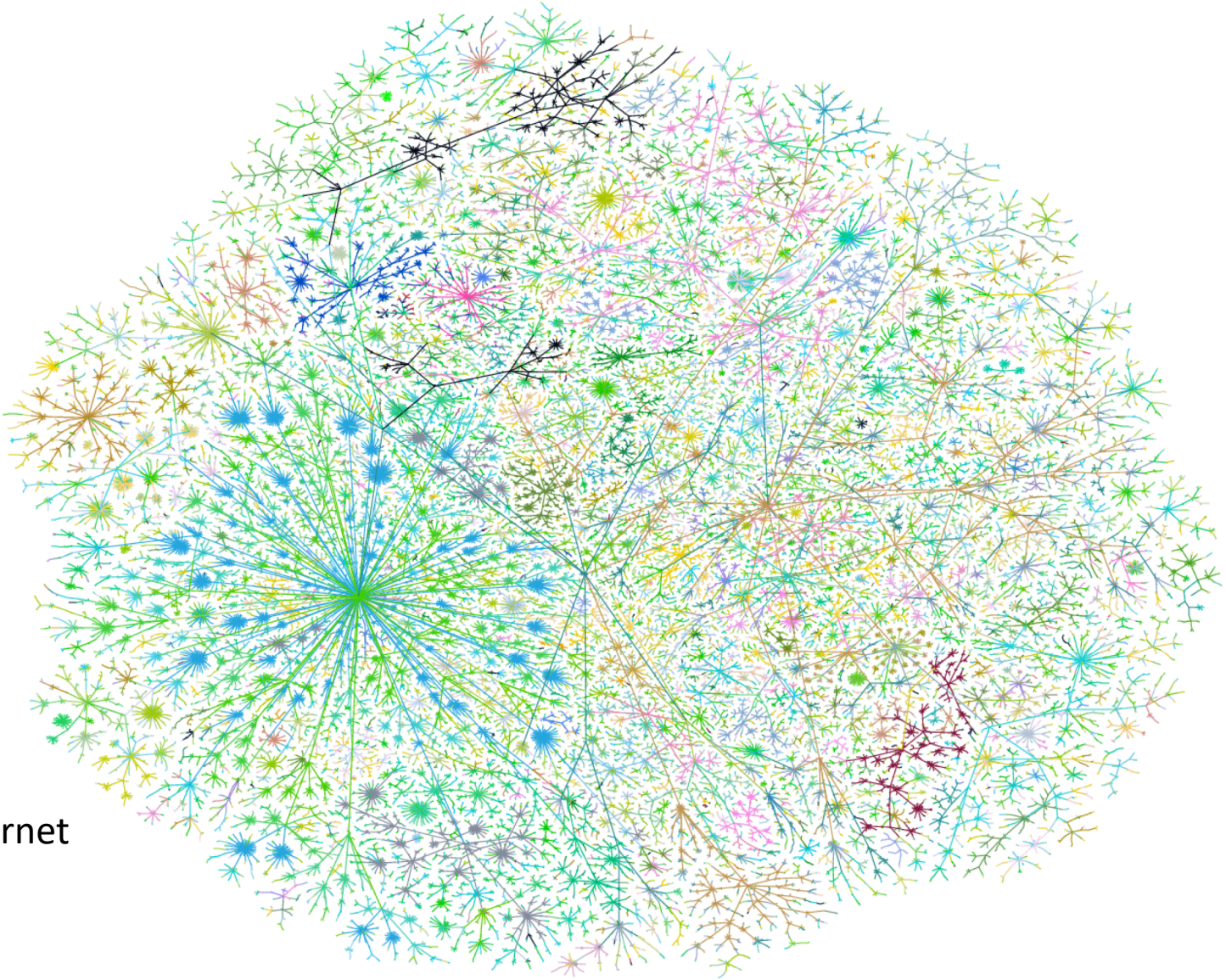
Lecture 11

Graph Algorithms

Wilson Rivera

Graphs



Graph of the internet
(circa 1999)

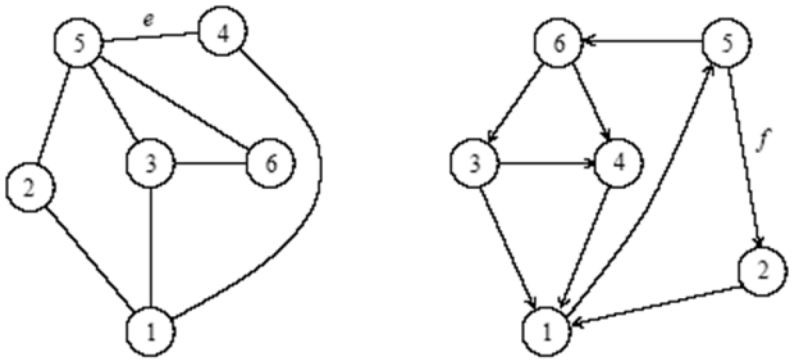
Facebook Friendship

Source: Paul Butler

Graph Definitions

- An *undirected graph G* is a pair *(V,E)*, where *V* is a finite set of points called *vertices* and *E* is a finite set of *edges*.
  - An edge $e \in E$ is an unordered pair *(u,v)*, where $u,v \in V$.

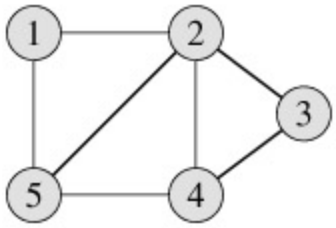
- In a directed graph, the edge *e* is an ordered pair *(u,v)*
  - An edge *(u,v)* is *incident from* vertex *u* and is *incident to* vertex *v*.



Graph Definitions

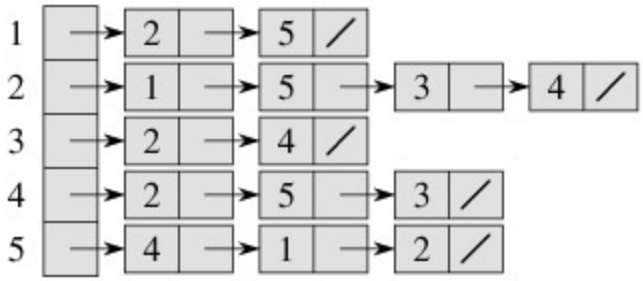
- An undirected graph is *connected* if every pair of vertices is connected by a path.
- A *tree* is a connected acyclic graph.
- A graph that has weights associated with each edge is called a *weighted graph*.

Graph Representation – Undirected



(a)

graph



(b)

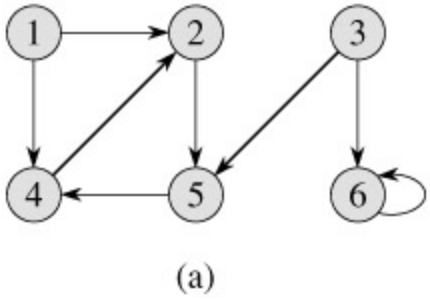
Adjacency list

	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

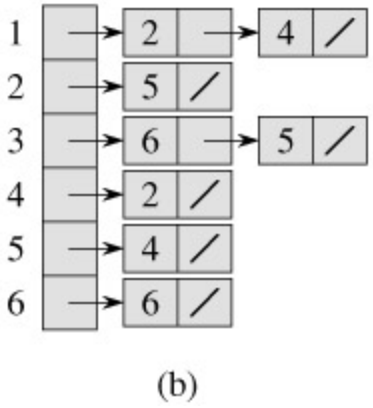
(c)

Adjacency matrix

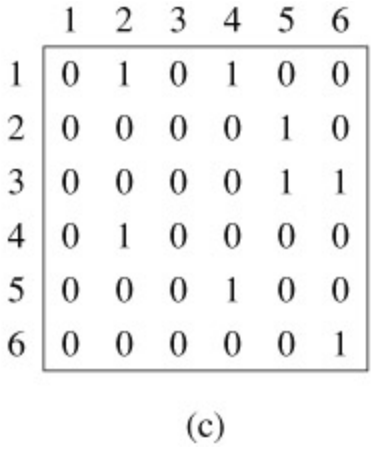
Graph Representation – Directed



graph



Adjacency list



Adjacency matrix

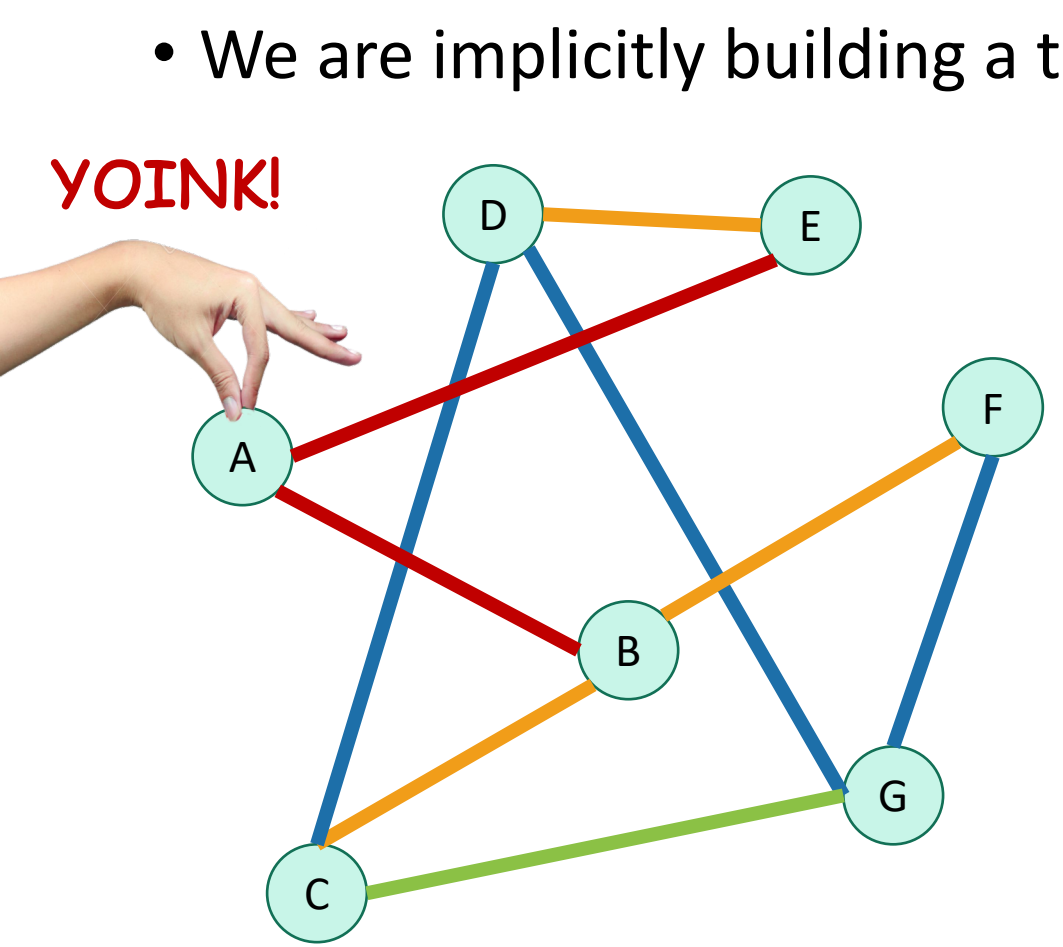
Graph Representation

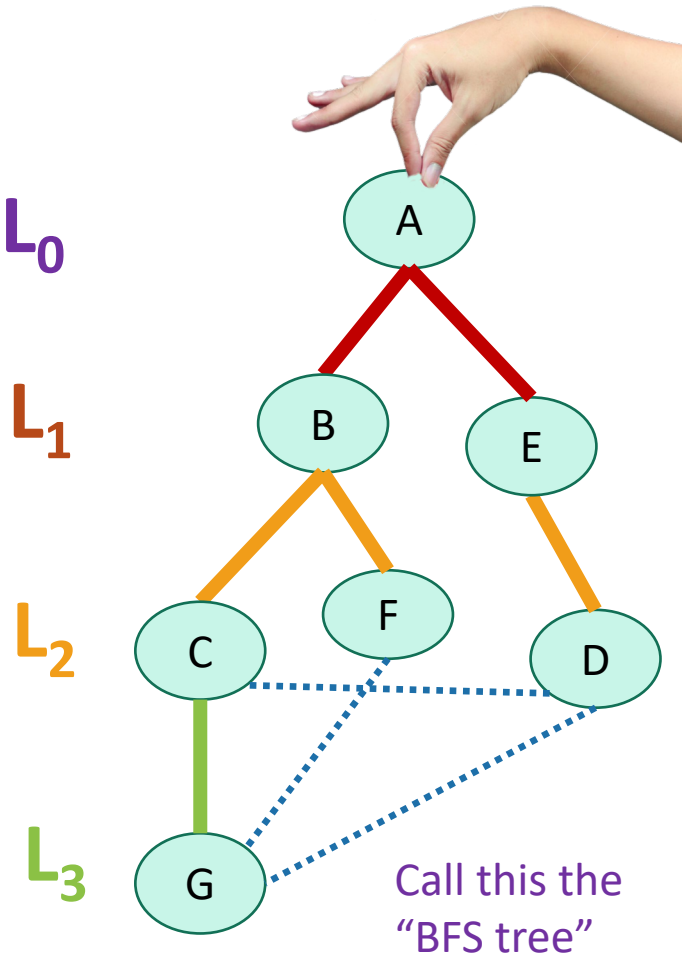
- Adjacency list representation is usually preferred since it is more efficient in representing sparse graphs.
  - Graphs for which $|E|$ is much less than $|V|^2$
- Adjacency list requires memory of the order of $\theta(V+E)$

Breadth-first search

Why is it called breadth-first?

- We are implicitly building a tree:

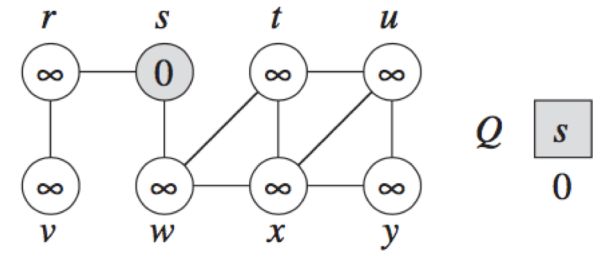


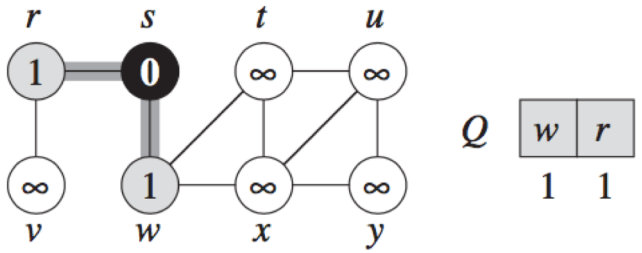


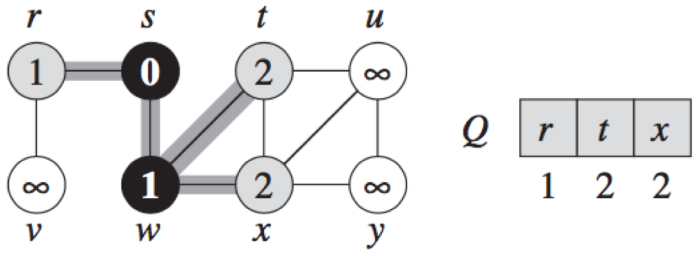
- And first we go as broadly as we can.

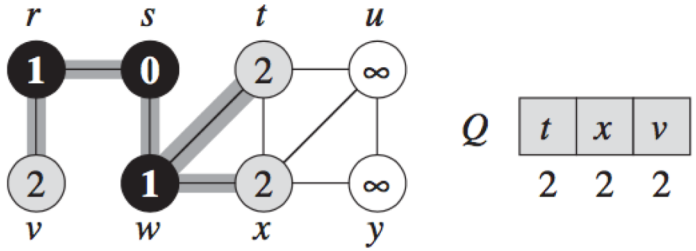
BFS

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each v ∈ G.Adj[u]
13            if v.color == WHITE
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)
18       u.color = BLACK
```

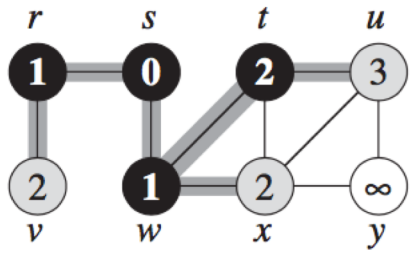


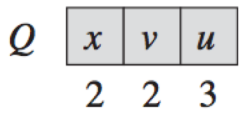


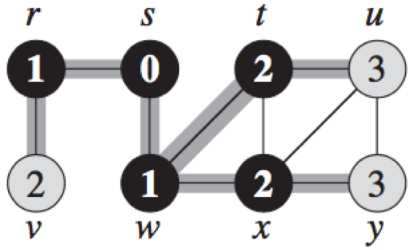


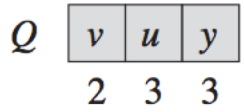


BFS

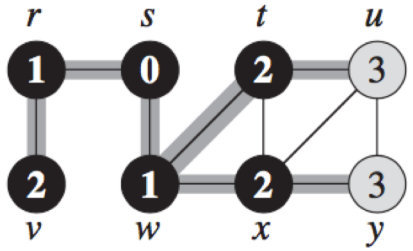


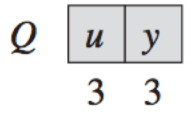


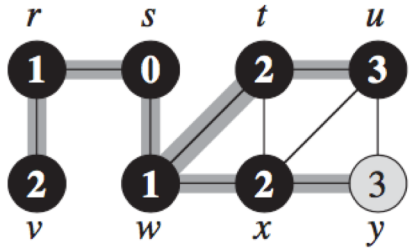


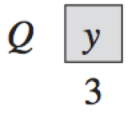


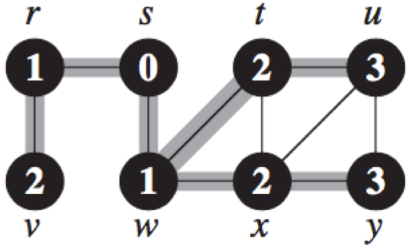

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each v ∈ G.Adj[u]
13            if v.color == WHITE
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)
18       u.color = BLACK
```











$$O(V + E)$$

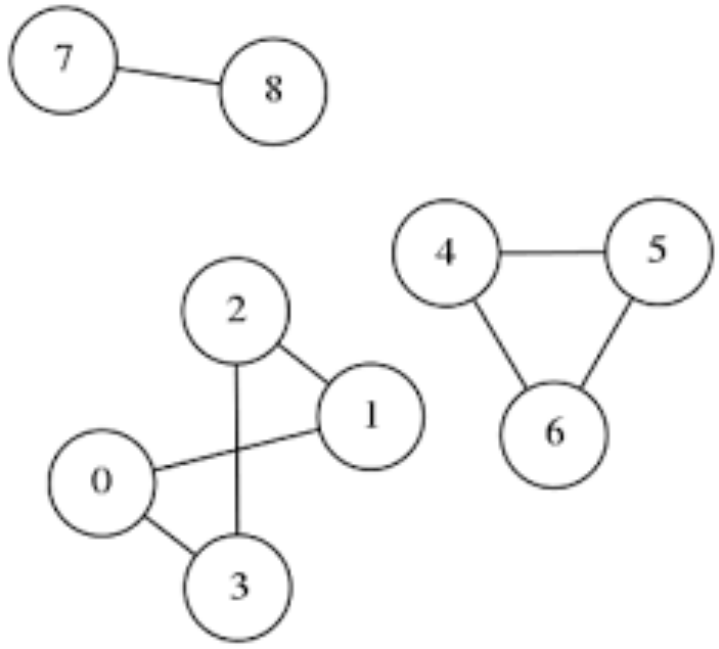
Shortest Path

The shortest-path distance $\delta(s, v)$ from s to v is defined as the minimum number of edges in any path from vertex s to vertex v

```
PRINT-PATH(G, s, v)
1  if v == s
2       print s
3  elseif v.π == NIL
4       print "no path from" s "to" v "exists"
5  else PRINT-PATH(G, s, v.π)
6       print v
```

Connected Components

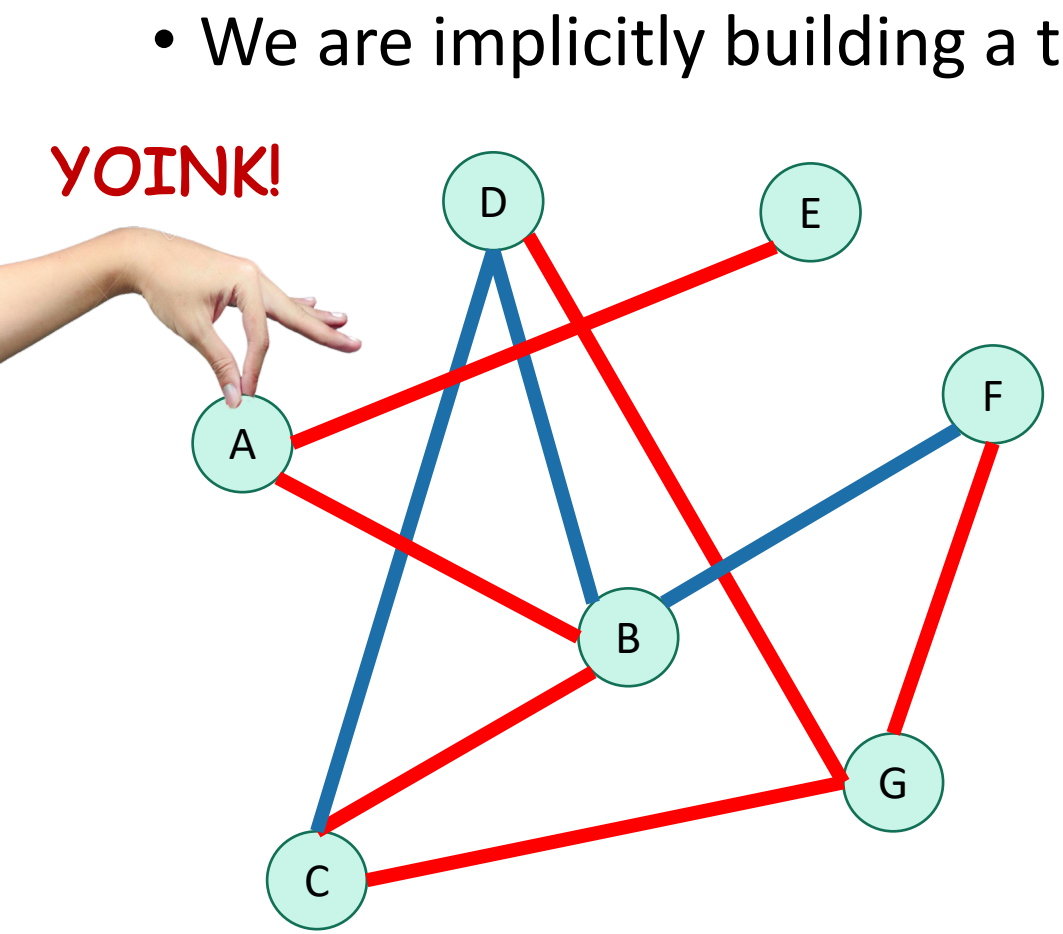
```
for i=1 to n
  if I is unexplored
     BFS(G, i)
```

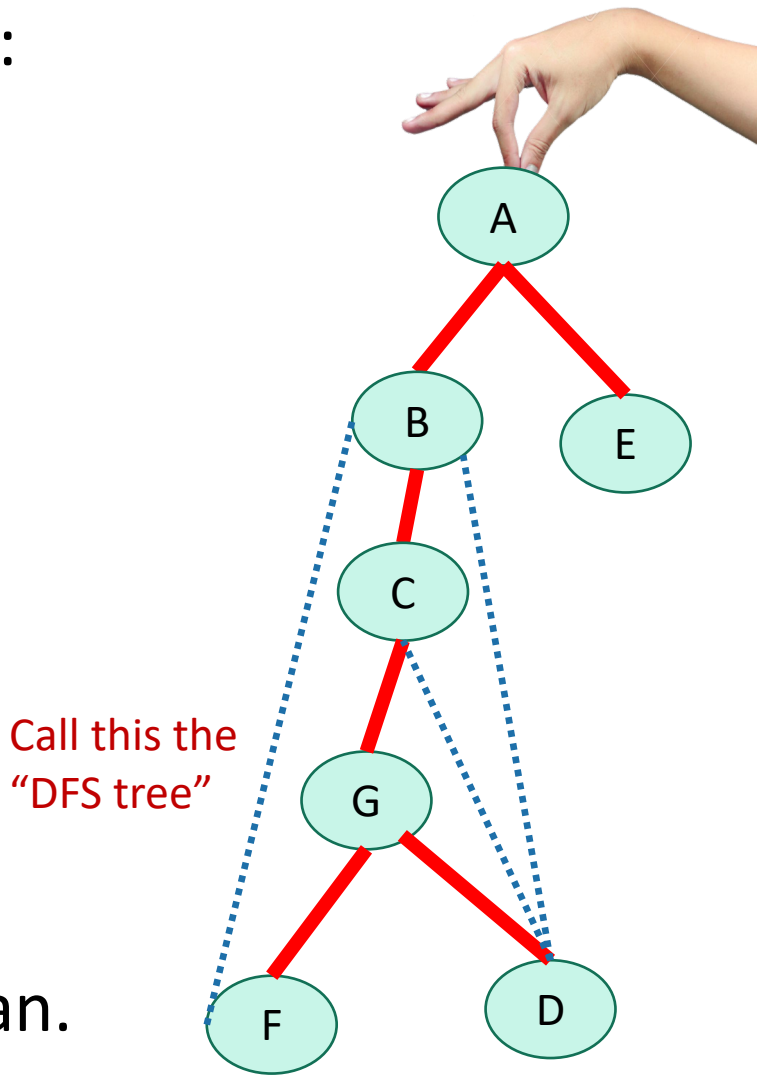


Depth-first search

Why is it called depth-first?

- We are implicitly building a tree:





Call this the "DFS tree"

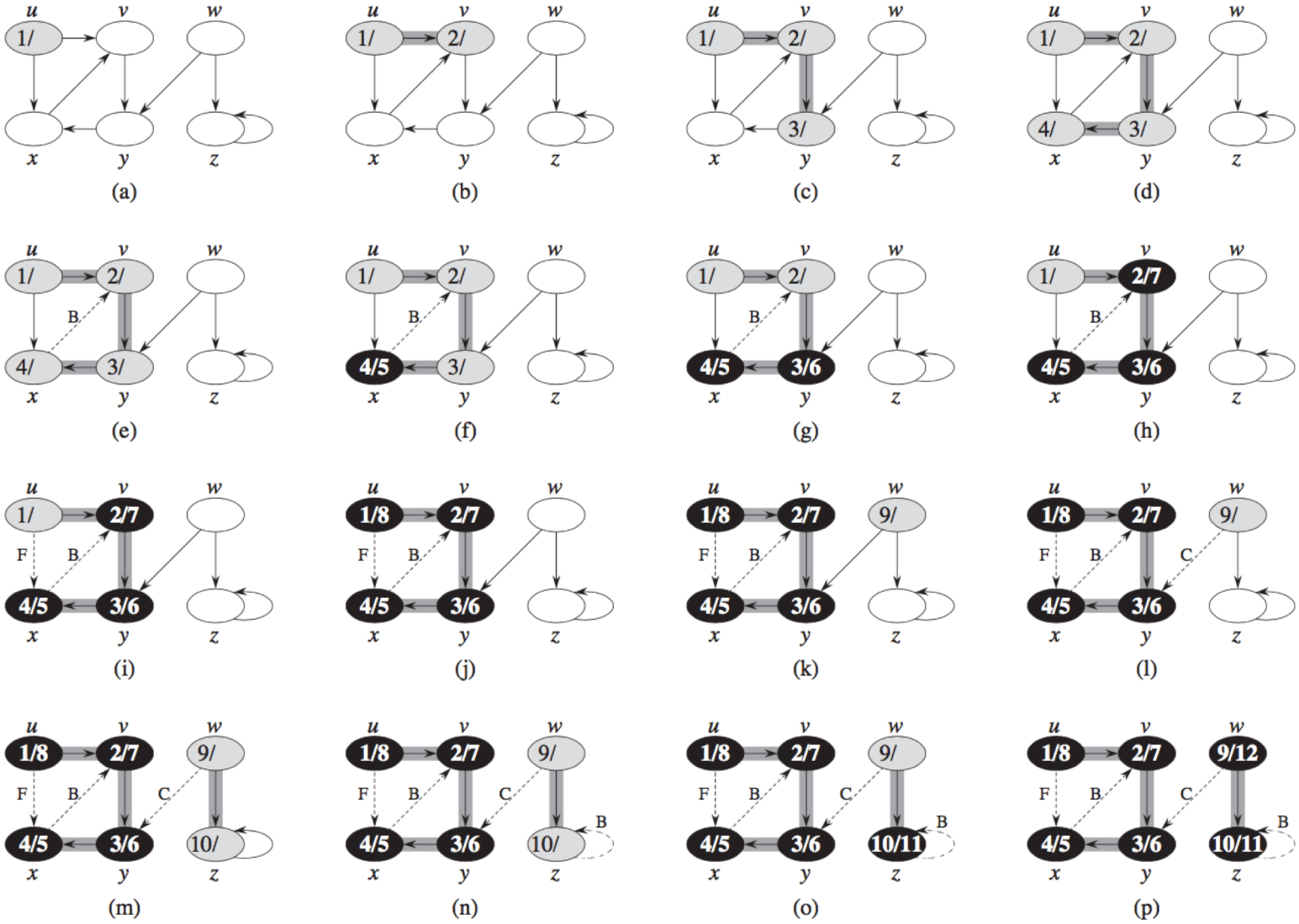
- And first we go as deep as we can.

DFS

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

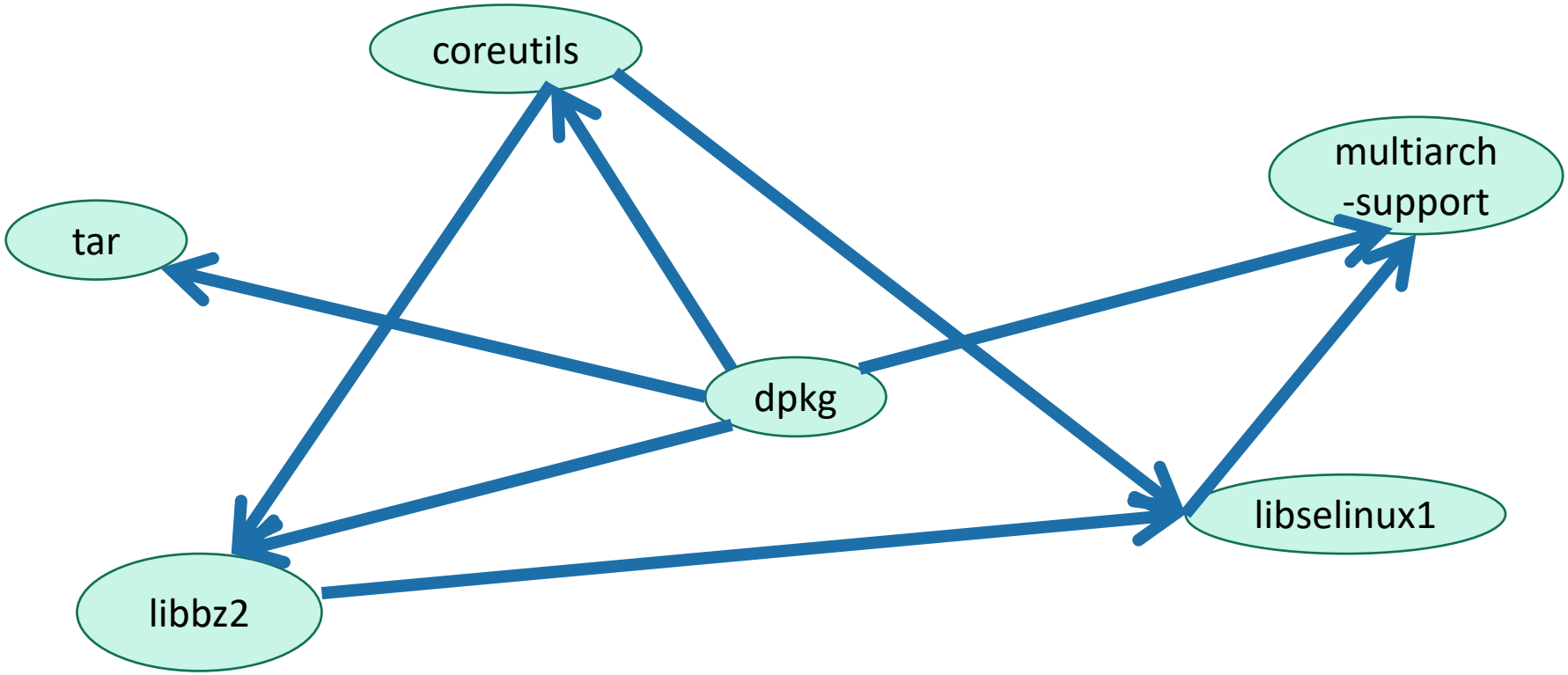
```
DFS-VISIT(G, u)
 1  time = time + 1
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f = time
```

DFS


u v w
1/
x y z
(a)
u v w
1/ 2/
x y z
(b)
u v w
1/ 2/
3/
x y z
(c)
u v w
1/ 2/
4/ 3/
x y z
(d)
u v w
1/ 2/
B
4/ 3/
x y z
(e)
u v w
1/ 2/
B
4/5 3/
x y z
(f)
u v w
1/ 2/
B
4/5 3/6
x y z
(g)
u v w
1/ 2/7
B
4/5 3/6
x y z
(h)
u v w
1/ 2/7
F B
4/5 3/6
x y z
(i)
u v w
1/8 2/7
F B
4/5 3/6
x y z
(j)
u v w
1/8 2/7 9/
F B
4/5 3/6
x y z
(k)
u v w
1/8 2/7 9/
F B C
4/5 3/6
x y z
(l)
u v w
1/8 2/7 9/
F B C
4/5 3/6 10/
x y z
(m)
u v w
1/8 2/7 9/
F B C
4/5 3/6 10/ B
x y z
(n)
u v w
1/8 2/7 9/
F B C
4/5 3/6 10/11 B
x y z
(o)
u v w
1/8 2/7 9/12
F B C
4/5 3/6 10/11 B
x y z
(p)

Application: Topological sorting

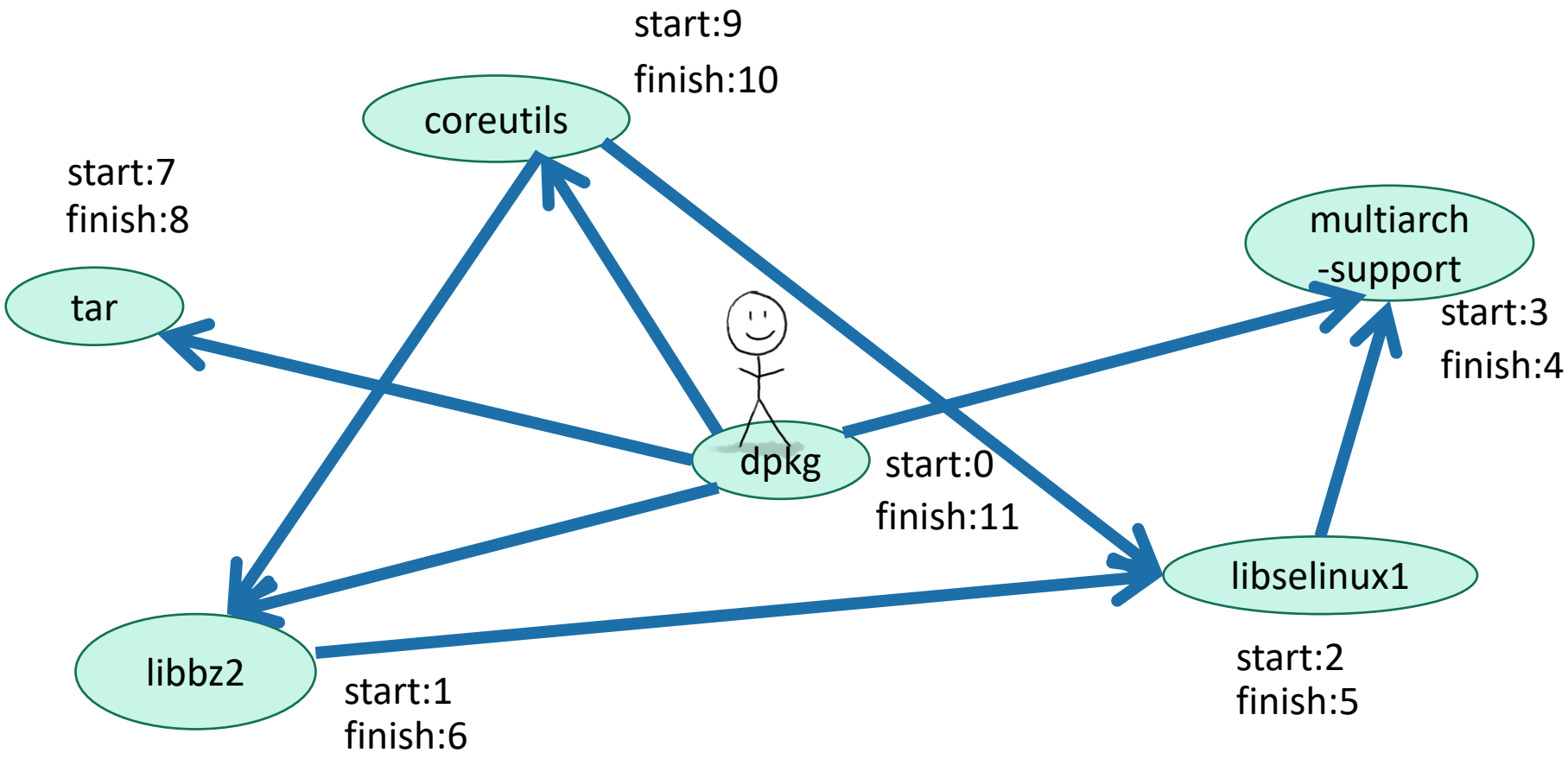
- Example: package dependency graph
- Question: in what order should I install packages?



Topological sorting

Problem of finding an ordering that respects all the dependencies

- Packages we should include earlier have larger **finish times.**



Summary Graph Searching

- Breath First Search (BFS)
  - Shortest path
  - Connected components
- Depth First Search (DFS)
  - Topological sorting
  - Connected components

$$O(V + E)$$

Minimum Spanning Tree

Minimum Spanning Tree

- A *spanning tree* of an undirected graph *G* is a subgraph of *G* that is a tree containing all the vertices of *G*.

- In a weighted graph, the weight of a subgraph is the sum of the weights of the edges in the subgraph.

- A *minimum spanning tree* (MST) for a weighted undirected graph is a spanning tree with minimum weight.

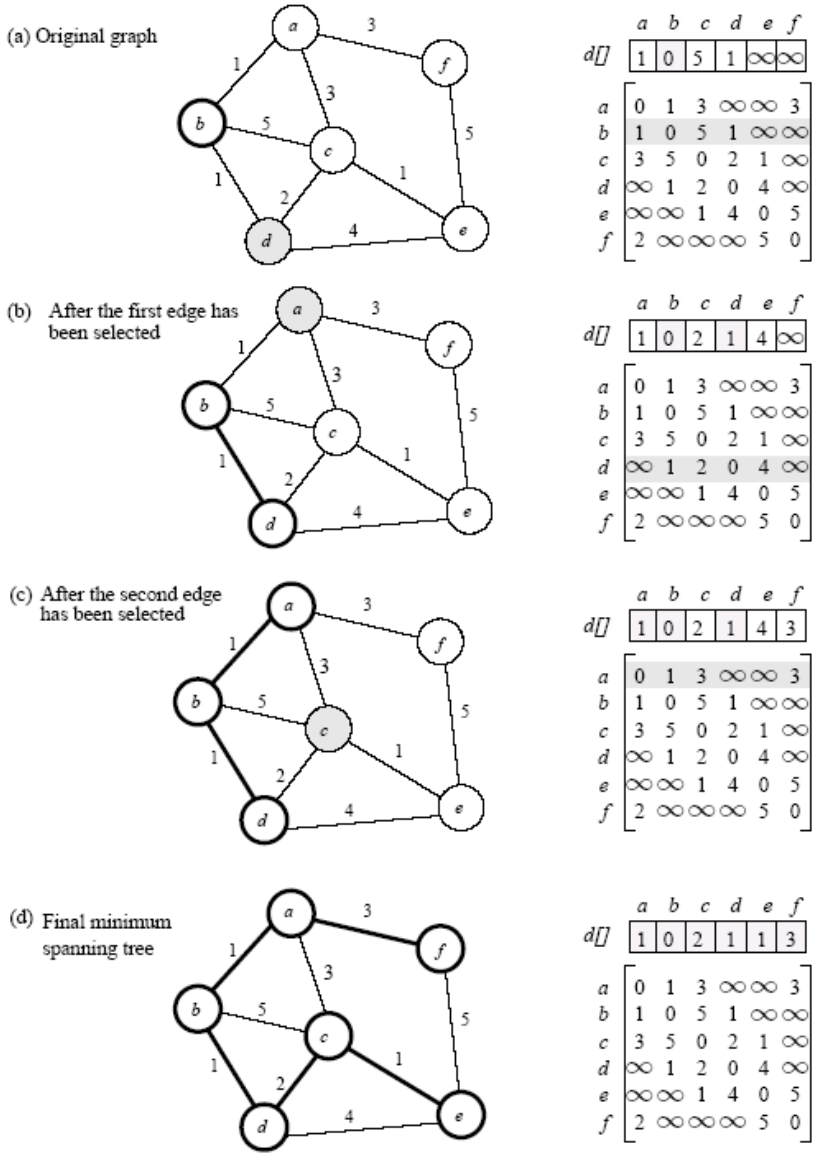
Minimum Spanning Tree: Prim's Algorithm

```
1.    procedure PRIM_MST(V, E, w, r)
2.    begin
3.        V_T := {r};
4.        d[r] := 0;
5.        for all v ∈ (V − V_T) do
6.            if edge (r, v) exists set d[v] := w(r, v);
7.            else set d[v] := ∞;
8.        while V_T ≠ V do
9.        begin
10.           find a vertex u such that d[u] := min{d[v]|v ∈ (V − V_T)};
11.           V_T := V_T ∪ {u};
12.           for all v ∈ (V − V_T) do
13.               d[v] := min{d[v], w(u, v)};
14.       endwhile
15.   end PRIM_MST
```

Prim's sequential minimum spanning tree algorithm.

Minimum Spanning Tree: Prim's Algorithm

```
1.  procedure PRIM_MST(V, E, w, r)
2.  begin
3.      V_T := {r};
4.      d[r] := 0;
5.      for all v ∈ (V − V_T) do
6.          if edge (r, v) exists set d[v] := w(r, v);
7.          else set d[v] := ∞;
8.      while V_T ≠ V do
9.      begin
10.         find a vertex u such that d[u] := min{d[v]|v ∈ (V − V_T)};
11.         V_T := V_T ∪ {u};
12.         for all v ∈ (V − V_T) do
13.             d[v] := min{d[v], w(u, v)};
14.     endwhile
15. end PRIM_MST
```



Prim's minimum spanning tree algorithm.

Single Shortest Path

Single Shortest Path

- For a weighted graph *G = (V,E,w)*, the *single-source shortest paths* problem is to find the shortest paths from a vertex $v \in V$ to all other vertices in *V*.

- Dijkstra's algorithm is similar to Prim's algorithm. It maintains a set of nodes for which the shortest paths are known.
  - Open Shortest Path First (OSPF)
  - A* Algorithm for Path finding

Single Shortest Path: Dijkstra's Algorithm

```
1.   procedure DIJKSTRA_SINGLE_SOURCE_SP(V, E, w, s)
2.   begin
3.       V_T := {s};
4.       for all v ∈ (V − V_T) do
5.           if (s, v) exists set l[v] := w(s, v);
6.           else set l[v] := ∞;
7.       while V_T ≠ V do
8.       begin
9.           find a vertex u such that l[u] := min{l[v]|v ∈ (V − V_T)};
10.          V_T := V_T ∪ {u};
11.          for all v ∈ (V − V_T) do
12.              l[v] := min{l[v], l[u] + w(u, v)};
13.      endwhile
14.  end DIJKSTRA_SINGLE_SOURCE_SP
```

Dijkstra's sequential single-source shortest paths algorithm.