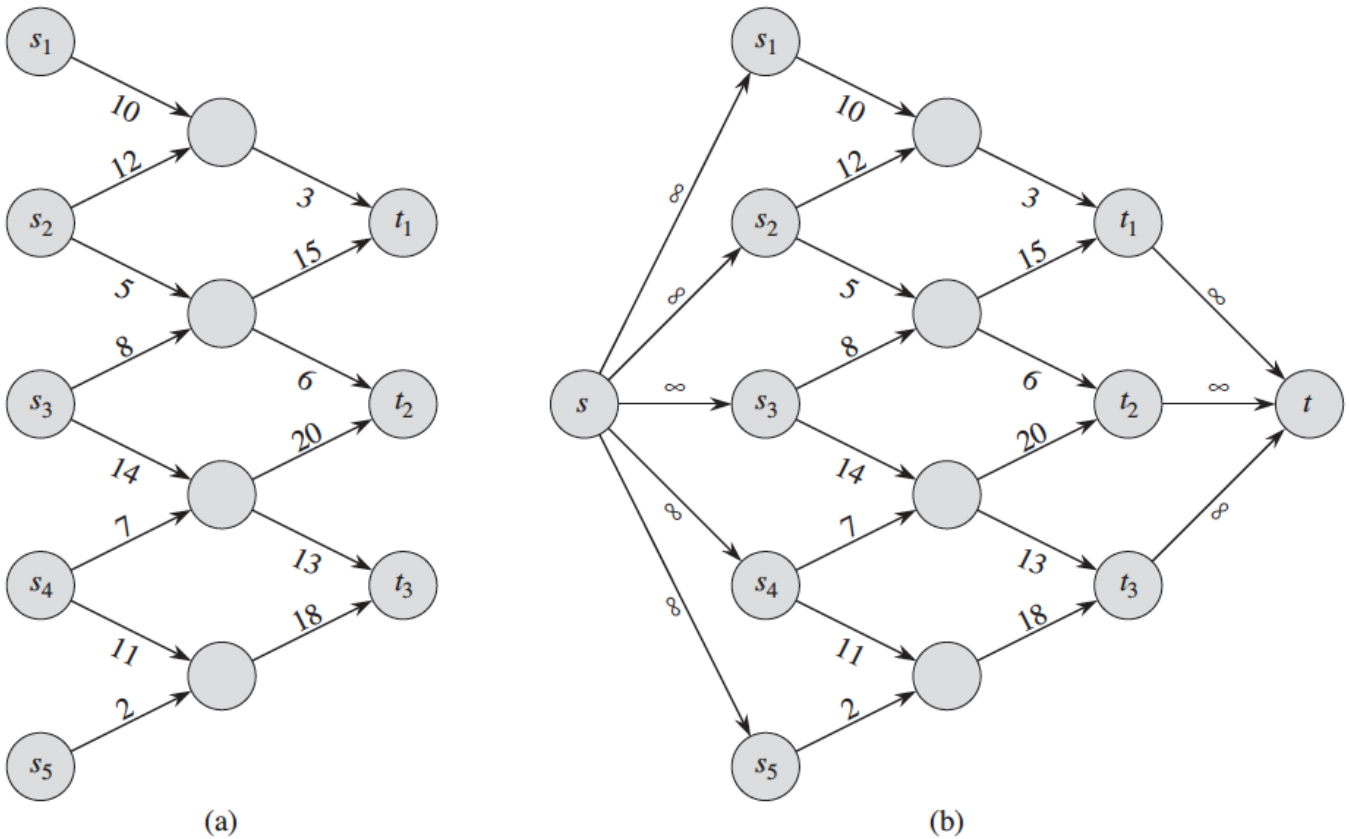
- $c(u, v) = 0, \ \forall \ (u, v) \notin E$.
- If $(u, v) \in E$, $(v, u) \notin E$.
- No self-loops, i.e. $(v, v) \notin E$ for all $v \in V$.
- Replace antiparallel edges by adding a new vertex v'.



Total value of the flow: $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$

Flow Networks: Multiple Sources and Sinks

If there are multiple sources and sinks, create a new *super-source* and *super-sink* node connecting all the sources and sinks respectively with infinite capacity edges.



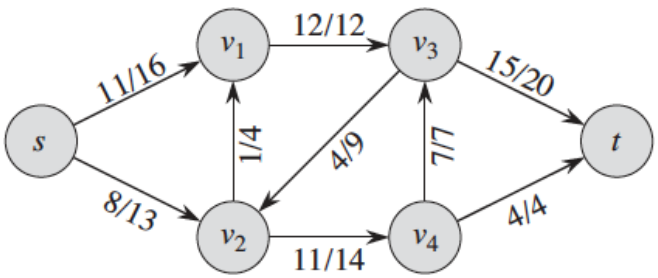
Residual Graphs and Augmenting Flows

- Residual Graph G_f: Replace every edge with two edges such that

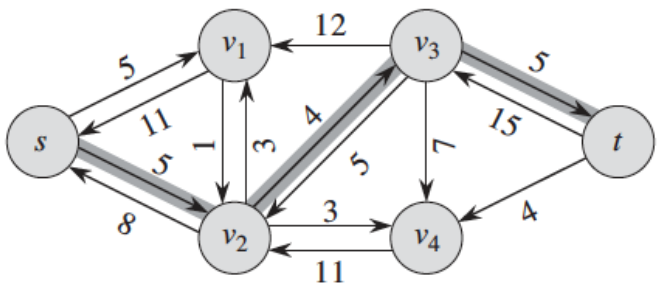
$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v), & \text{if } (u,v) \in E, \\ f(v,u), & \text{if } (v,u) \in E, \\ 0, & \text{otherwise} \end{cases}.$$

- Augmenting flow: If there is a flow f' in G_f, then

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) - f'(u,v) - f'(v,u), & \text{if } (u,v) \in G_f, \\ 0, & \text{otherwise} \end{cases}.$$



(a)



(b)

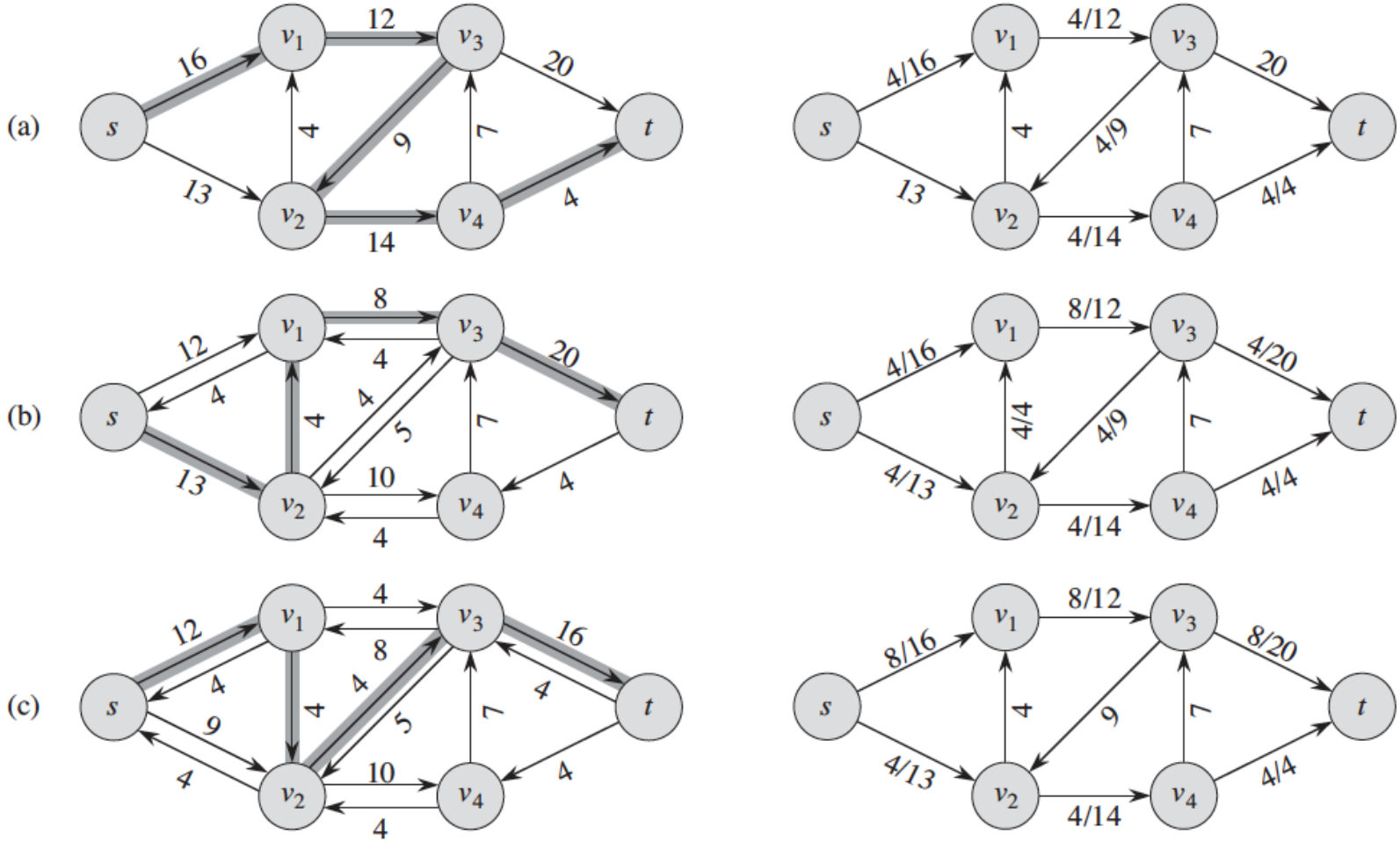
Ford-Fulkerson Algorithm

FORD-FULKERSON(G, s, t)

1 **for** each edge $(u, v) \in G.E$
2 $\quad (u, v).f = 0$
3 **while** there exists a path p from s to t in the residual graph G_f
4 $\quad c_f(p) = \min\{c_f(u, v) | (u, v)$ is in $p\}$
5 $\quad$ **for** each edge (u, v) in p
6 $\quad\quad$ **if** $(u, v) \in E$
7 $\quad\quad\quad (u, v).f = (u, v).f + c_f(p)$
8 $\quad\quad$ **else** $(v, u).f = (v, u).f - c_f(p)$

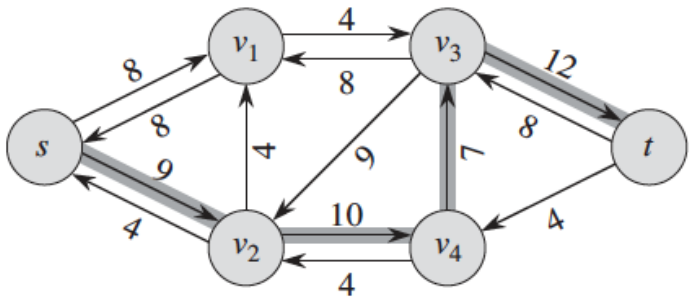
- Depends on how we find augmenting paths.
- Can find a path in $O(|E|)$ time using any search algorithm.
- If edge capacities are integers, can increase flow by one unit in every augmenting path.
  - Run-time: $O(|E| \cdot f^*)$, where f^* is the maximum flow.
- Irrational capacities $\Rightarrow$ does not converge to max. flow.

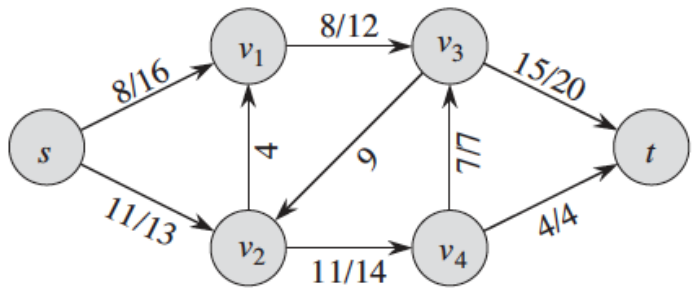
Ford-Fulkerson Algorithm (cont.)



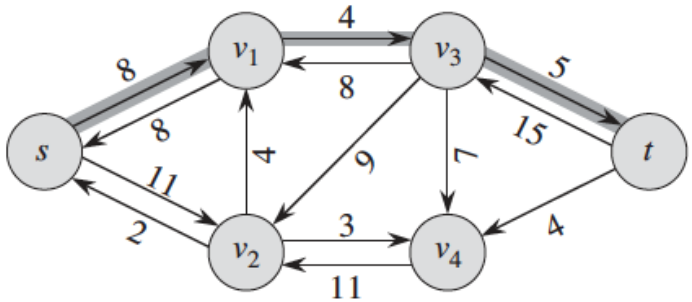
Ford-Fulkerson Algorithm (cont.)

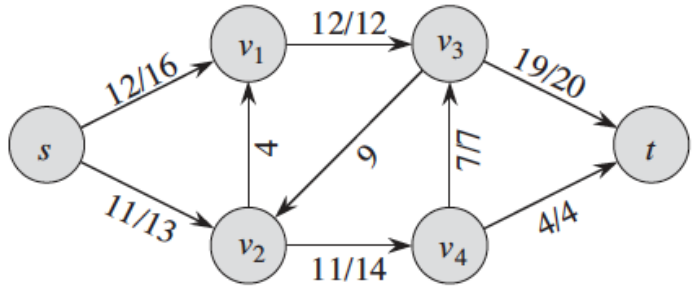
(d)



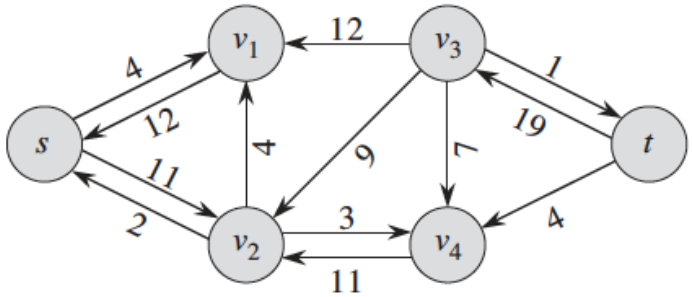


(e)





(f)



Edmond-Karp Algorithm

Can we improve the run-time by finding better augmenting paths at each iteration?

Find a shortest length path from s to t using **BFS** algorithm in each G_f, assuming unit weights on each edge.

- $O(|V||E|)$ flow augmentations to obtain max. flow.
- Run-time: $O(|V||E|^2)$