COMP 8547
Advanced Computing Concepts

Course instructor: Dr. Olena Syrotkina

Winter 2024: 06 – Graph algorithms

Chapter 6 – Graph algorithms

Contents

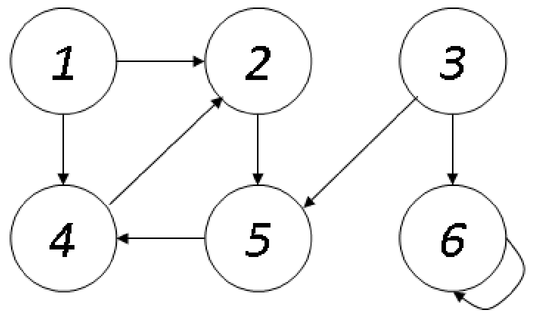
Graphs

- **Definition:** A *graph* G is a pair (V, E), where
  - V = {v_1, v_2, …,v_m} is a set of vertices
  - E $\subseteq$ V $\times$ V is a binary relation on V
- **Directed graph:**
  - E contains ordered pairs, i.e., pair (u,v) $\neq$ (v,u)
- **Undirected graph:**
  - E contains unordered pairs, i.e., each pair (u,v) is a set {u,v}

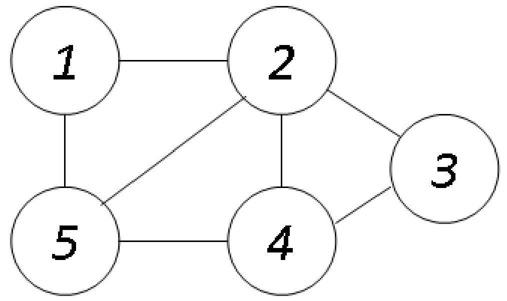
Definitions:

- **Path:** a sequence of vertices
- **Simple path:** all vertices are different
- **Cycle:** first and last vertices in path are equal
- **Acyclic graph:** It has no cycles
- **DAG:** Directed acyclic graph

Directed:



Undirected:

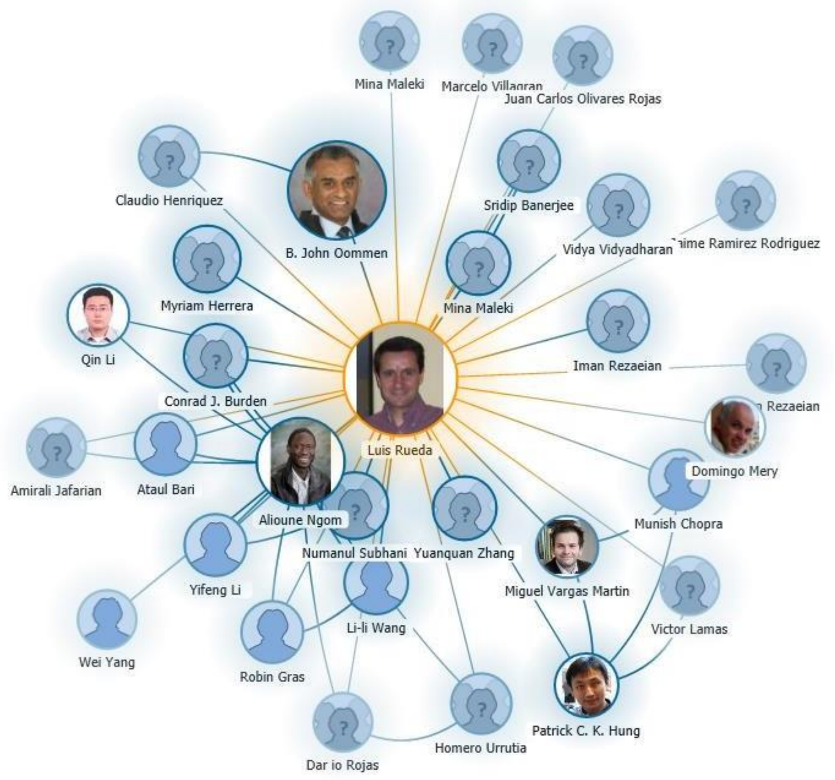


Graphs – sparseness + application examples

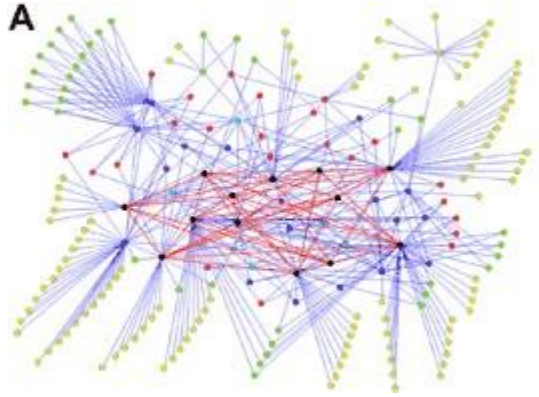
- Sparse graph:
  - G = (V, E) is *sparse,* if |E| <<< |V|^2
    Space used is O(|V|+|E|)

- Dense graph:
  - G = (V, E) is dense, if |E| ≅ |V|^2
    Space used is O(|V|+|E|)

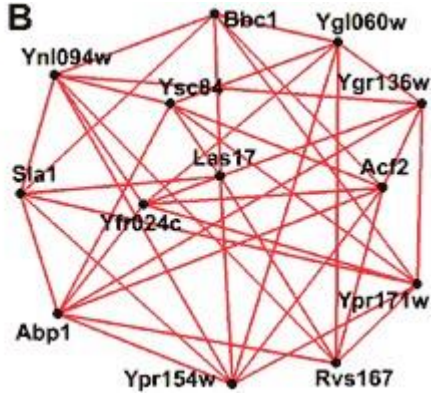
Sparse: Publication co-authorship network



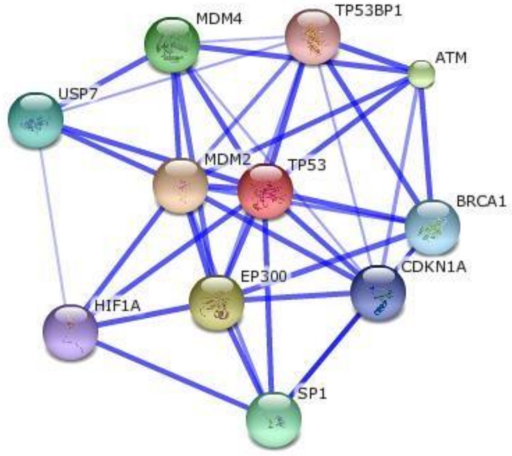
Sparse:



Dense:

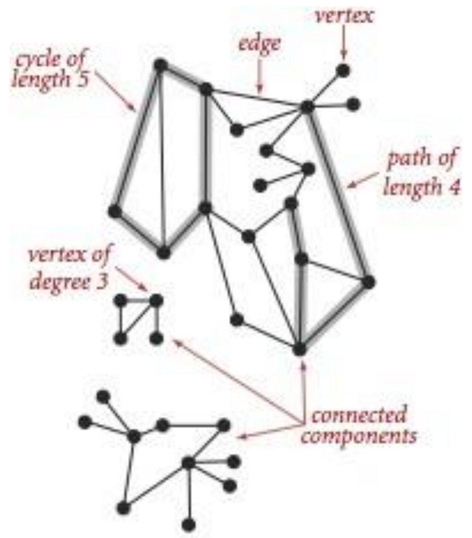


Dense: Protein interaction network

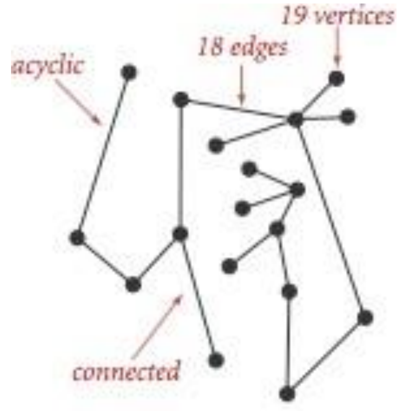


Graphs – more definitions, notation

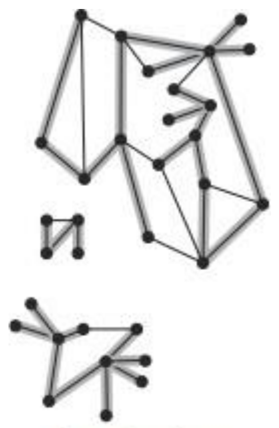
- Vertex w is adjacent to v if (v,w) $\in$ E
  - Notation: Adj[v]
- Tree: Undirected acyclic graph
- Forest: a disjoint set of trees
- Reachable: Vertex u is reachable from v if there is a path from v to u
- A vertex v is connected to another vertex u if there is a path from v to u
- A graph is connected if there is a path from every vertex to every other vertex
- Spanning tree: A subgraph that is a tree and contains all vertices
- Spanning forest: Spanning trees of an unconnected graph



Anatomy of a graph

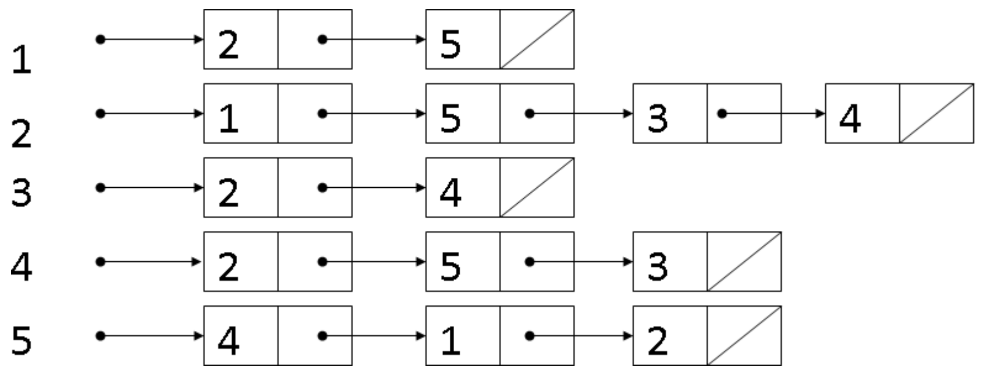


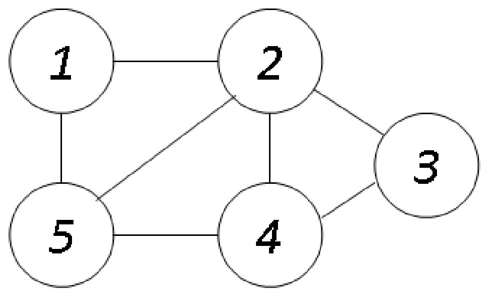
A tree


A spanning forest

Graphs - representations

- **Adjacency list:**
  - An array of |V| singly linked lists: Adj[u]
  - $\forall$ u $\in$ V, Adj[u] contains all v, s.t. u $\neq$ v, and (u,v) $\in$ E



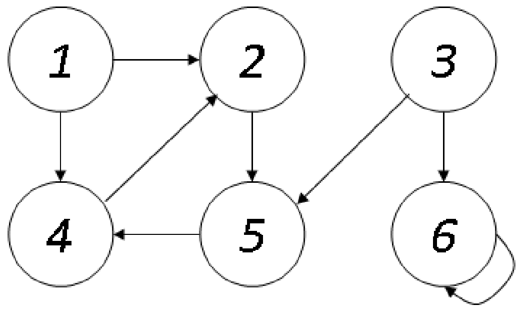


Space complexity: O(|E|+|V|)

- **Adjacency matrix:**
  - It is a |V| $\times$ |V| matrix A = {a_{ij}}, where

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

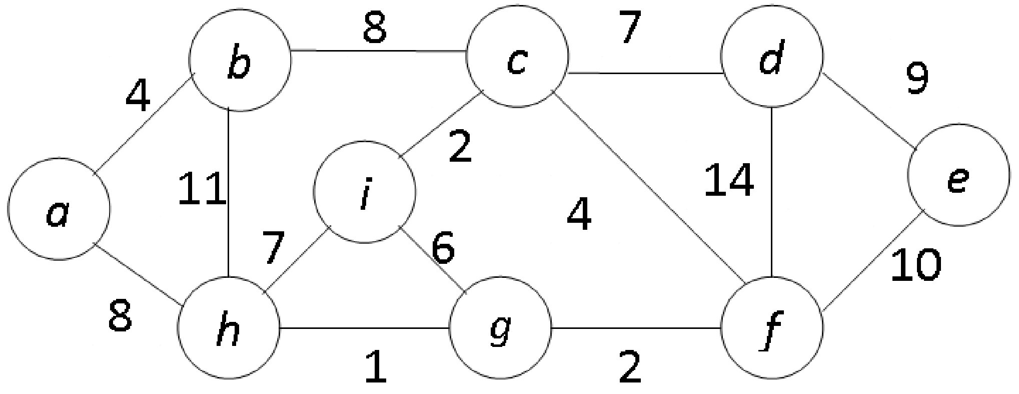
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

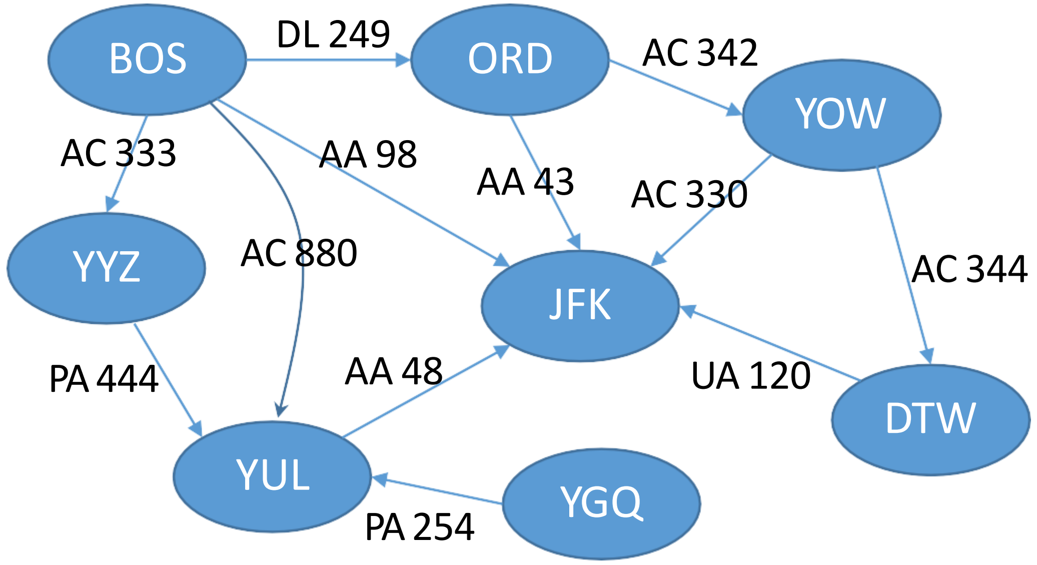


Space complexity: O(|V|^2)

Weighted graphs

- **Definition** (weighted graph):
  - A connected, undirected graph G = (V, E) is *weighted*, if
  - ∀ (u,v) ∈ E, ∃ a *weight*, given by function w(u,v)
- More general:
  - w(u,v) can be an arbitrary object
  - Example: flights from one airport to another.
  - Object may contain more info about flight (e.g., date, departure time, arrival time, etc.)





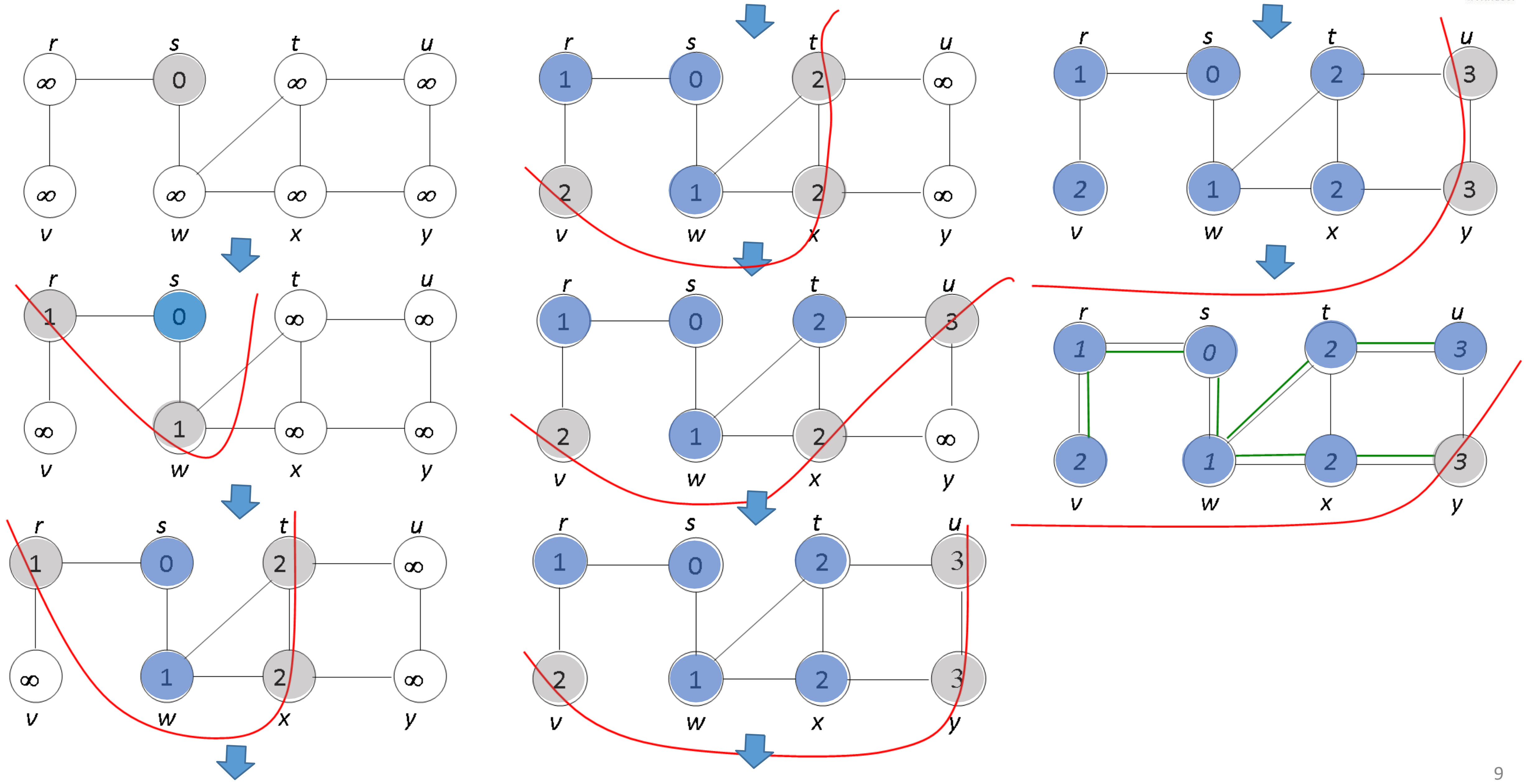
Graphs – traversal

Breadth-first search

- Given G = (V, E), and a *source* vertex, s,
- Breadth-first explores *every* vertex, v, reachable from s:
  - Computes distance from s to v
  - Produces a breadth-first tree whose root is s
- Path from s to v is the shortest one
- Works on both *directed* and *undirected* graphs
- Worst-case time of BFS is O(|V| +|E|)

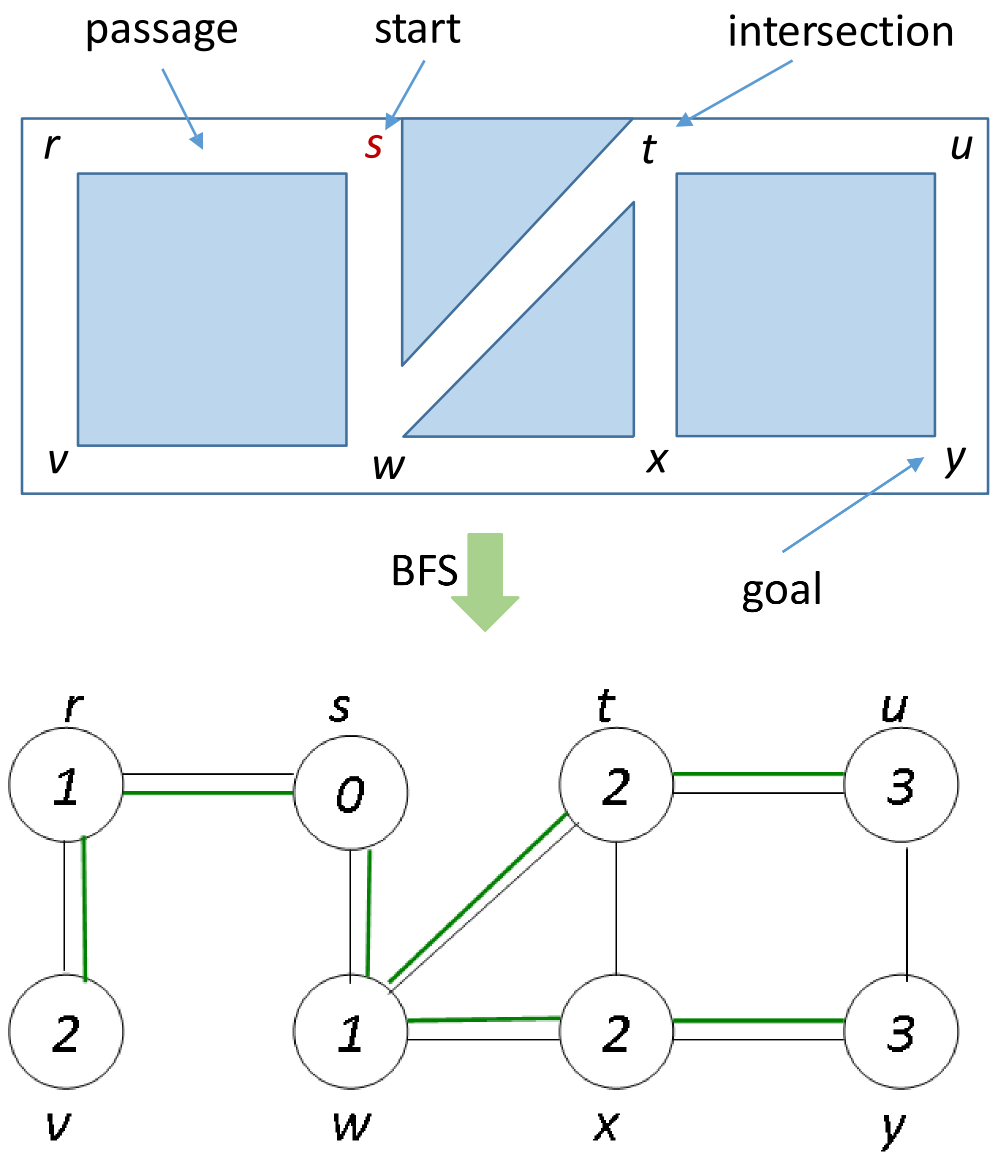
```
Algorithm BFS(G, s)
for each $u \in V - \{s\}$
    color[u] $\leftarrow$ "white"
    d[u] $\leftarrow \infty$;  $\pi$[u] $\leftarrow$ nil
color[s] $\leftarrow$ "gray"
d[s] $\leftarrow$ 0; $\pi$[s] $\leftarrow$ nil        // p[u] =  predecessor of u
Q $\leftarrow \varnothing$                  // Create an empty queue
enqueue s to Q
while Q $\neq \varnothing$
    u $\leftarrow$ dequeue from Q
    for each v $\in$ Adj[u]
        if color[v] ="white"
            color[v] $\leftarrow$ "gray"; d[v] $\leftarrow$ d[u] + 1; $\pi$[v] $\leftarrow$ u
            enqueue v to Q
    color[u] $\leftarrow$ "blue"
```

Breadth-first search - example



Application of BFS: Search in a maze

- Maze
  - Given a maze represented by a graph
  - Each passage is an edge in the graph
  - Each vertex is an intersection in the graph
- Problem
  - Given a starting point in the maze, *s*
  - Explore the maze using BFS
- BFS can be used to:
  - Explore the maze
  - Find a lost object or a goal in the maze
- BFS will avoid visiting any passage or intersection twice



Graphs – traversal

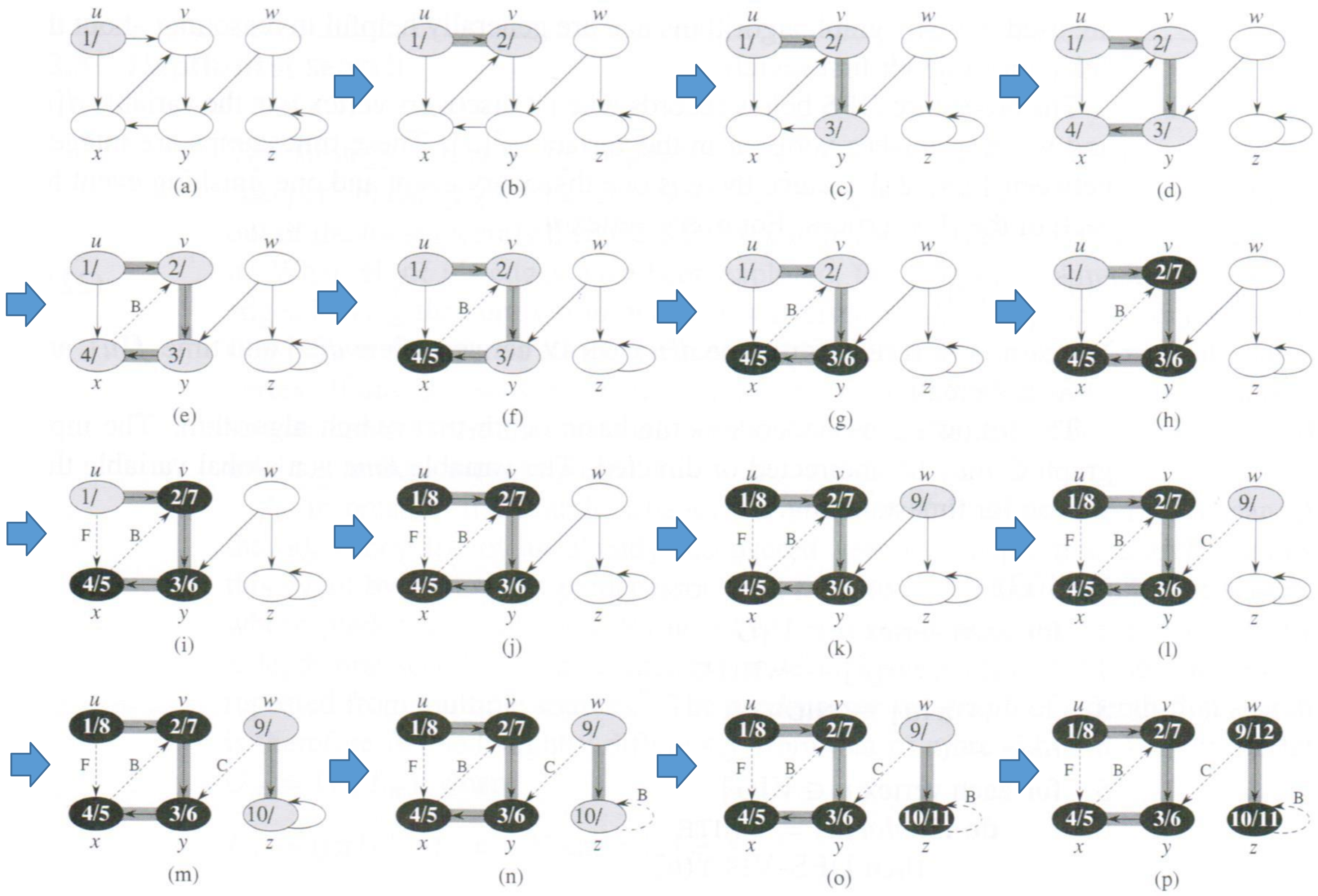
Depth-first search (DFS)

- Given G = (V, E), and a *source* vertex, s,
- Depth-first explores *every* vertex, v, reachable from s
- Simplest way to implement DFS is through recursion
- Unlike BFS, DFS goes "deep" first and then continues the search
- The recursion stack allows to go deep first
- Nonrecursive DFS uses a stack instead of a queue
- Works on both *directed* and *undirected* graphs
- Running time of DFS is O(|V|+|E|)
- DFS provides both preorder and postorder traversals of the graph

```
Algorithm DFS(G)
for each u ∈ V
    color[u] ← "white"
    p[u] ← nil
time ← 0
for each u ∈ V
    if color[u] ="white"
        DFS-Visit(u)
```

```
DFS-Visit(u)
color[u] ← "gray"
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]
    if color[v] ="white"
        p[v] ← u
        DFS-Visit(v)
color[u] ← "black"
f[u] ← time
time ← time + 1
```

Depth-first search – example: directed graph



Example not discussed in full detail –
found on page 542 of [5]

Shortest paths

- **Definition:**
  - A path of length $\delta(s, v)$ is a *shortest* path from s to v if it has the *minimum* number of edges.
  - No path from s to v $\Rightarrow \delta(s, v) = \infty$
- **Single-source shortest-path problem:**
  - Given:
    - A weighted, directed graph G = (V, E)
    - A weight function $w : E \to \mathbf{R}^+$
    - A source vertex $s \in V$
    - Weights are nonnegative
  - Aim: Find shortest path from s to *every* $v \in V, v \neq s$
- Dijkstra's algorithm is the most popular
- Other algorithms: Ch. 14 of [1]

Related problems:

- Single-destination shortest-path:
  - Given t (a destination), find a shortest path form every $v \in V$.
- Single-pair shortest-path:
  - Given $u, v \in V$, find shortest path from u to v.
- All-pairs shortest-path:
  - Given a weighted, directed graph G = (V, E), $\forall\ u, v \in V$, find a shortest-path from u to v.

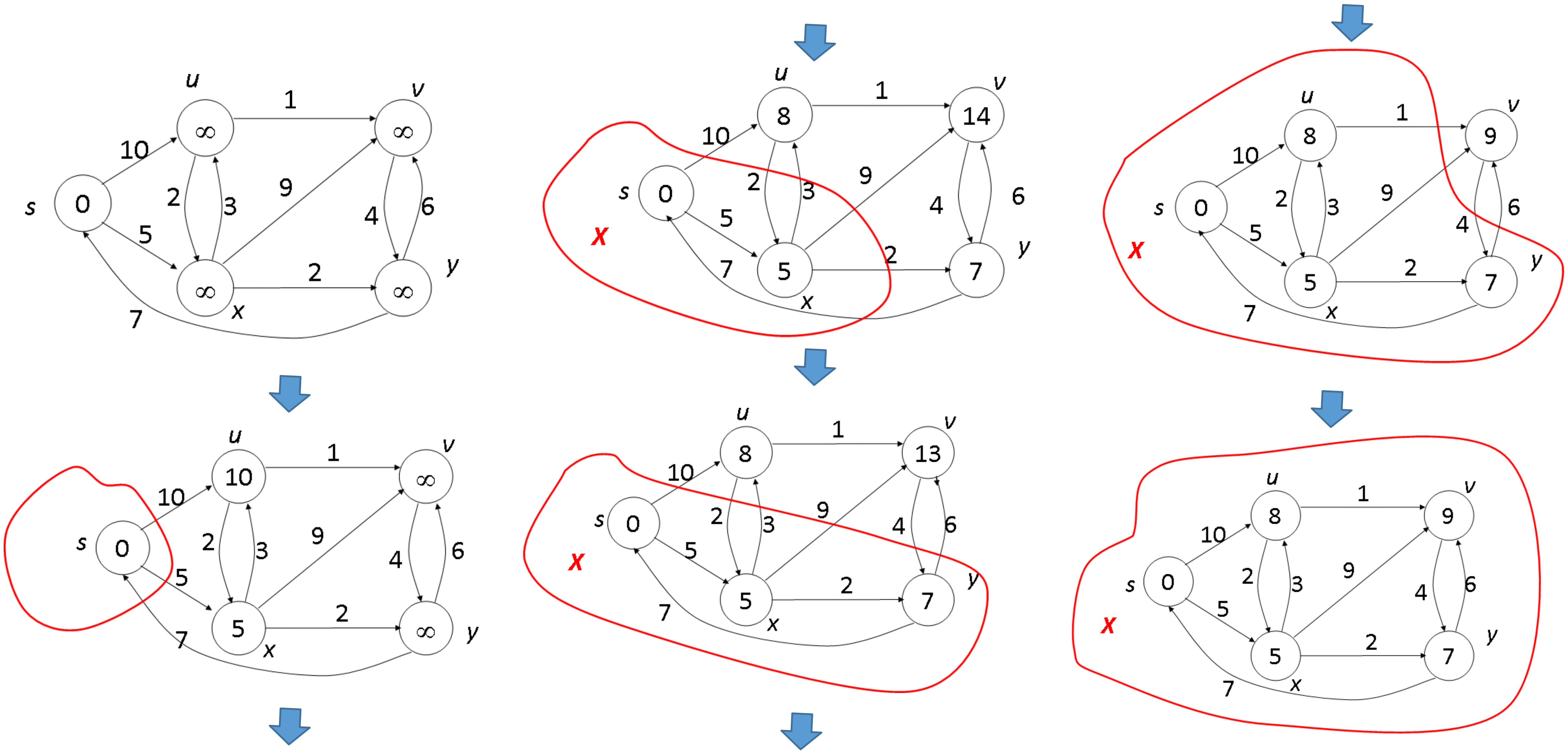
```
Algorithm SP-Dijkstra(G, w, s)
Initialize-Single-Source(G,s)
S ← ∅
Q ← V    // the queue is a heap
while Q ≠ ∅
   u ← Extract-Min(Q)
   S ← S ∪ {u}
   for each v ∈ Adj[u]
      Relax(u,v,w)

Initialize-Single-Source(G, s)
for each v ∈ V
   d[v] ← ∞
   p[v] ← nil
d[s] ← 0

Relax(u, v, w)
if d[v] > d[u] + w(u,v)
   d[v] ← d[u] + w(u,v)
   p[v] ← u
```

- Dijkstra's algorithm is greedy
- Running time: $O(|E|+|V| \log |V|)$ by using a Fibonacci heap

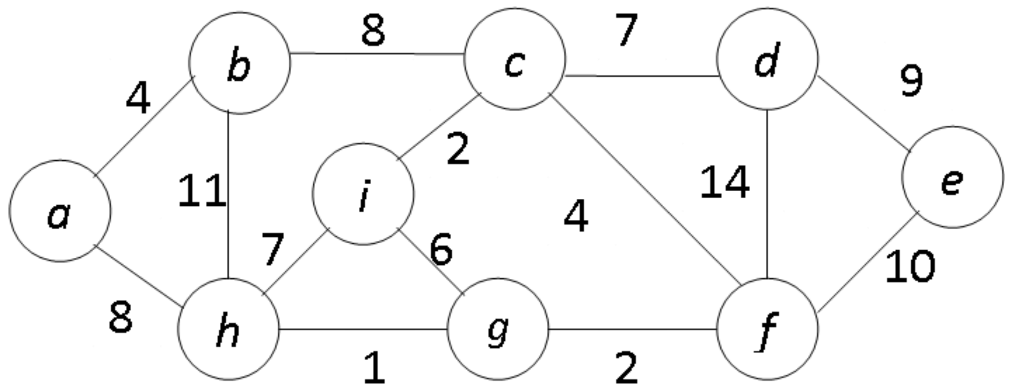
Dijkstra’s algorithm – example



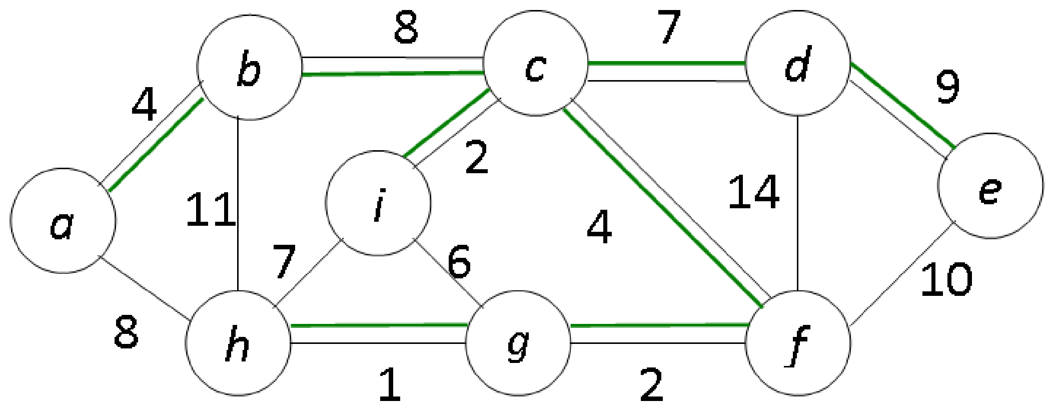
Minimum spanning trees - MST

- BFS and DFS work for “unweighted” graphs
  - …i.e. the edges have *all* the same weight
- **Definition** (MST):
  - Given a connected, undirected, weighted graph G = (V, E),
  - a MST is an *acyclic* graph T = (V_T, E_T), where:
  - V_T = V and $E_T \subseteq$ E, and
  - $w(T) = \sum_{(u,v) \in E_T} w(u,v)$ is *minimum*
- Called MST since it “spans” graph G
- Algorithms for finding the MST:
  - Kruskal’s algorithm
  - Prim’s algorithm
  - both algorithms are *greedy*
- Assumptions:
  - *G* is connected, undirected, and weighted

Weighted graph:



MST:



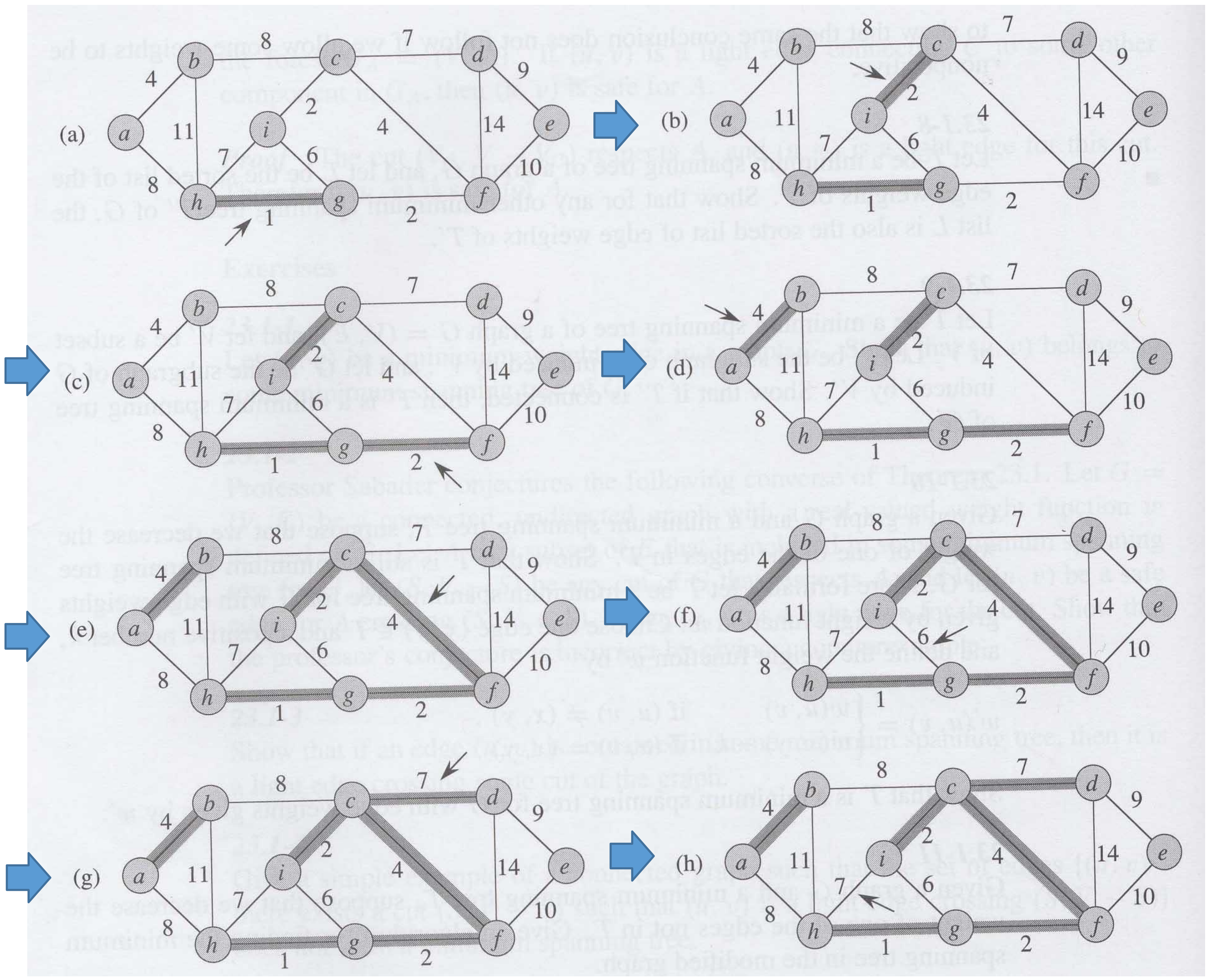
Kruskal's algorithm

- It is a *greedy* algorithm
- The set A is a forest
- Use a specific rule to find a safe edge
- Safe edge (u,v):
  - an edge whose weight is the smallest, and
  - that connects two trees, C_1 and C_2, in the forest,
  - yielding a new tree
- (u,v) is a light edge connecting C_1 to another tree,

  $\Rightarrow$ (u,v) is a safe edge for C_1
- Worst case running time: O(|E| log |E|)
- Implementation of Make-Set(v), Find-Set(u), and Union(u,v) can be found in [4]

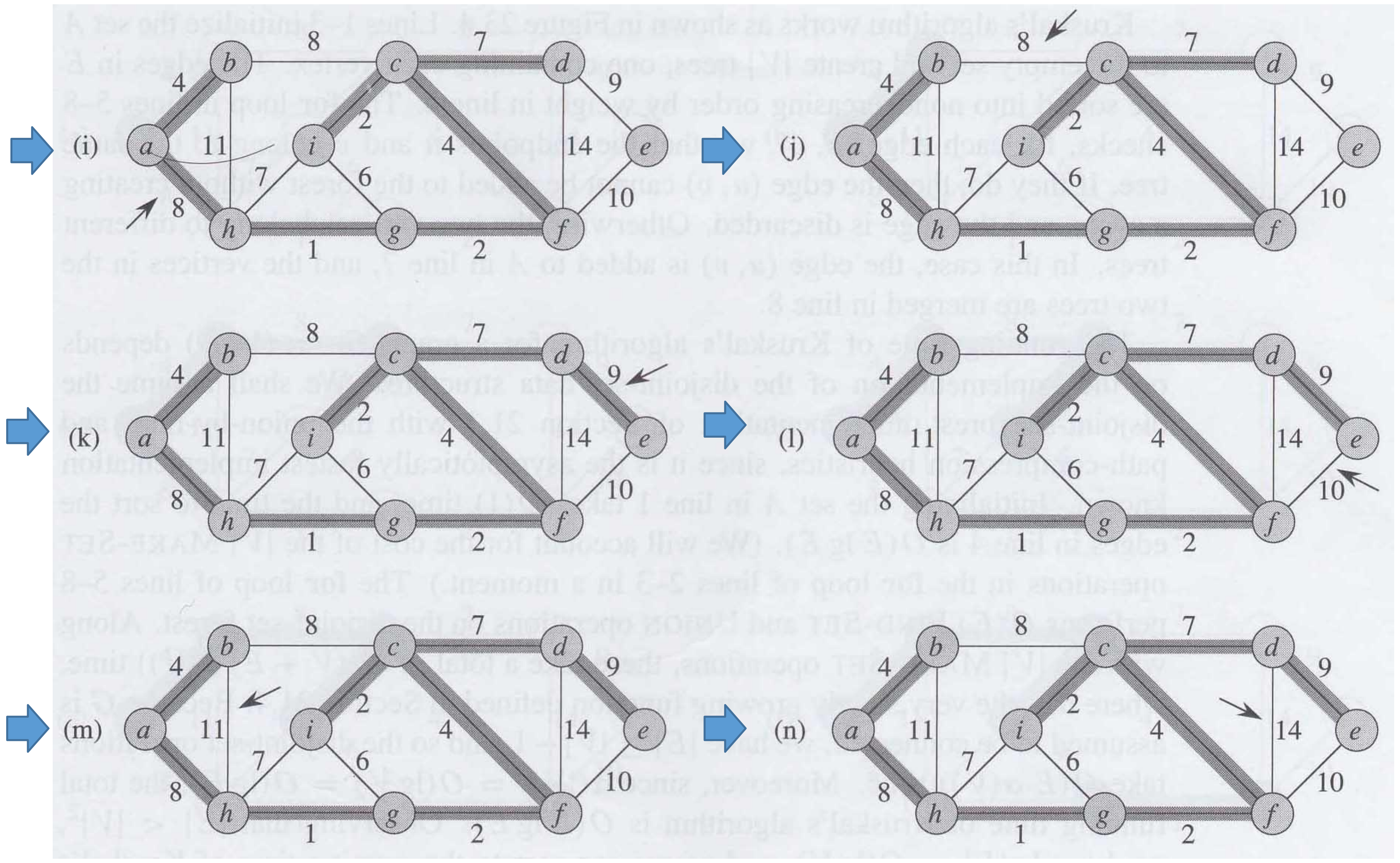
```
Algorithm MST-Kruskal(G, w)
A $\leftarrow \emptyset$
for each v $\in$ V
    Make-Set(v)
sort E in increasing order of weight w
for each (u,v) $\in$ E
    if Find-Set(u) $\neq$ Find-Set(v)
        A $\leftarrow$ A $\cup$ {(u,v)}
        Union(u,v)
return A
```

Kruskal's algorithm - example



Example not discussed in full detail – found on page 568 of [5]

Kruskal's algorithm - example



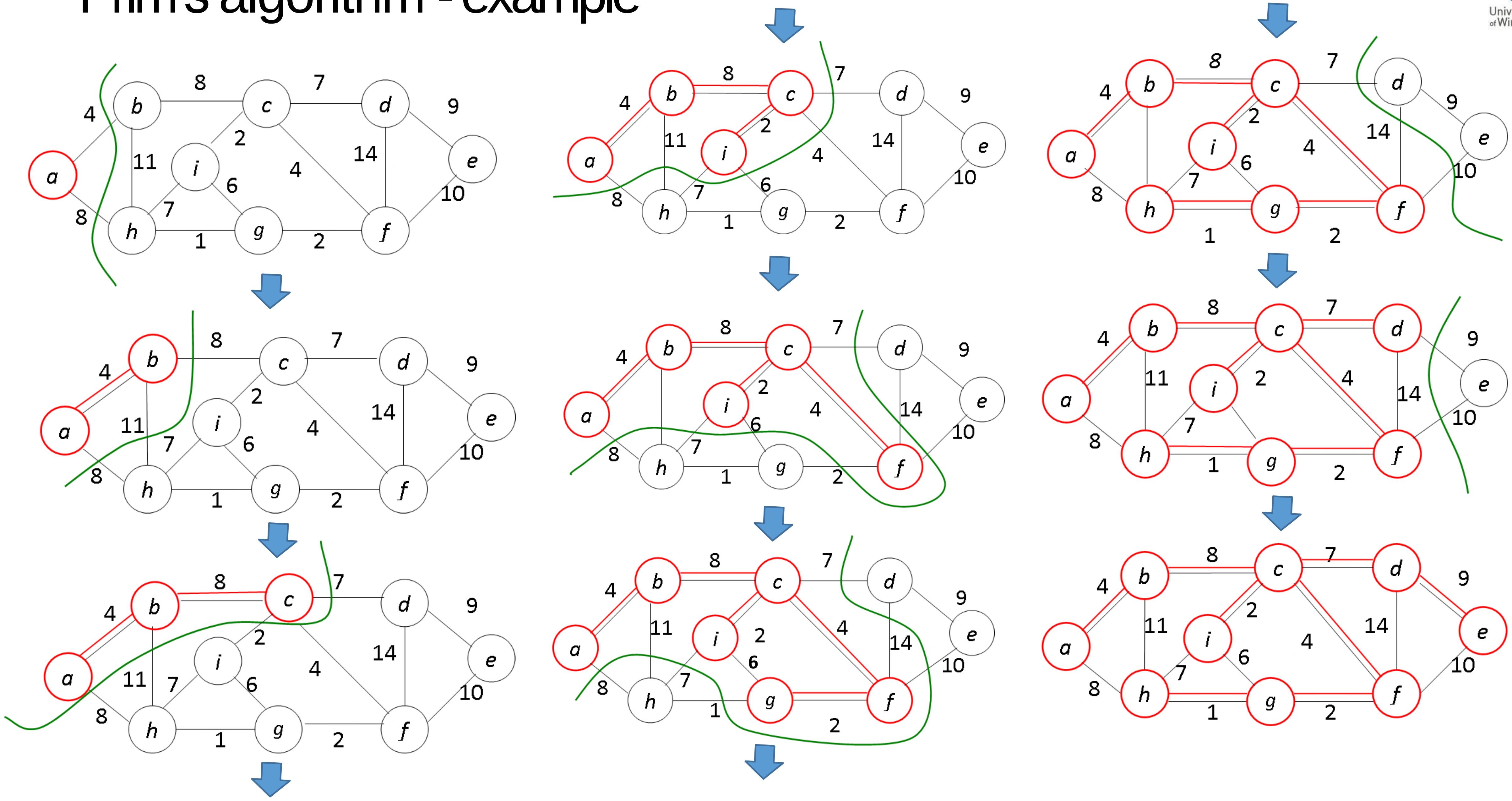
Prim’s algorithm

- Unlike Kruskal’s algorithm, it maintains a *single* tree, A.
- Starts from an arbitrary vertex, say r,

  and spans all vertices in V
- At each step,
  - adds a *safe* vertex to A
  - $\Rightarrow$ at the end, A forms a *minimum spanning tree*.
- Worst-case running time:
  - Using a binary heap: O(|E| log |V|)

```
MST-Prim(G, w, r)
for each u $\in$ V
    key[u] $\leftarrow \infty$
    p[u] $\leftarrow$ nil
key[r] $\leftarrow$ 0
Q $\leftarrow$ V                  // Q is a heap
while Q $\neq \emptyset$
    u $\leftarrow$ Extract-Min(Q)
    for each v $\in$ Adj[u]
        if v $\in$ Q and w(u,v) < key[v]
            p[v] $\leftarrow$ u
            key[v] $\leftarrow$ w(u,v)
```

Prim’s algorithm - example



Connected components – undirected graphs

- Connected components can be found in an undirected graph

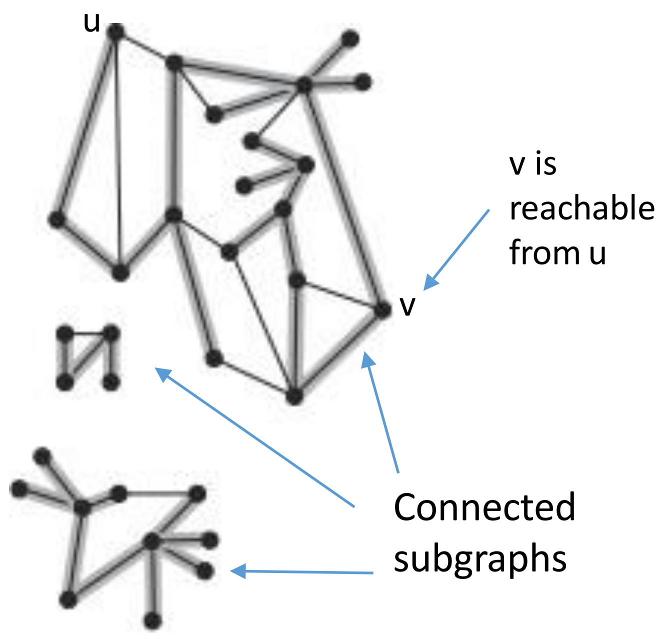
Important definitions:

- Reachable: Vertex u is reachable from v if there is a path from v to u
- Connected: A vertex v is connected to another vertex u if there is a path from v to u
- A graph is connected if there is a path from every vertex to every other vertex

High-level pseudocode:

- Given graph G
- Run DFS on G
- If DFS fails to find all vertices in G:
  - Restart DFS on unvisited vertices
- Return spanning forest

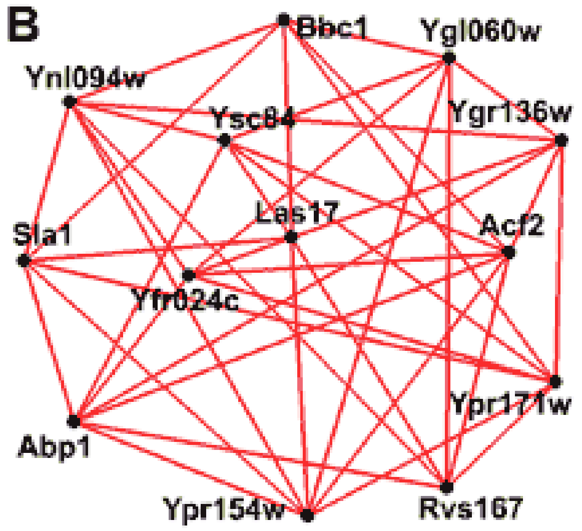
Spanning forest:

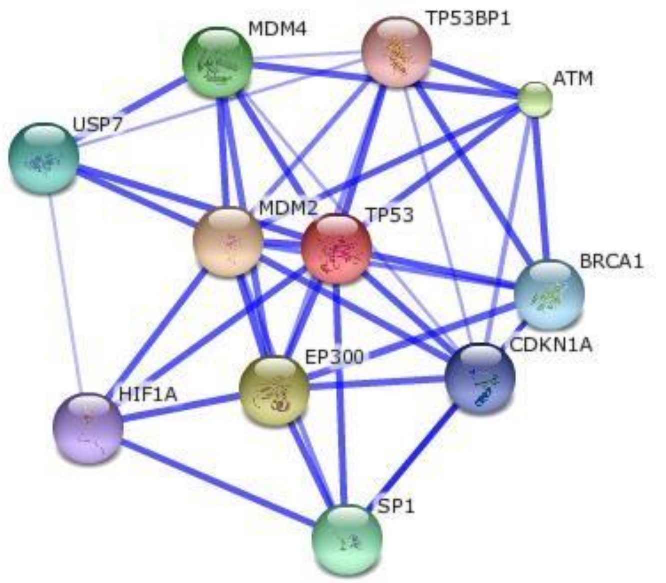


- Connected components can be found in O(|V|+|E|)

Application – Biological networks

- Protein interaction networks
- Proteins are large molecules that fold into a well defined structure
- More than 80% of the functions in living organisms are carried out by proteins
- To do some tasks protein interact with other proteins and molecules
- For example
  - hemoglobin is a complex composed of 4 interacting proteins
  - The complex is responsible for carrying oxygen thru the vessels
- Interactome: all interacting proteins in a specific organism
- Protein interaction networks are useful for a large scale view/analysis of the interactome
- Best representation of a network is by using a graph





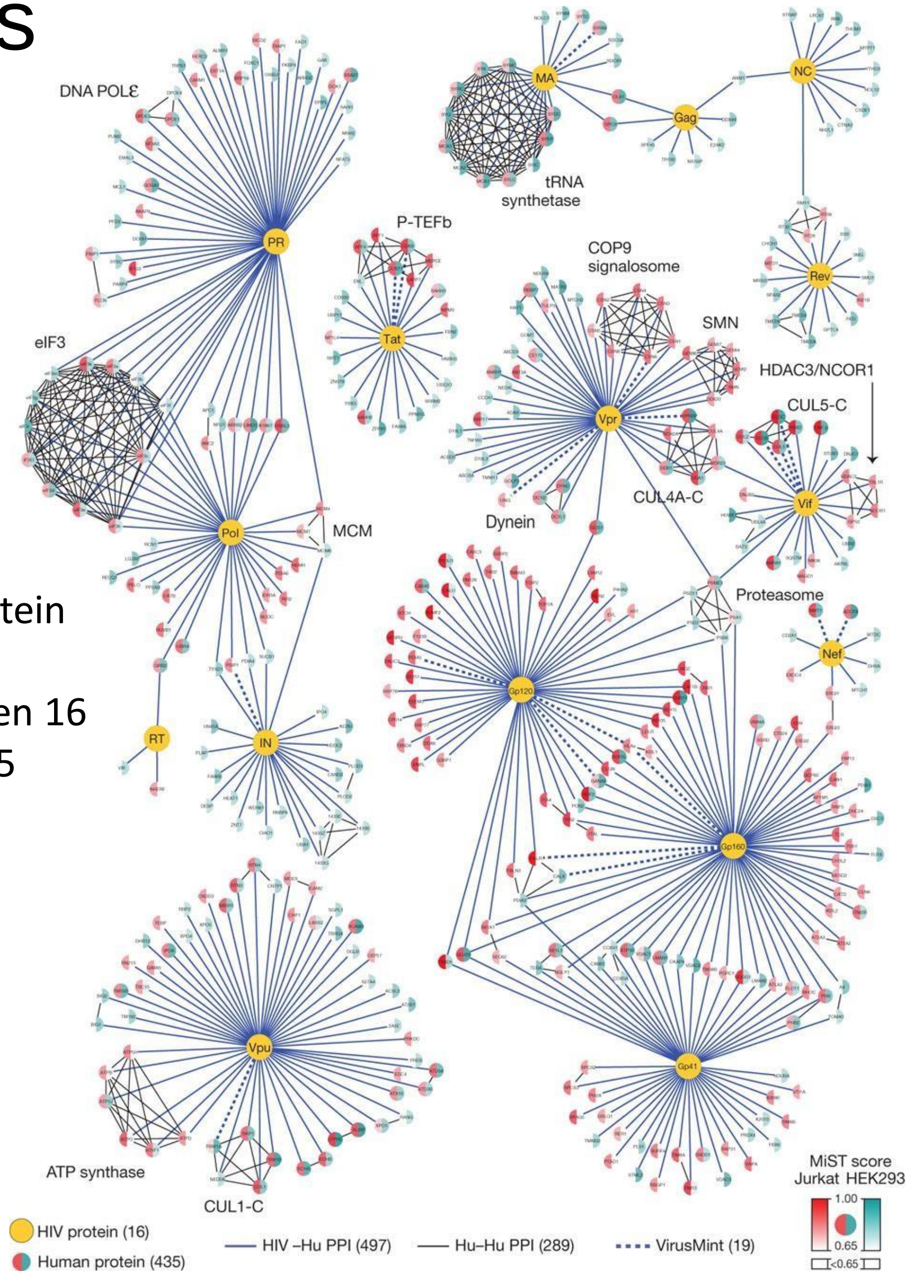
Protein interaction networks - problems

Important problems in PPI networks

- Connected components:
  - Allows to find sub-networks of proteins with related functional activity
- Hubs:
  - Vertices that are connected to a large number of other vertices
- Clusters
  - Find groups of proteins interacting with each other
  - Proteins in a group may have related functional activity
- Visualization
  - Visualizing PPI networks in a way to better understand biological processes
- Alignment of graphs
  - Allows to compare two or more different networks
- Many of these problems discussed in bioinformatics/data mining courses

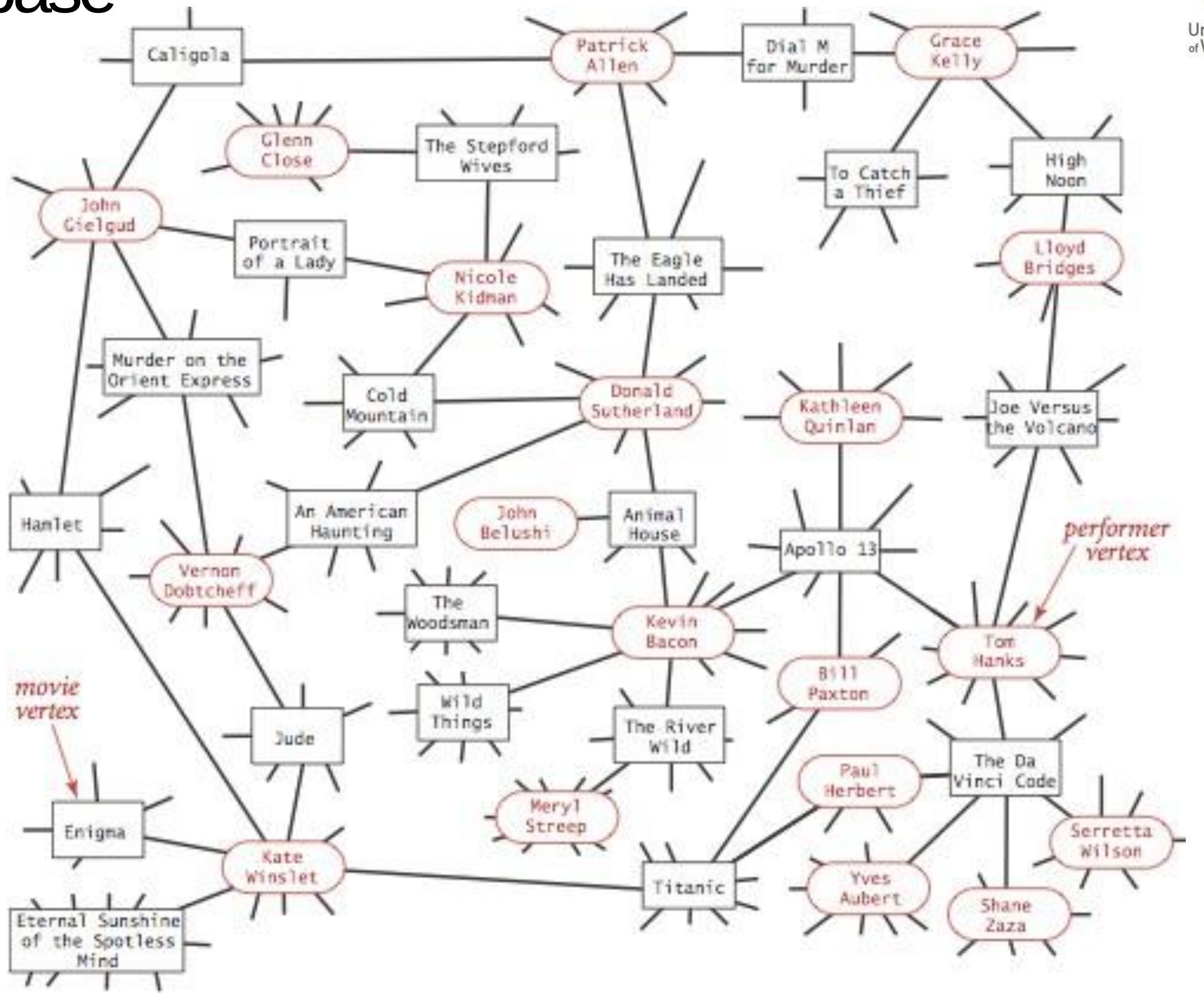
Example:
497 HIV–human protein interactions (blue) representing between 16 HIV proteins and 435 human factors [6]

clusters:



Application – the Internet movie database

- The Internet movie database (IMDB) is a repository for movies, TV and celebrities
- Its main portal at www.imdb.com contains several search tools for navigating and retrieving relevant data
- A graph of the movies and performers is available at http://algs4.cs.princeton.edu/code/ [4]
- The data for the graph is available at the file repository, called movies.txt
- The vertices correspond to movies and performers
- Edges connect movies and performers
- Entries in file:
  - Movie name and year
  - Followed by performers separated by "/"



Notting Hill (1999)/Chahidi, Paul/ ... /Grant, Hugh (I)/.../Roberts, Julia (I)/…
…
Titanic (1997)/… /DiCaprio, Leonardo/ …
…

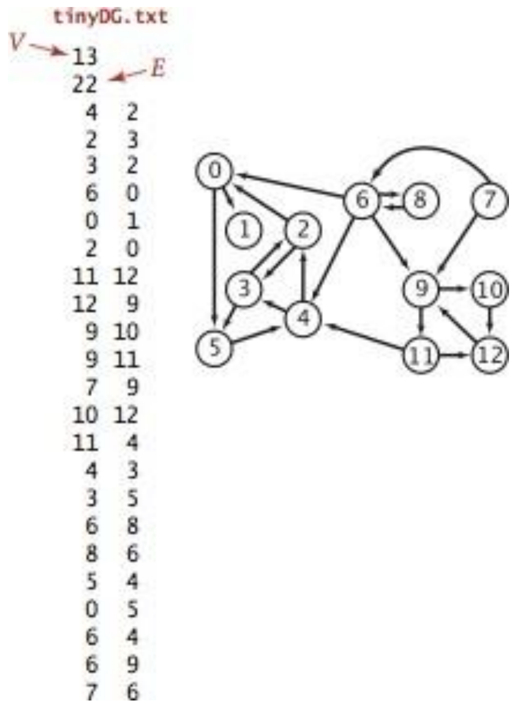
Java library for graphs

- Ref. [4] provides a good set of libraries for graphs and algorithms
- Available at
  - http://algs4.cs.princeton.edu/code/
- Includes implementations of many algorithms for graphs:
  - DFS, BFS
  - Shortest paths
  - MST
  - Connected components, and more
- Overview of graphs and algorithms
  - http://algs4.cs.princeton.edu/40graphs/
- Java documentation at
  - http://algs4.cs.princeton.edu/code/javadoc/

Examples of small graphs

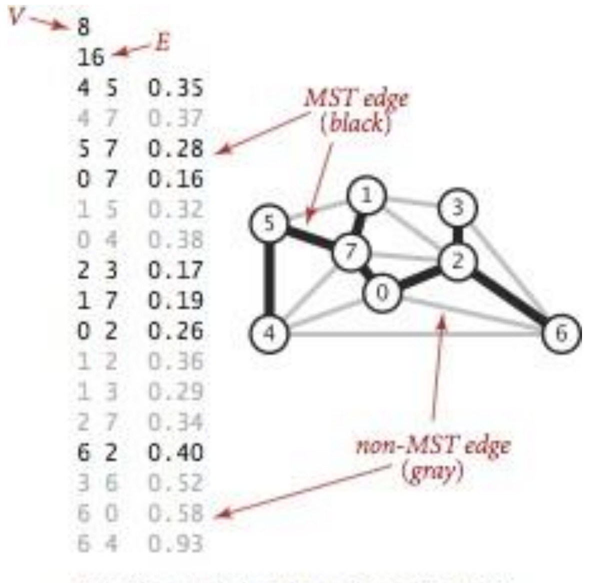
TinyDG.txt

TinyEWG.txt



```
13
22
 4  2
 2  3
 3  2
 6  0
 0  1
 2  0
11 12
12  9
 9 10
 9 11
 7  9
10 12
11  4
 4  3
 3  5
 6  8
 8  6
 5  4
 0  5
 6  4
 6  9
 7  6
```

directed graph



```
8
16
4 5  0.35
4 7  0.37
5 7  0.28
0 7  0.16
1 5  0.32
0 4  0.38
2 3  0.17
1 7  0.19
0 2  0.26
1 2  0.36
1 3  0.29
2 7  0.34
6 2  0.40
3 6  0.52
6 0  0.58
6 4  0.93
```

An edge-weighted graph and its MST

undirected weighted graph

Review and Further Reading

- Graphs, definitions, representation, examples, algorithms
  - Ch. 13 of [1], Ch 14 of [2], Ch. 9 of [3], Ch. 4 of [4], Ch. 22 of [5]
- Data structures
  - Secs. 13.1 of [1], Sec 14.2 of [2]
- Graph traversal
  - Ch 13 of [1], Sec 14.3 of [2]
- Shortest path and MST
  - Chs. 14, 15 of [1], Sec 14.6, 14.7 of [2], Secs 9.3, 9.5 of [3], Secs 4.3, 4.4 of [4], Chs 23, 24, 25 of [5]
- Connected components
  - Sec. 13.5 of [1], Sec. 9.6 of [3], Sec. 4.2 of [4], Sec 22.5 of [5]
- Examples, Java library and documentation [8]
  - http://algs4.cs.princeton.edu/40graphs/

References

1. Algorithm Design and Applications by M. Goodrich and R. Tamassia, Wiley, 2015.
2. Data Structures and Algorithms in Java, 6th Edition, by M. Goodrich and R. Tamassia, Wiley, 2014.
3. Data Structures and Algorithm Analysis in Java, 3rd Edition, by M. Weiss, Addison-Wesley, 2012.
4. Algorithms, 4th Edition, by R. Sedgewick, Addison-Wesley, 2011.
5. Introduction to Algorithms, 3rd Edition, by T. Cormen et al., McGraw-Hill, 2009.
6. Global landscape of HIV–human protein complexes by S. Jäger et al. Nature 481, 365–370 (19 January 2012).
7. Internet movie database: http://www.imdb.com/
8. Java library, examples and documentation: http://algs4.cs.princeton.edu/40graphs/

Lab – Practice

1. DFS: Use DepthFirstOrder.java and mediumDG.txt
   a) Write a program that shows the vertices of the graph in pre-order and post-order.

2. Shortest Path: Use DijkstraSP.java and mediumEWG.txt
   a) Write a program that finds the shortest path between a source vertex and every other vertex.

3. MST: Use KruskalMST.java and mediumEWG.txt
   a) Write a program that finds the minimum spanning tree of an undirected graph.

4. Connected components: Use TarjanSCC.java and mediumDG.txt
   a) Write a program that finds the strongly connected components.

5. Modify your inputs and test your programs #1 to #4. Comment on the results.
   a) Run programs several times on large graphs. Find average running times and compare them with the complexities of the corresponding algorithms.

Exercises

1. Give examples of directed and undirected graphs. Find real applications of directed and undirected graphs.
2. Give examples of sparse and dense graphs. Find real applications of sparse and dense graphs.
3. When will you use the adjacency list representation? And the adjacency matrix? Give examples and discuss complexity issues.
4. Consider the graphs of page 6. Find the (strongly) connected components. Find the shortest paths for all pairs of vertices. Give examples of reachable pairs of vertices.
5. Apply BFS and DFS to the graphs of page 6. Show the differences between the two algorithms and complexity issues.
6. Consider the undirected graph of page 6. Suppose it has no weights. Design a maze and show how to use BFS to explore the maze. Show how to go from one place to another. Find a real application for maze represented as a graph.
7. *For Kruskal's algorithm, discuss the implementations of Make-Set(v), Find-Set(u), and Union(u,v). You will find them in [4].
8. *Consider a robot that has to navigate through a map of polygons. Consider starting and end points for the robot to move through. Using Dijkstra's algorithm, write a program that finds the shortest path between the starting and end point.
9. *Consider a robot vacuum cleaner that has to clean a room in which there are some objects. Assume the robot moves from one tile to another and that the tiles are vertices of a graph, and adjacent tiles are connected by edges. Design (and implement) an algorithm that creates a cleaning path plan.
10. For the graph of page 23, explain how we would use DFS to find all connected components.
11. For the graphs of page 25, explain how you find strongly connected components and MST (using the algorithms, not the programs run in the lab).
12. *Consider a map of cities in Ontario and the corresponding driving distances and costs between neighbor cities. Write a program that finds the shortest path for driving between one city to another. Take both distances and costs into account, while letting the user to decide which of these have to be optimized.
13. Give examples of directed graphs with n vertices that have: (a) n strongly connected components, (b) one strongly connected component.
14. For the graphs of page 6, give examples of paths, connected vertices, cycles. Are these graphs acyclic? Find subgraphs that are acyclic.
15. For the graph representing the Internet movie database, write an algorithm that lists all performers. Do the same for all movies. You may use any of the algorithms discussed in this chapter. Provide an explanation of why you used such an algorithm and derive the worst-case running time for your algorithm.
16. Design an algorithm that finds the longest path between two vertices, and discuss its complexity.