Algorithm and Object-Oriented Programming for Modeling
Part 4: Graph Related Algorithms

MSDM 5051, Yi Wang (王一), HKUST

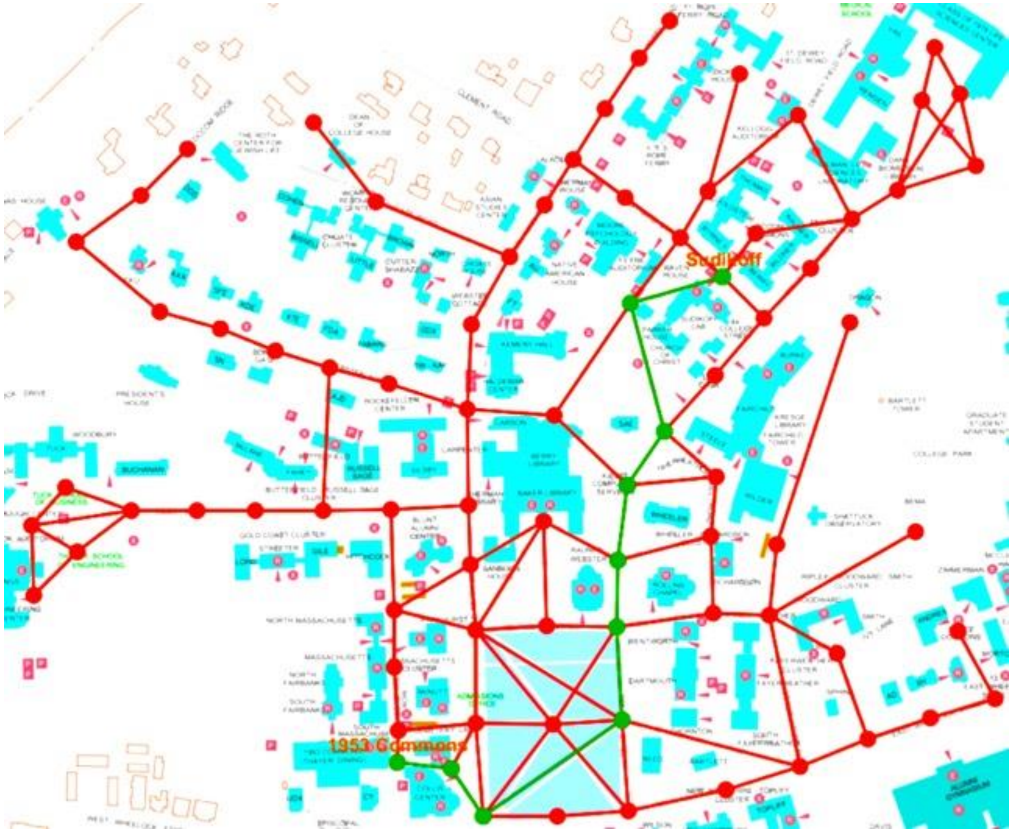
Classification of data structures:

- Linear: at most one predecessor & one successor
- Tree: at most one predecessor & any # of successor
- Graph: any # of predecessor & successor

Section 1. Why graphs?

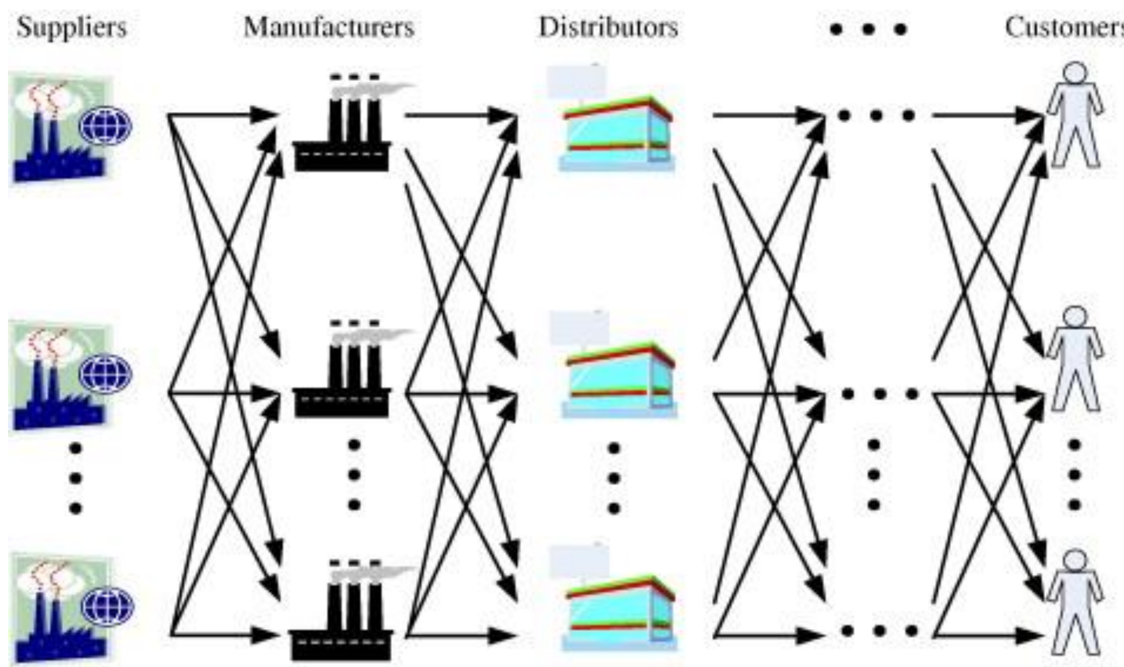
Why graphs? Model of many real-world systems

Traffic



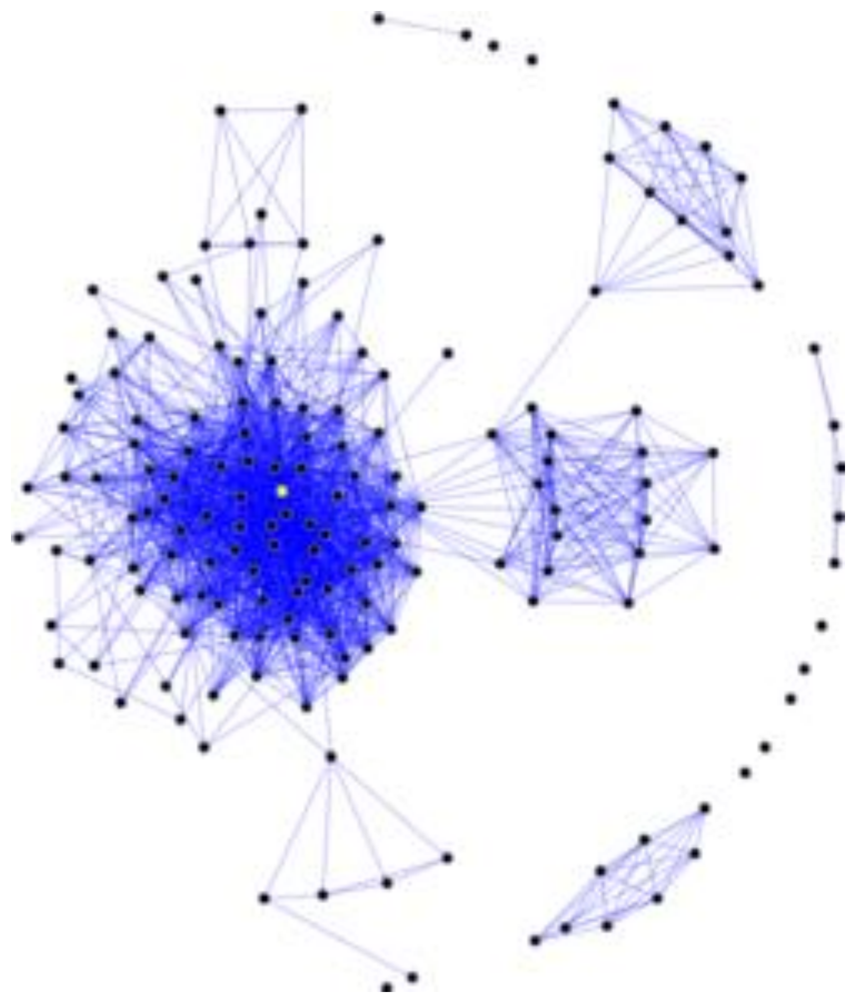
Why graphs? Model of many real-world systems

Supply chain



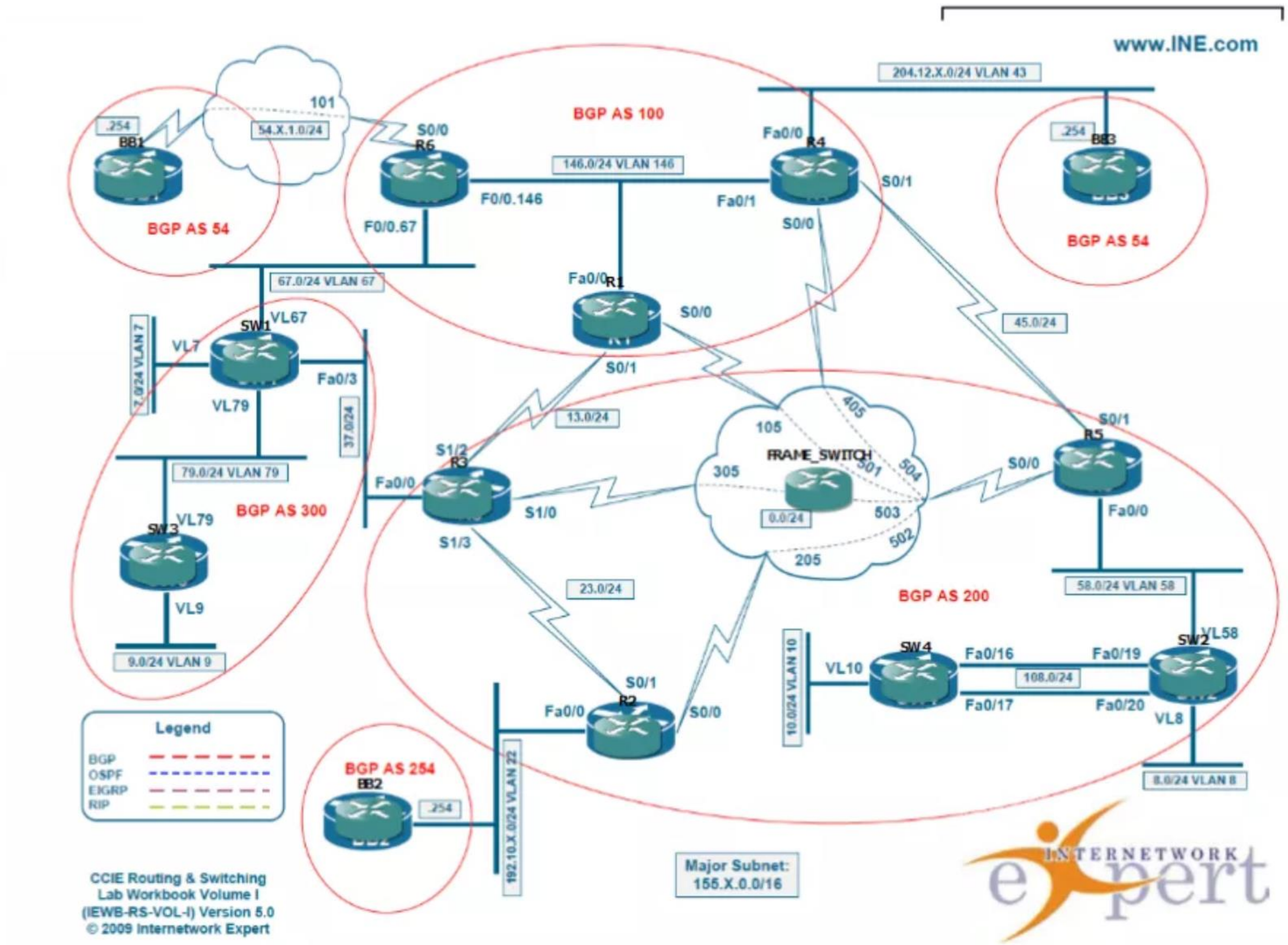
Why graphs? Model of many real-world systems

Social network



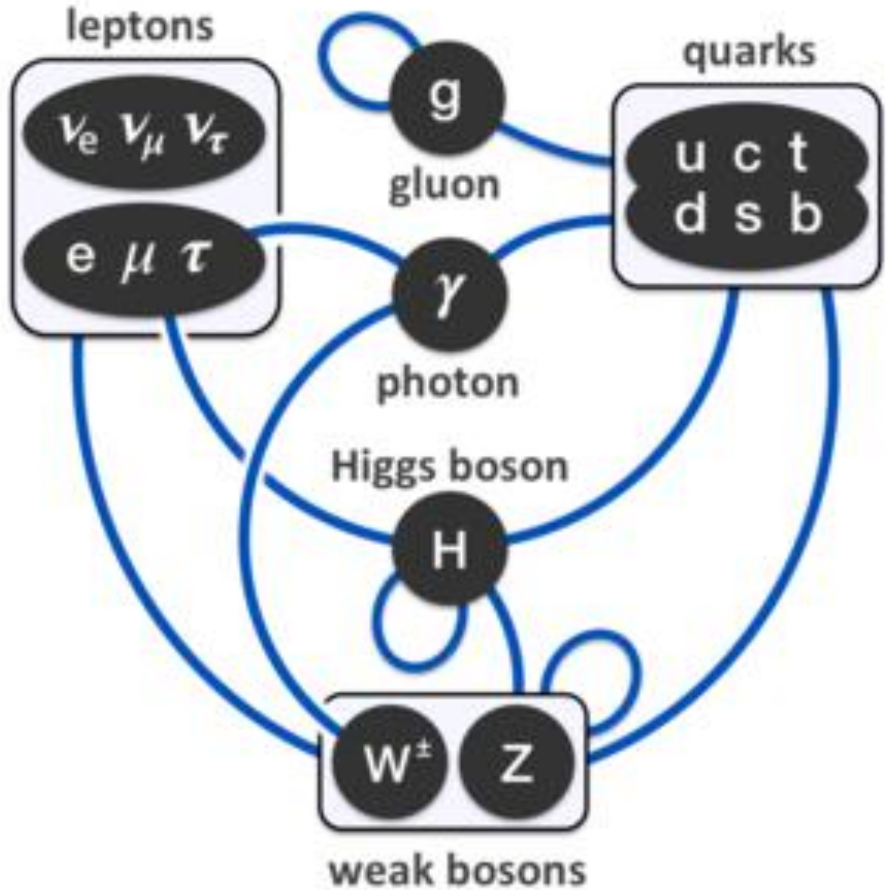
Why graphs? Model of many real-world systems

Computer network



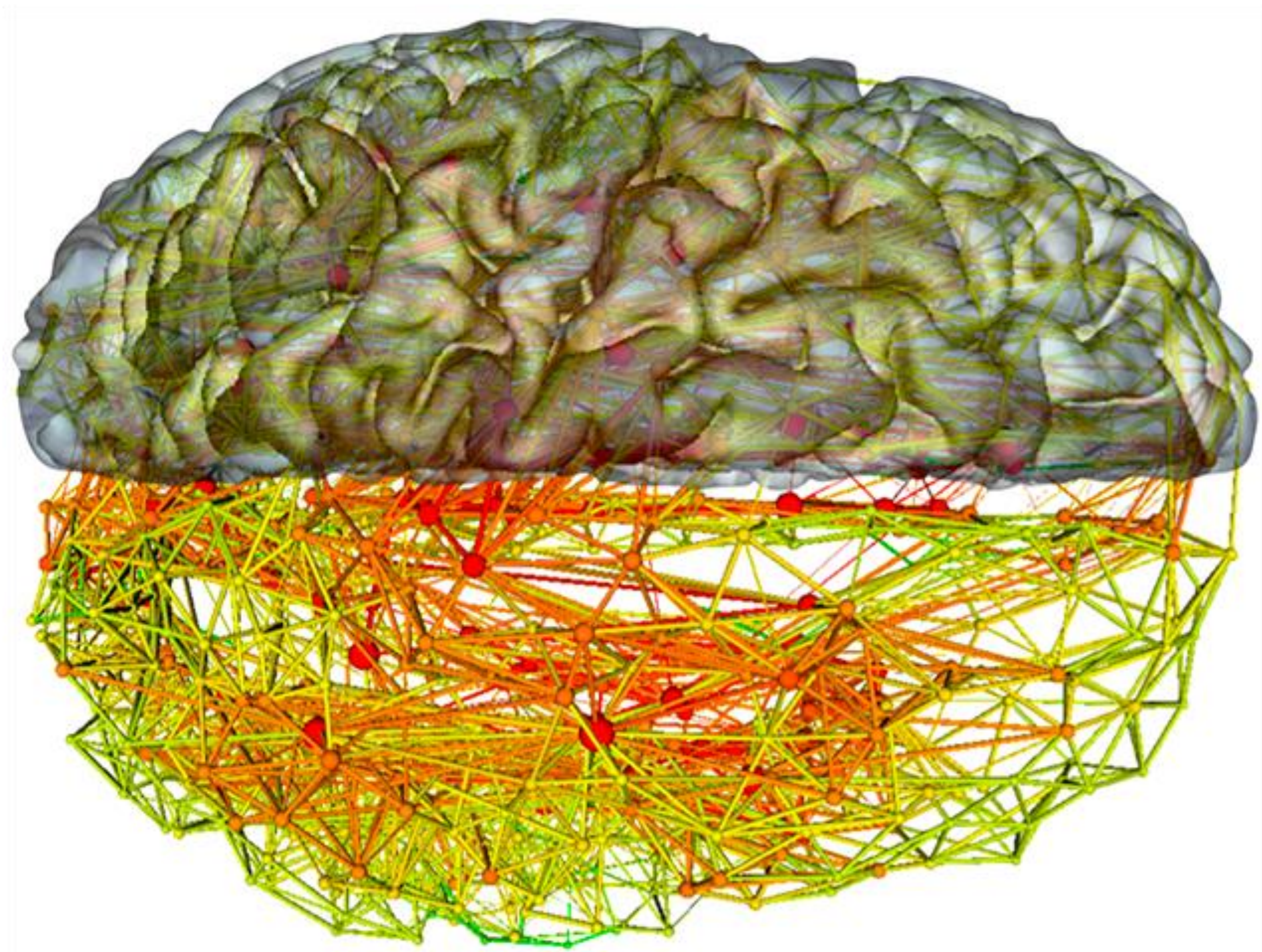
Why graphs? Model of many real-world systems

Interactions of elementary particles

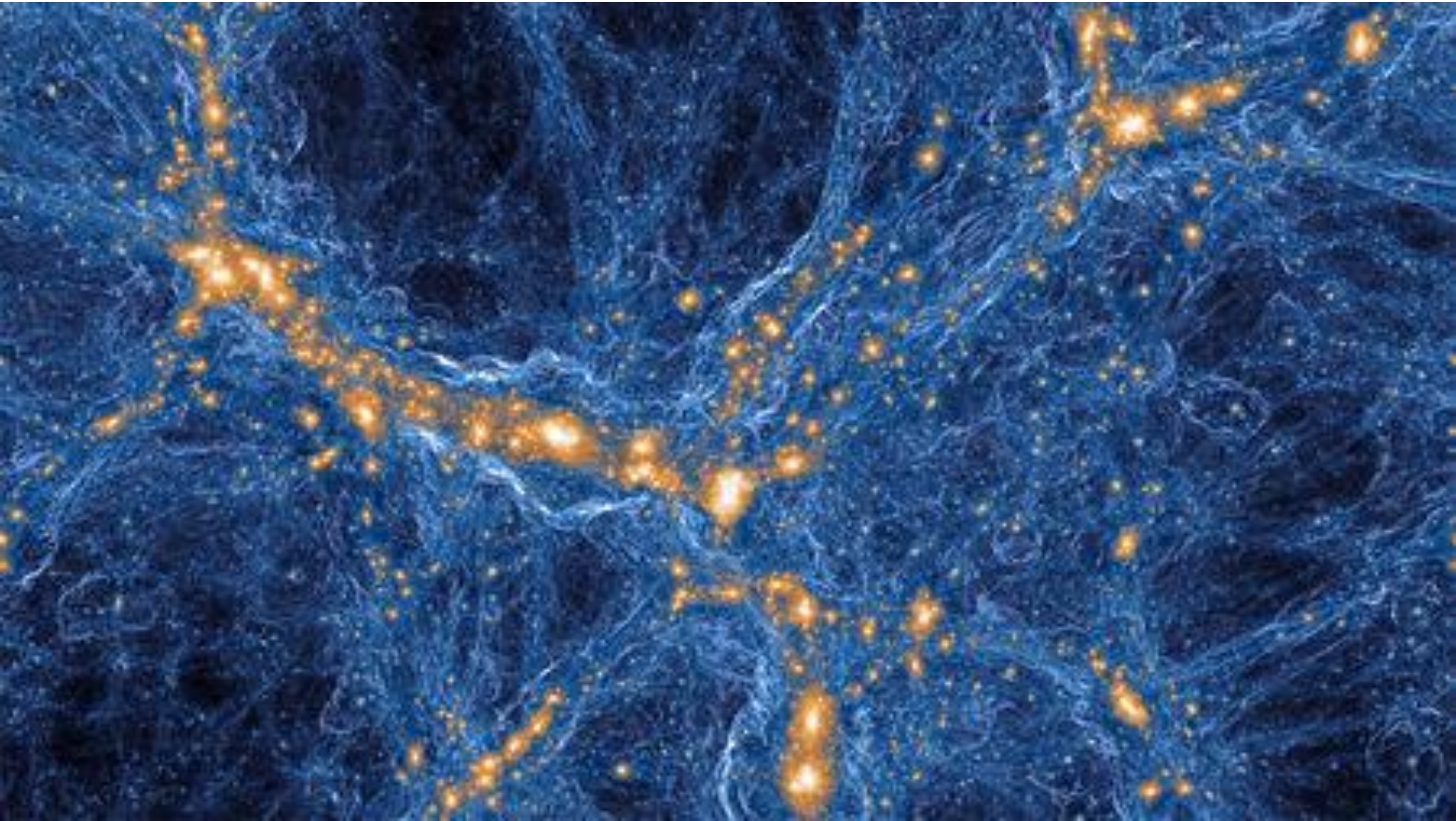


Why graphs? Model of many real-world systems

Neural network

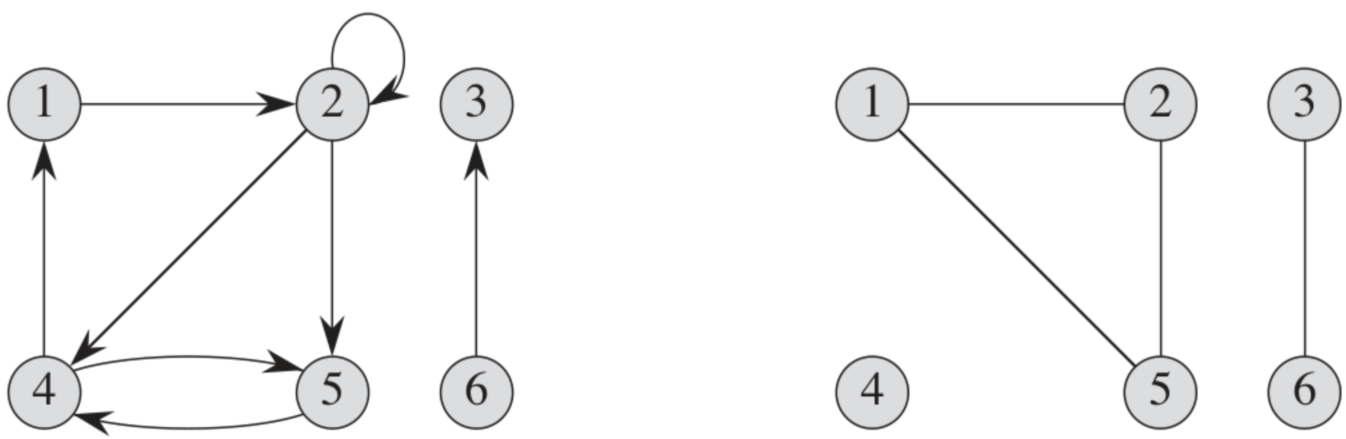


Why graphs? Cosmology



Section 2. Definitions

Direct and undirected diagrams



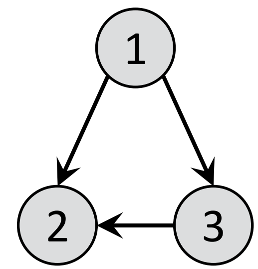
(a) (b)

Example: This is not a tree because it has cycles

Graph without cycle: is it always a tree?

Graph without cycles:

- Tree
- Directed acyclic graph (DAG)



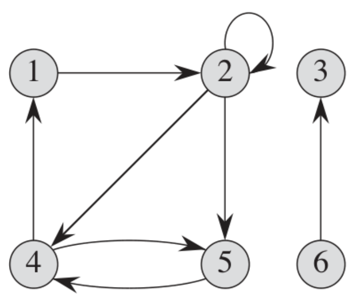
How to store a diagram?

G = (V, E)

V: A list of vertices

E: Edges stored in an adjacency list

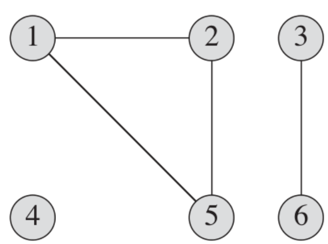
- For directed diagram: e.g. {[v1, v2], [v2, v3], …}



V = {1, 2, 3, 4, 5, 6}

E = {[1,2], [2,2], [2,4], [2,5],
[4,1], [4,5], [5,4], [6,3]}

- For undirected diagram: e.g {{v1, v2}, {v2, v3}, …}



V = {1, 2, 3, 4, 5, 6}

E = {{1,2}, {1,5}, {2,5}}

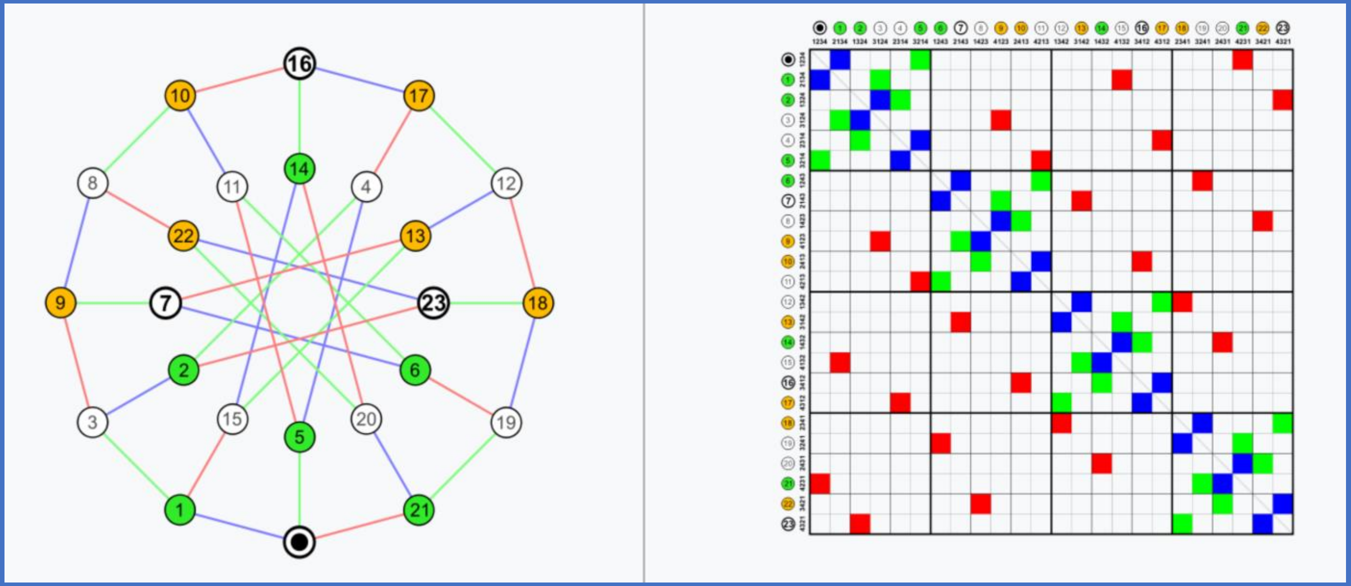
How to store a diagram?

$$G = (V, E)$$

V: A list of vertices

E: Edges stored in an adjacency list

Or adjacency matrix



Issue of adjacency list: not associated to edge.

Thus practically, additional workload for time complexity.

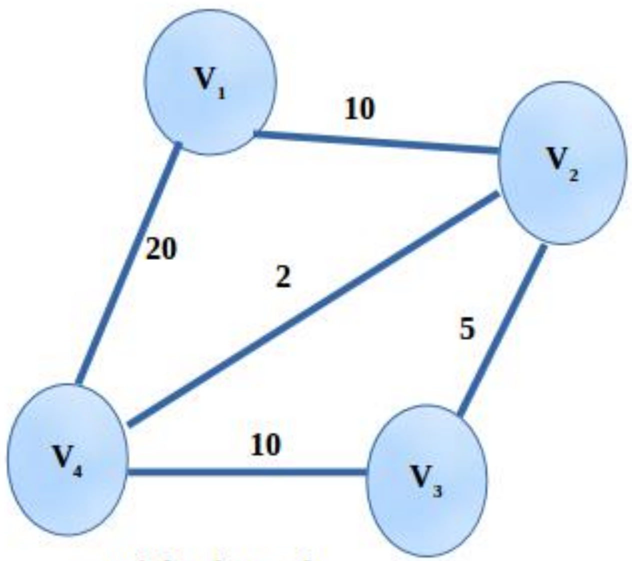
Practical implementation:

For each vertex, maintain a list of adjacent vertices.

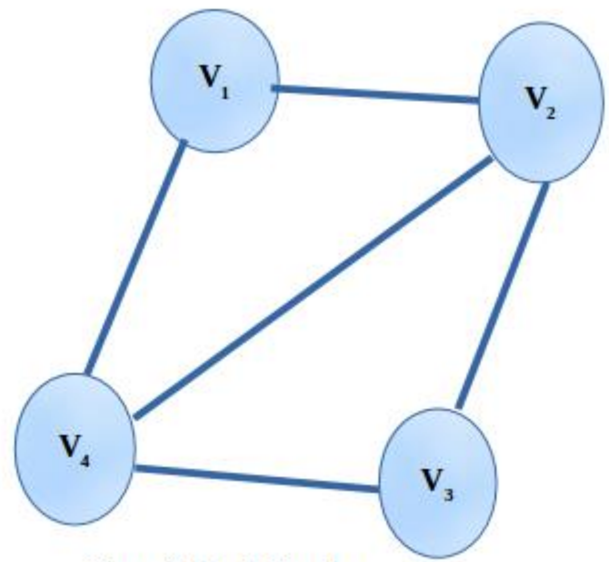
- Can use a list for each vertex: v.adj = [list of adjacent vertices]
- Can use a dict for the whole graph: adj[v] = [list of adjacent vertices]

(Or sometimes adjacent vertices calculated in real time.)

Weighted and unweighted graphs



Weighted Graph



Unweighted Graph

```python
class Graph:
    def __init__(self, V, E):
        self.vertices = V
        self.edges = E
        self.build_adj(E)

    def build_adj(self, E):
        self.adj_dict = {}
        for edge in E:
            if type(edge) == set: # undirected
                self.add_adj(list(edge))
                self.add_adj(list(edge)[::-1])
            else: # directed
                self.add_adj(edge)

    def adj(self, s):
        return self.adj_dict[s]

    def add_adj(self, edge):
        u, v = edge
        if u not in self.adj_dict:
            self.adj_dict[u] = [v]
        else:
            self.adj_dict[u].append(v)
```

```python
g = Graph(["a","s","d","f","z","x","c","v"], [{"a","z"},{"a","s"},{"s","x"},{"x","d"}
, {"x","c"},{"d","f"},{"c","f"},{"c","v"},{"f","v"}])
print(g.adj_dict)
```

```
{'a': ['z', 's'], 'z': ['a'], 's': ['a', 'x'], 'x': ['s', 'd', 'c'], 'd': ['x', 'f'], 'c': ['x', 'f', 'v'], 'f': ['d', 'c'
, 'v'], 'v': ['c', 'f']}
```

Section 3: Breadth-First Search (BFS)

```python
from collections import deque

    def BFS_simple(self, s):
        visited = {s}
        frontier = deque(s)
        while frontier:
            u = frontier.pop()
            print(u)
            for v in self.adj(u):
                if v not in visited:
                    visited.add(v)
                    frontier.appendleft(v)
```

Why called “simple”? See some realistic cases of problem solving.

Given an unweighted graph G, and a given vertex V,

how to find the shortest path from anywhere to V?

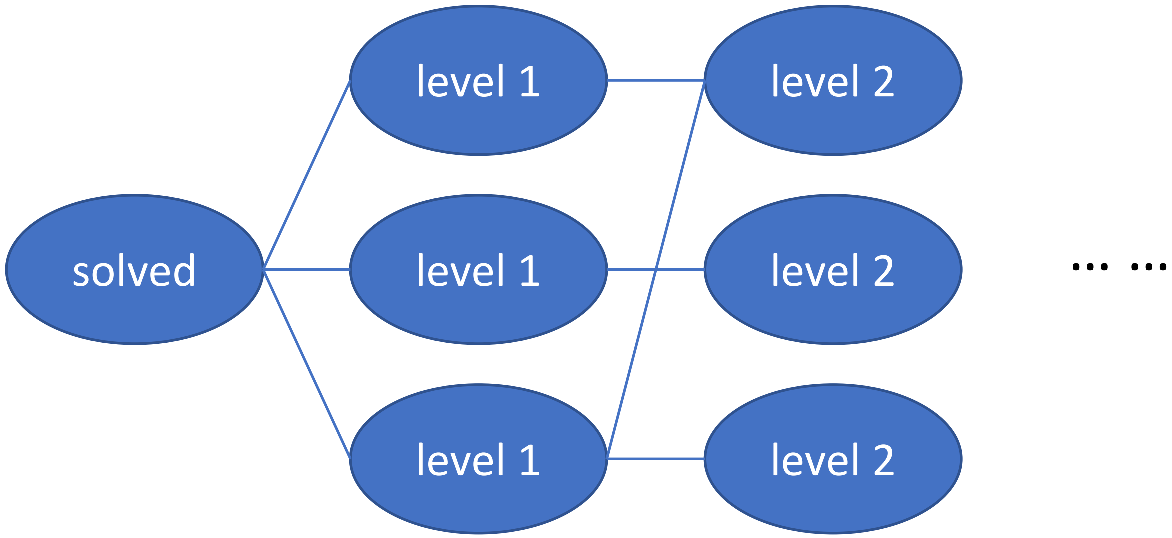
For example: How to solve a rank-2 magic cube from any state?

Given an unweighted graph G, and a given vertex V,

how to find the shortest path from anywhere to V?

For example: How to solve a rank-2 magic cube from any state?

Note: $(\#\text{ states}) = \frac{8! \times 3^8}{24 \times 3} = 3{,}674{,}160$: already lots of states!



Idea: first search for level 1, then level 2, …

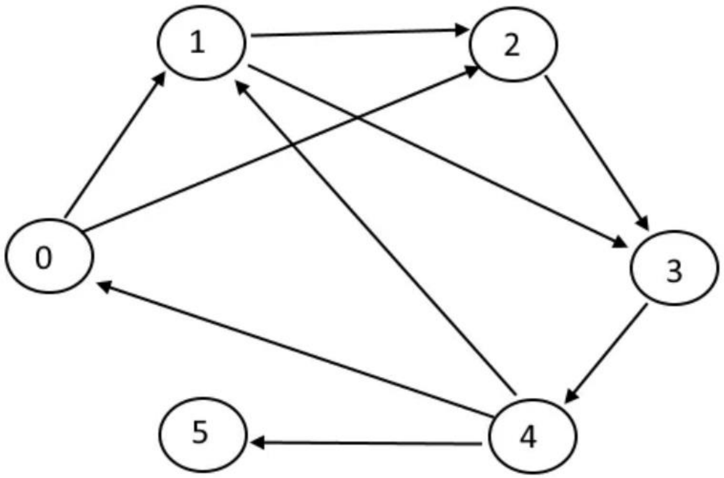
Note: store visited to avoid cycles. Otherwise will not end.

Breadth first search on a graph

Example: start from vertex 0.

Find shortest path for all vertices.

"Find": by
(1) Labelling depth
(2) Labelling parents

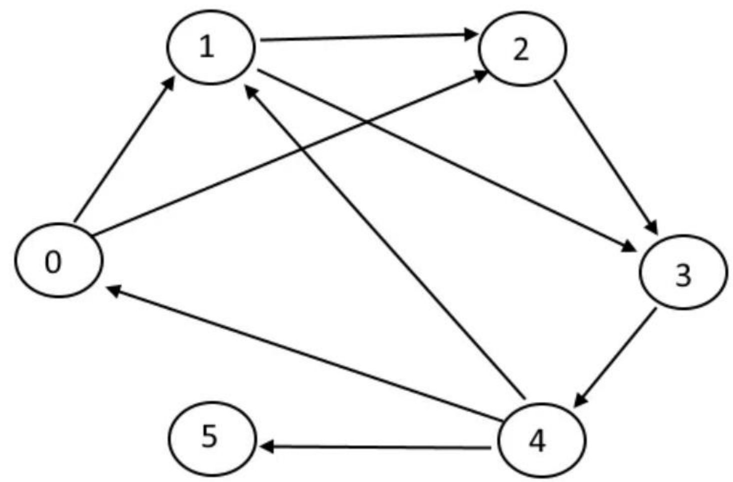


Breadth first search on a graph

Two usages of level:
(1) Visited flag
(2) Level

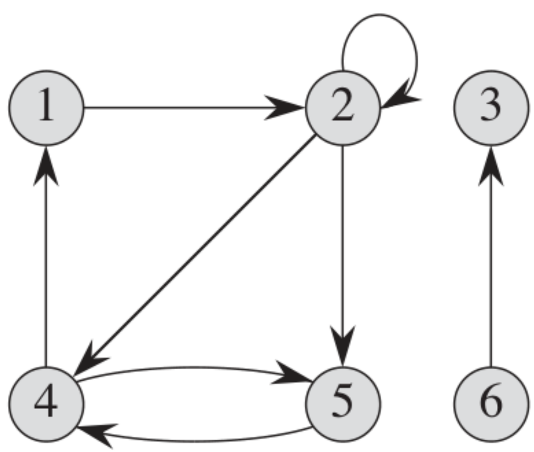
Q: how to realize without dict?

```
def BFS(self, s):
    parent = {s: None}
    level = {s: 0}
    frontier = [s]
    level_cnt = 0
    while frontier:
        level_cnt += 1
        next = []
        for u in frontier:
            for v in self.adj(u):
                if v not in level:
                    level[v] = level_cnt
                    parent[v] = u
                    next.append(v)
        frontier = next
    return [parent, level]
```



Disconnected parts?

1. For the current problem of single source shortest path: set to infinity
2. In general: Can run in a loop to visit all disconnected parts



Shortest path from v to the starting point:

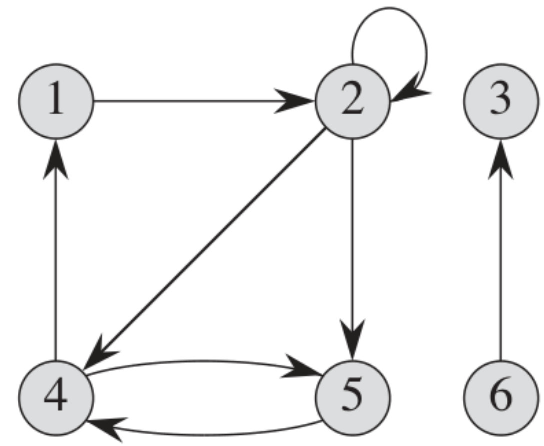
v $\leftarrow$ parent[v] $\leftarrow$ parent[parent[v]] $\leftarrow \cdots \cdots$

Length of the shortest path: level[v]

Section 4. Depth-First Search (DFS)

Depth first search for a vertex

```python
def DFS_visit(self, s, parent = None):
    if parent is None:
        parent = {s: None}
    for u in self.adj(s):
        if u not in parent:
            parent[u] = s
            self.DFS_visit(u, parent)
    return parent
```



$_1(\; _2(\; \cancel{2} \; _4(\; \cancel{1} \; _5(\; \cancel{4} \;)_5 \;)_4 \; \cancel{5} \;)_2 \;)_1$

$\cancel{2}$

$3(\;)_3$

$\cancel{4} \; \cancel{5}$

$_6(\; \cancel{3} \;)_6$

Depth first search for a graph

```python
def DFS(self):
    parent = {}
    for s in self.vertices:
        if s not in parent:
            parent[s] = None
            self.DFS_visit(s, parent)
    return parent
```

Edge classification (DFS starting from s)

- **Tree edge** (generated by parent pointers)
- **Forward edge** (pointing from ancestor to descendant，孙)
- **Back edge** (pointing from descendant to ancestor，爷)
- **Cross edge** (connecting distinct subtrees，表 or no relation)

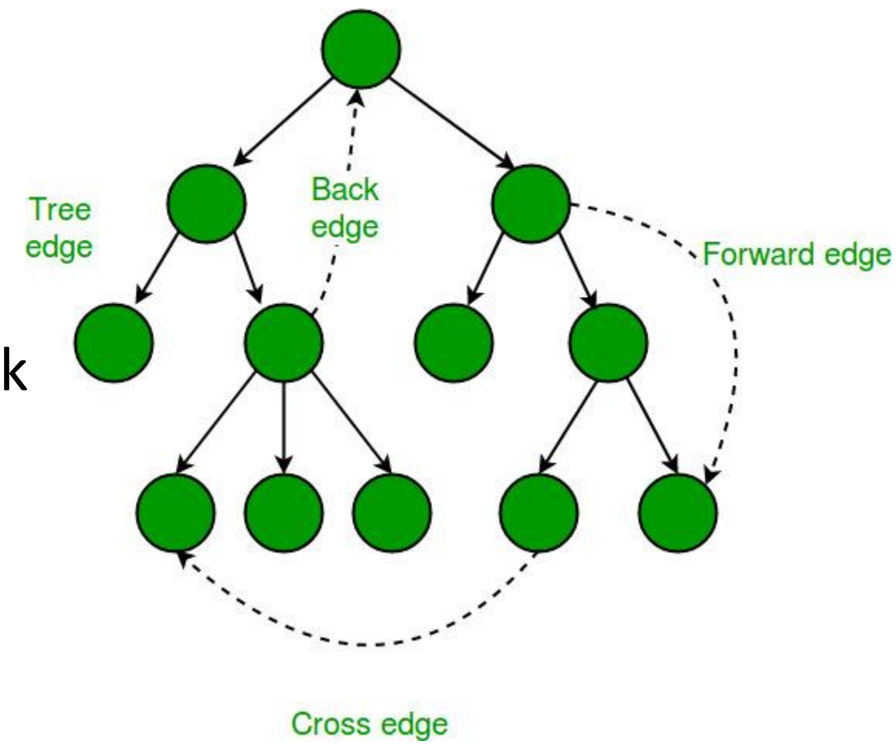
Cycle detection:

In DFS: back edge $\Leftrightarrow$ cycle

How to?

keep an explicit visit stack

test if pointing to edge in stack

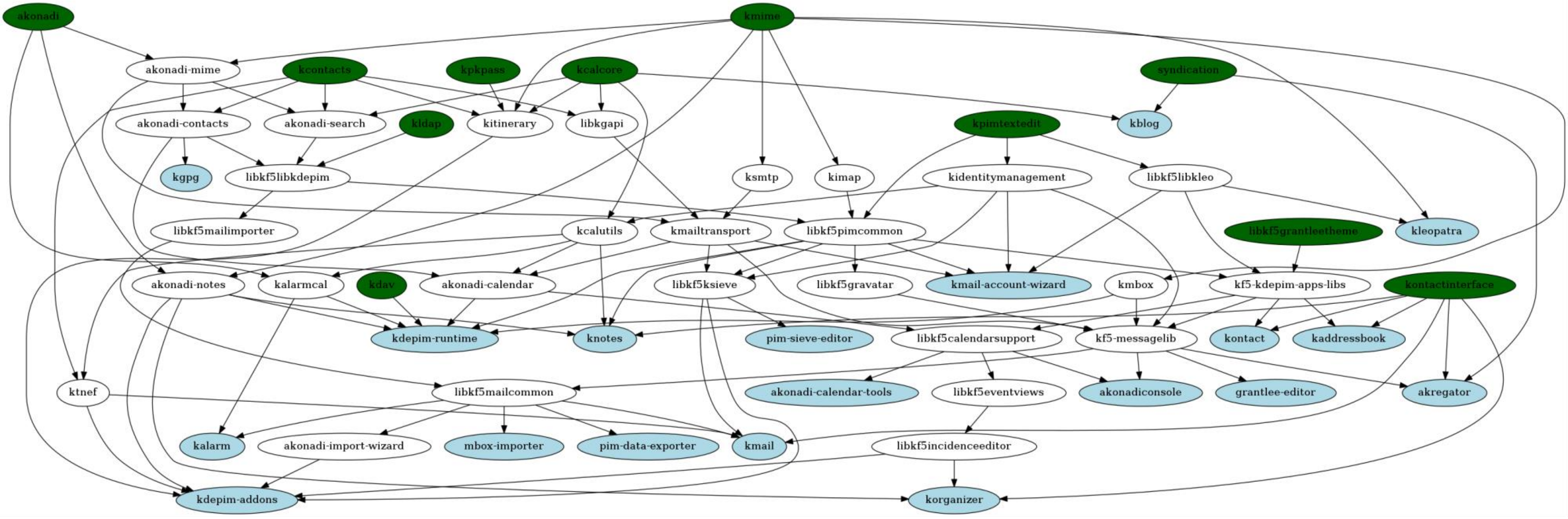


BFS vs DFS (each simplest version):

- Property differences:
  - BFS can find shortest path; DFS cannot.
  - DFS can find back edge (cycle); BFS cannot (why?)
  - DFS can do topological sort (next slide); BFS cannot
- Performance differences:
  - If some solutions are close to root, BFS
  - If solutions are deep but frequent, DFS

Section 5. Topological Sort

KDE dependency

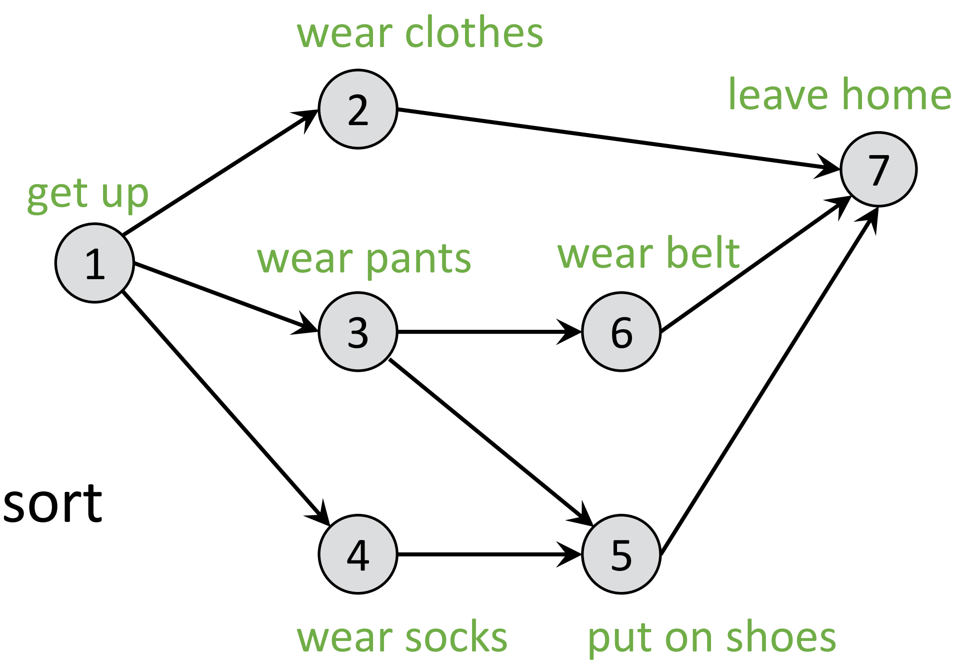


A directed acyclic graph (DAG) can represent order of doing things

If (u,v) is an edge, then u must happen before v

Example:

- Dress up
- Classes with prerequisites
- Software dependences



From DAG to order: topological sort

How to plan what to do from the diagram?

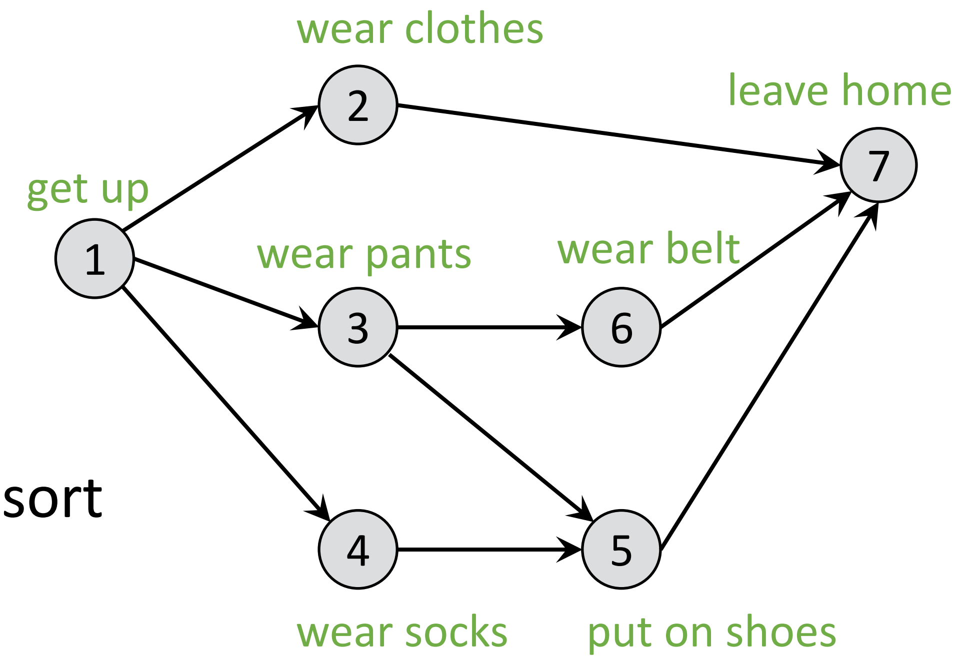
Job scheduling

A directed acyclic graph (DAG) can represent order of doing things

If (u,v) is an edge, then u must happen before v

Example:

- Dress up
- Classes with prerequisites
- Software dependences



From DAG to order: topological sort

Topological sort:

$_1(\ _2(\ _7()_7\ )_2\ _3(_5()_5\ _6()_6\ )_3\ _4(\quad)_4)_1$

DFS

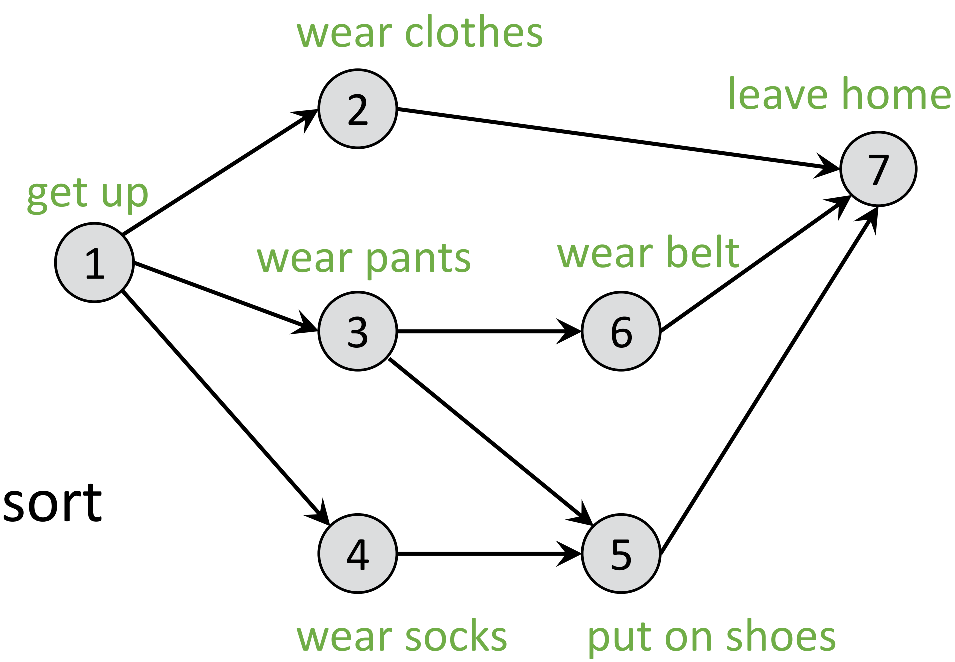
Job scheduling

A directed acyclic graph (DAG) can represent order of doing things

If (u,v) is an edge, then u must happen before v

Example:

- Dress up
- Classes with prerequisites
- Software dependences



From DAG to order: topological sort

Topological sort:

${}_1(\ {}_2(\ {}_7()_7\ )_2\ {}_3({}_5()_5\ {}_6()_6\ )_3\ {}_4(\quad)_4)_1$

DFS: Reversed order of exit is topological sort: 1 4 3 6 5 2 7

Why this “magic” can work?

DFS: If there are multiple arrows pointing to a vertex (e.g. 7) (i.e. it has to wait multiple things (e.g. 2 5 6) before being done), then DFS make sure to visit it in the stack of the first thing (e.g. 2).

When 7 exits, we are sure that none of 2, 5, 6 has exited.

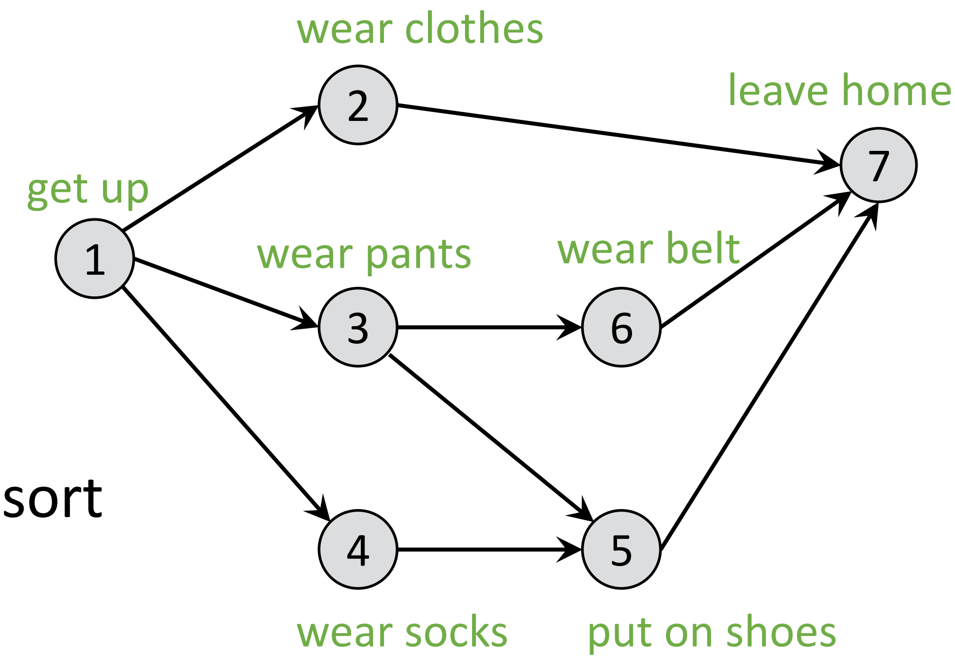
Then by reversed order, all prerequisite are done.

Job scheduling

) can represent order of doing things

efore v

cal sort



Topological sort:

$_1(\ _2(\ _7()_7\ )_2\ _3(_5()_5\ _6()_6\ )_3\ _4(\ \ )_4)_1$

DFS: Reversed order of exit is topological sort: 1 4 3 6 5 2 7

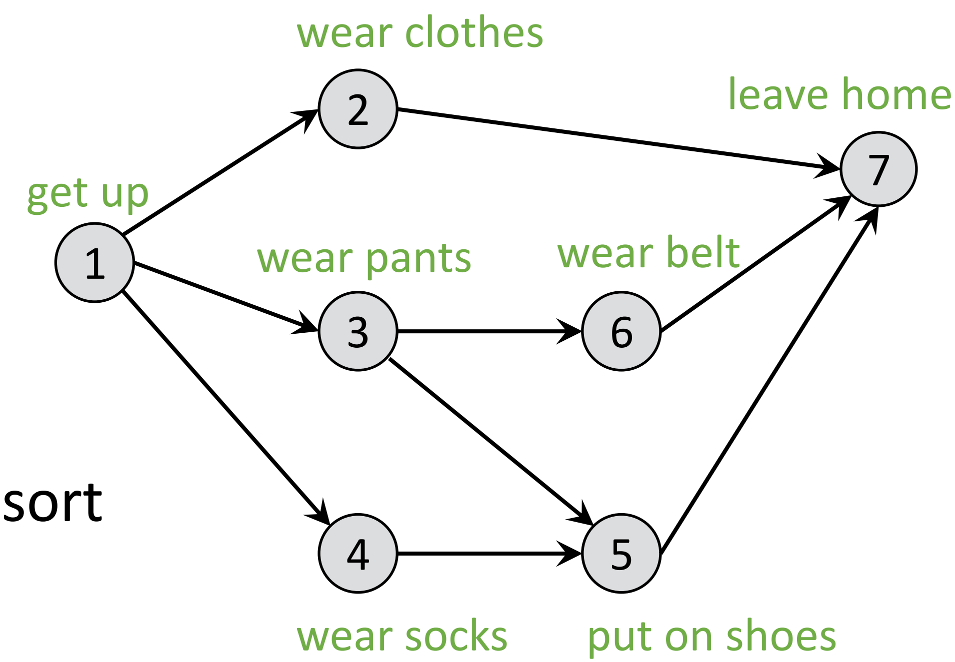
A directed acyclic graph (DAG) can represent order of doing things

If (u,v) is an edge, then u must happen before v

More generally, start from any node, do DFS and print exit order

If the whole graph is not visited, start from another node and to DFS until whole graph is visited.

al sort



Topological sort:

$_1(\ _2(\ _7()_7\ )_2\ _3(_5()_5\ _6()_6\ )_3\ _4(\ \ )_4)_1$

DFS: Reversed order of exit is topological sort: 1 4 3 6 5 2 7

Another algorithm for topological sort: Kahn’s algorithm

Idea: Repeatedly find nodes without incoming edges, and remove them

```
def Kahn(self):
    # Pseudo code from https://en.wikipedia.org/wiki/Topological_sorting
    # Did not optimize. Use adj_dict should be better
    L = []
    edges = self.edges[:] # make a copy since we will remove edges
    nodes_with_incoming_edges = [e[1] for e in edges]
    S = [v for v in self.vertices if v not in nodes_with_incoming_edges]
    while S:
        n = S.pop()
        L.append(n)
        edges_from_n = [e for e in edges if e[0] == n]
        nodes_with_edges_from_n = [e[1] for e in edges_from_n]
        for m in nodes_with_edges_from_n:
            edges = [e for e in edges if e != [n, m]]
            if m not in [e[1] for e in edges]:
                S.append(m)
    return "Graph has at least one cycle." if edges else L
```

Both algorithms: $O(|V|+|E|)$.

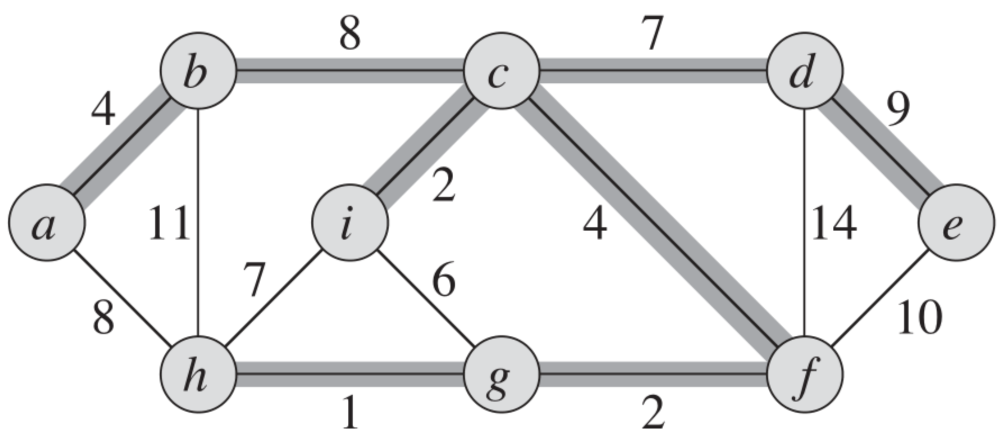
Why do we care about topological sort?

(1) Of its own importance

(2) Dynamic programming

Section 6. Minimum Spanning Tree

Minimum Spanning Tree



For weighted & undirected diagram,

find a subset of edges to connect all vertices with minimal total weight

GENERIC-MST(G, w)

```
A = ∅
while A does not form a spanning tree
    find an edge (u, v) that is safe for A
    A = A ∪ {(u, v)}
return A
```

Safe: A U {(u, v)} is still a subset of a minimum spanning tree.

How to find an edge safe for A?

- Kruskal: add edges to merge sub-trees
- Prim: add best vertices to one tree

Kruskal: add smallest edges & merge trees

MST-KRUSKAL(G, w) # G: graph, w: weight

$A = \emptyset$ # The set of edges that finally makes the MST

for each vertex $v \in G.V$

  MAKE-SET(v) # make a tree for each vertex

sort the edges of $G.E$ into nondecreasing order by weight w # greedy

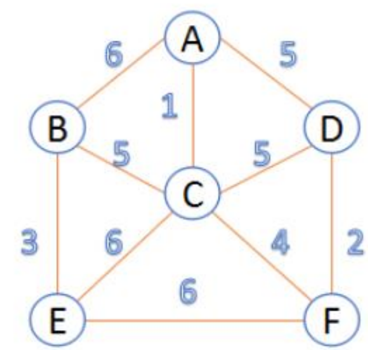
for each edge $(u, v) \in G.E$, taken in nondecreasing order by weight

  if FIND-SET$(u) \neq$ FIND-SET(v) # if same tree, will form loop

    $A = A \cup \{(u, v)\}$

    UNION(u, v) # merge two trees

return A



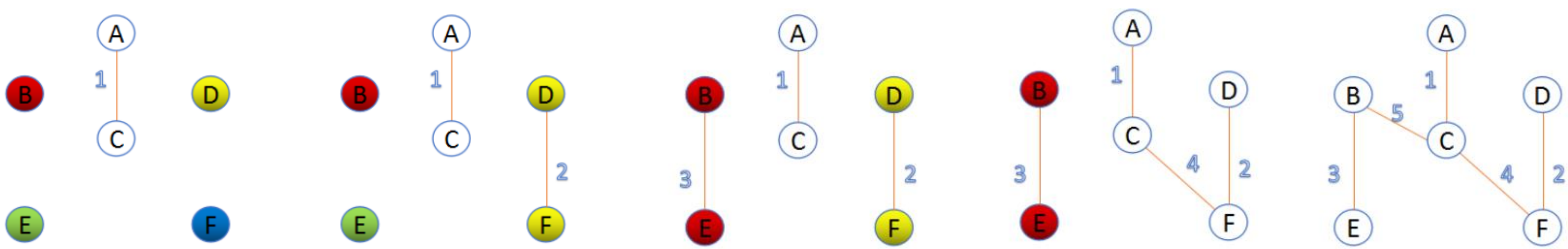


Image credit: https://blog.csdn.net/luoshixian099/article/details/51908175

Prim: add vertex with minimal distance to *one tree*, until tree spanning graph

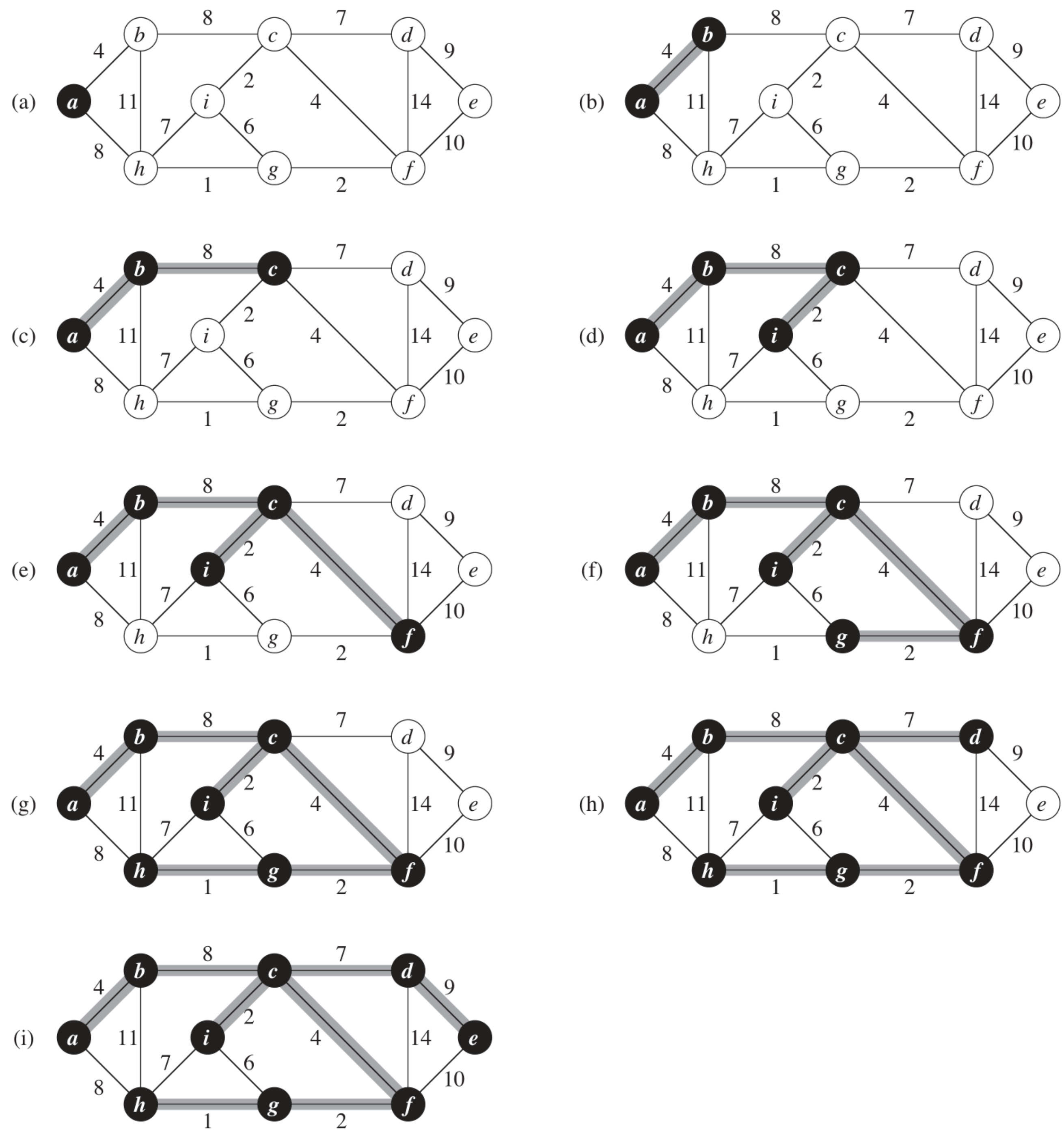
```
MST-PRIM(G, w, r)                      # r: any given root vertex
for each u ∈ G.V
    u.key = ∞                          # key: minimal distance to the existing tree
    u.π = NIL                          # π: parent of u in the tree
r.key = 0
Q = G.V                                # Q: vertices to be added, min-priority queue
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key   # update distances to tree (also update Q)
            v.π = u
            v.key = w(u, v)
```

minimal distance to the so far constructed tree

Time complexity:

O(V * Extract-Min + E * Decrease-Key)

May use min-heap:
O(V lg V + E lg V)
Fibonacci heap:
O(V lg V + E)



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

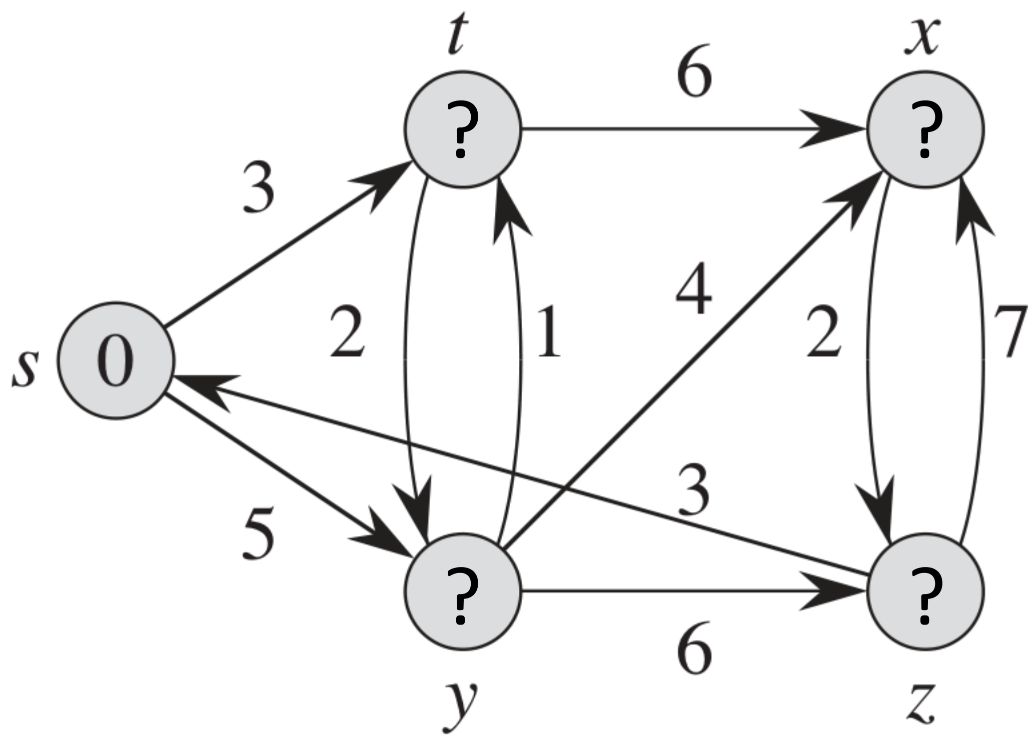
Section 7. Shortest path problem starting from one vertex

Problem: Starting from a vertex s.

Given the weights labelled on the graph,

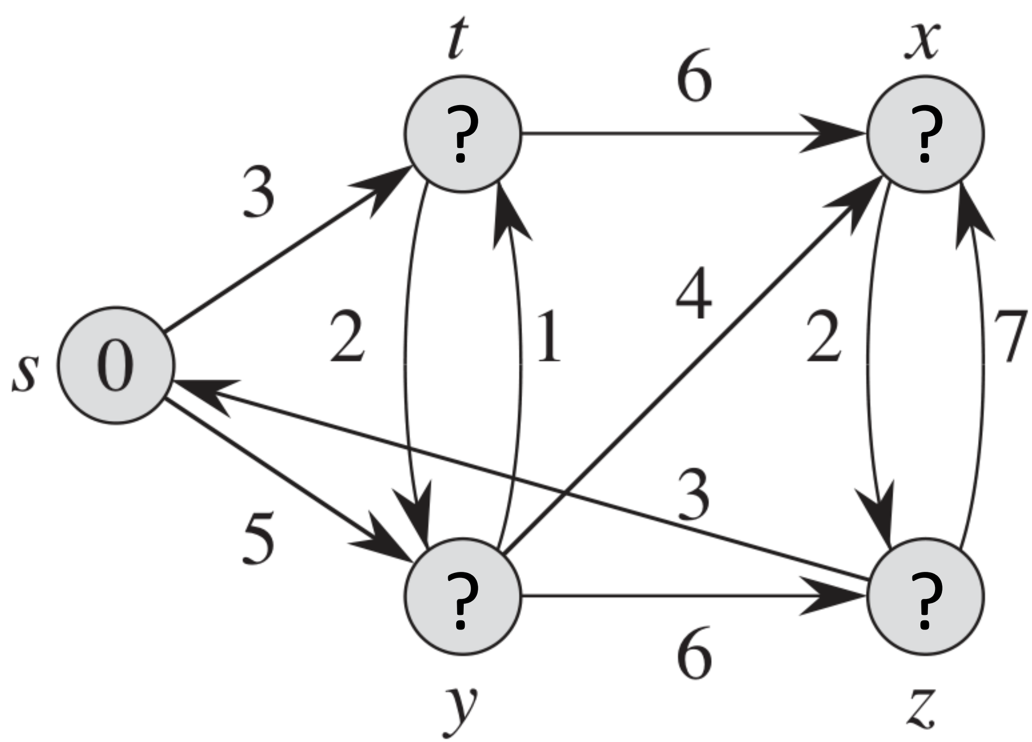
what's the shortest path from s to t, x, y, z?

from s to v:
δ(s, v)



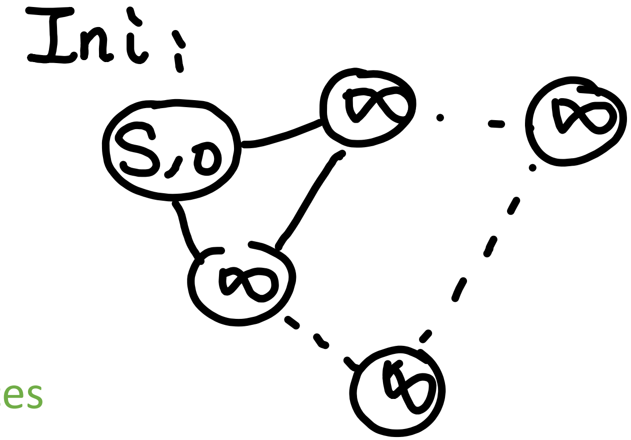
Problem: Starting from a vertex s.
Given the weights labelled on the graph,
what's the shortest path from s to t, x, y, z?

Naïve solution: enumerate all paths & compare.
Problem 1: cycles
Problem 2: exponential complexity
Can we do better?



The idea of relaxation:

Starting from the worst estimate



INITIALIZE-SINGLE-SOURCE(G, s)

for each vertex $v \in G.V$ # for all vertices
    $v.d = \infty$ # the best path found so far
    $v.\pi = \text{NIL}$ # the predecessor for the current best path
$s.d = 0$ # start from s, thus s has no distance to itself

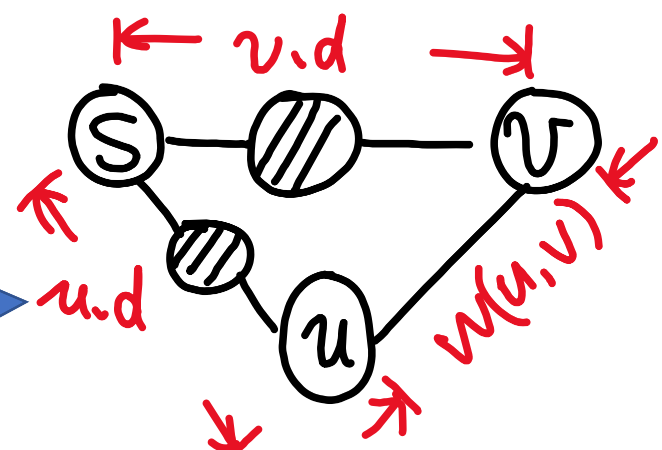
Try to improve the estimate: look at its neighbour, can it give us a better estimate?

RELAX(u, v, w)

if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$

Better: make the change
Otherwise: do nothing

Relaxation: If shorter:
Hey, I found a better way.
Let's update our knowledge.



The idea of relaxation:

Starting from the worst estimate

INITIALIZE-SINGLE-SOURCE(G, s)

for each vertex $v \in G.V$ # for all vertices
    $v.d = \infty$ # the best path found so far
    $v.\pi = \text{NIL}$ # the predecessor for the current best path
$s.d = 0$ # start from s, thus s has no distance to itself

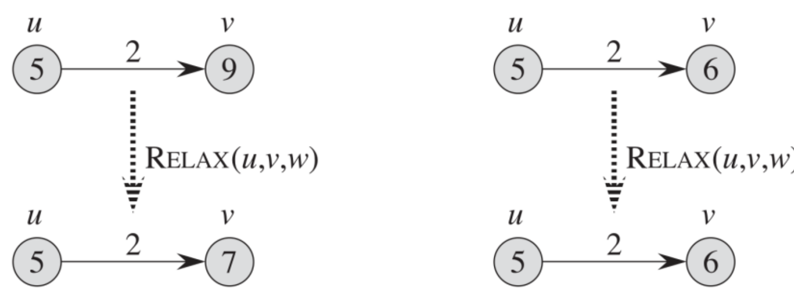
Try to improve the estimate: look at its neighbour, can it give us a better estimate?

RELAX(u, v, w)

if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$

Better: make the change

Otherwise: do nothing

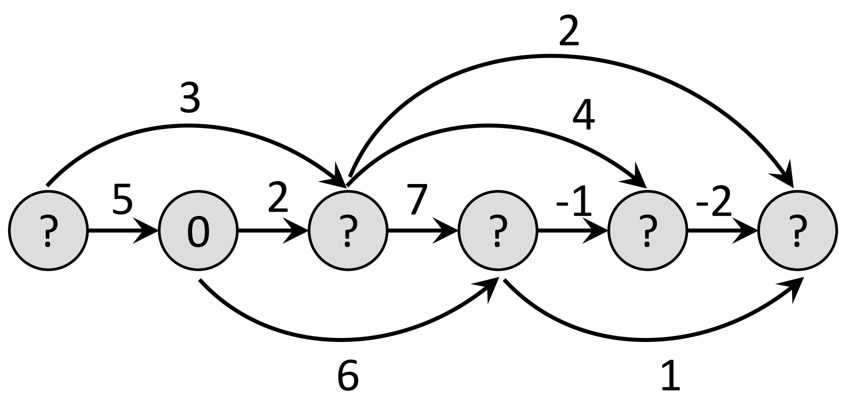


Idea: keep doing relaxation, until find shortest paths

Example: Shortest Path for Directed Acyclic Diagram (DAG)

Idea:

1. Topologically sort
2. Relax: for each vertex in topological order, relax all edges from this vertex

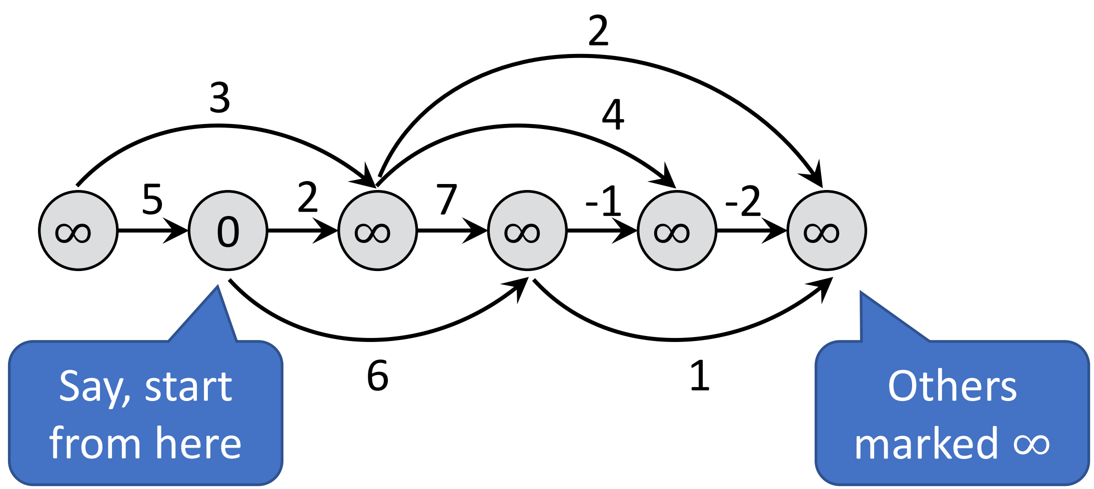


Result of topological sort

Example: Shortest Path for Directed Acyclic Diagram (DAG)

Idea:

1. Topologically sort
2. Relax: for each vertex in topological order, relax all edges from this vertex

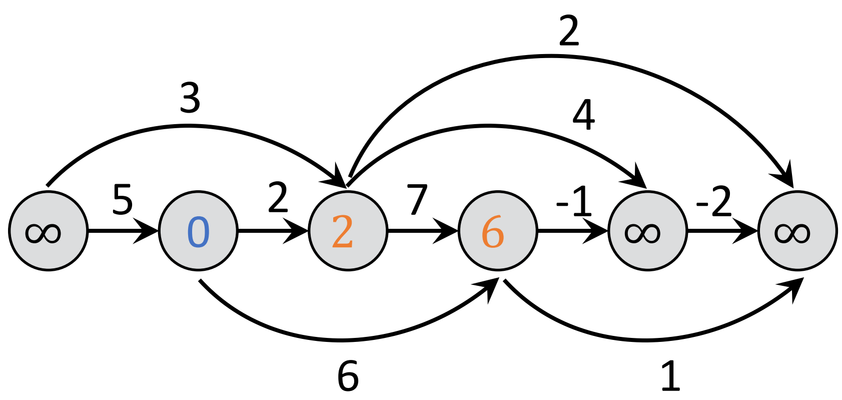


Init of relaxation

Example 1: Shortest Path for Directed Acyclic Diagram (DAG)

Idea:

1. Topologically sort
2. Relax: for each vertex in topological order, relax all edges from this vertex

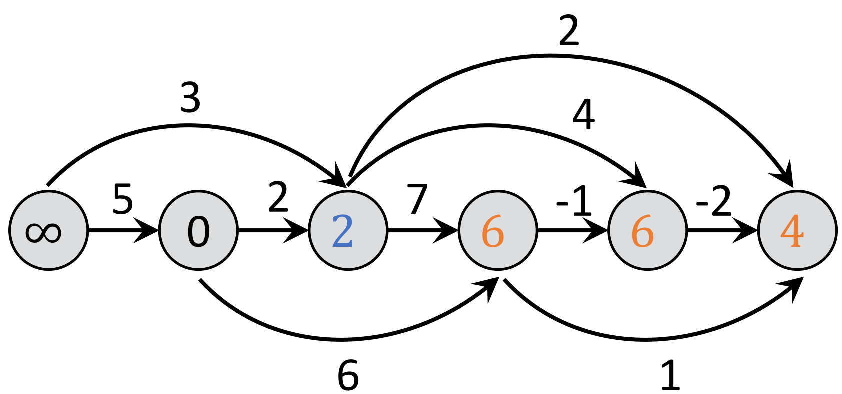


Relax the edges starting from s

Example 1: Shortest Path for Directed Acyclic Diagram (DAG)

Idea:

1. Topologically sort
2. Relax: for each vertex in topological order, relax all edges from this vertex

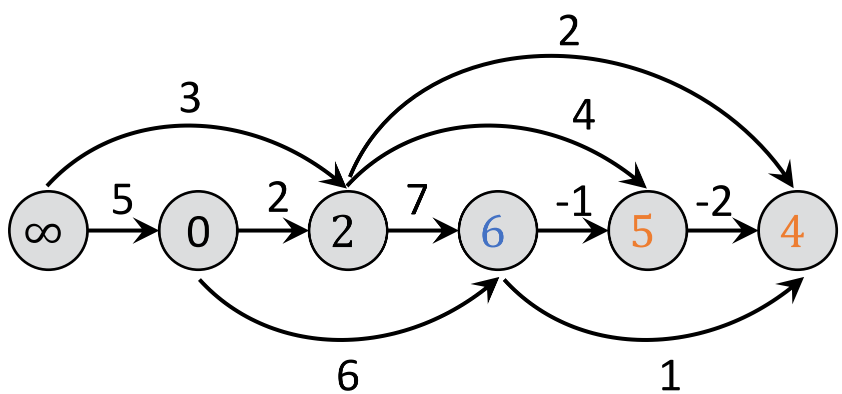


Relax the next vertex

Example 1: Shortest Path for Directed Acyclic Diagram (DAG)

Idea:

1. Topologically sort
2. Relax: for each vertex in topological order, relax all edges from this vertex

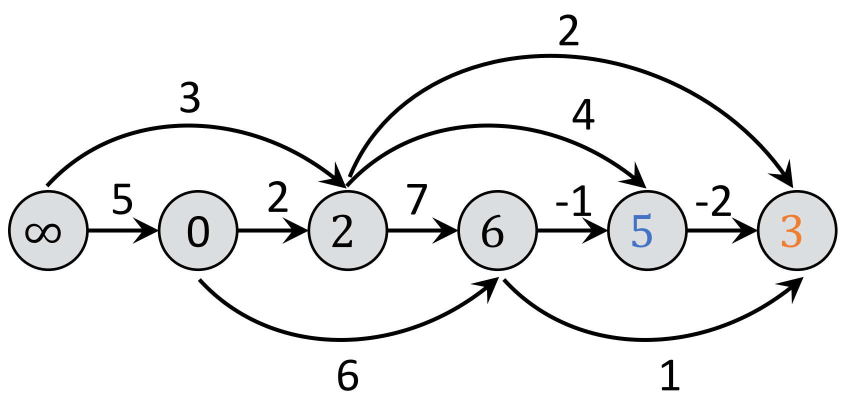


Relax the next vertex

Example 1: Shortest Path for Directed Acyclic Diagram (DAG)

Idea:

1. Topologically sort
2. Relax: for each vertex in topological order, relax all edges from this vertex

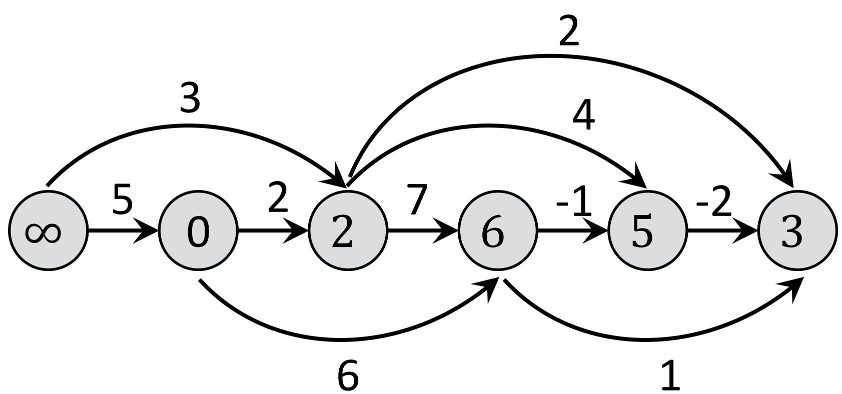


Relax the next vertex, until reaches the last one

Example 1: Shortest Path for Directed Acyclic Diagram (DAG)

Idea:

1. Topologically sort
2. Relax: for each vertex in topological order, relax all edges from this vertex



Time complexity: O(V+E)

This algorithm is related to dynamic programming (more later)

Example 2: Shortest Path without negative weight (Dijkstra)

Idea: similar to Prim algorithm of minimal spanning tree

Find easiest-to-get places from s first.

Then next-easiest-to-get places, …

If no negative edges, already explored part won't change.

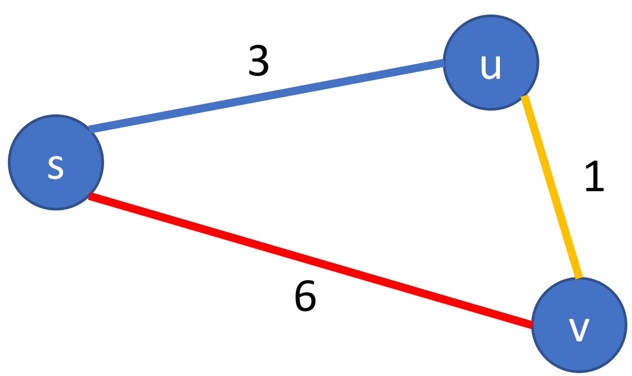
Example 2: Shortest Path without negative weight (Dijkstra)

d(s) = 0
d(ow) = ∞

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)      # Init relaxation starting from s
S = Ø                               # Set of vertices that we know the shortest path
Q = G.V                             # All vertices
while Q ≠ Ø
    u = EXTRACT-MIN(Q)  { # 1. Select u by minimal d(u) (Greedy)
                          # 2. Delete u from Q
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)              # Relax every vertices coming out of u
```

d(u) is best known (so far) distance to s

Will select s in its first run



Example 2: Shortest Path without negative weight (Dijkstra)

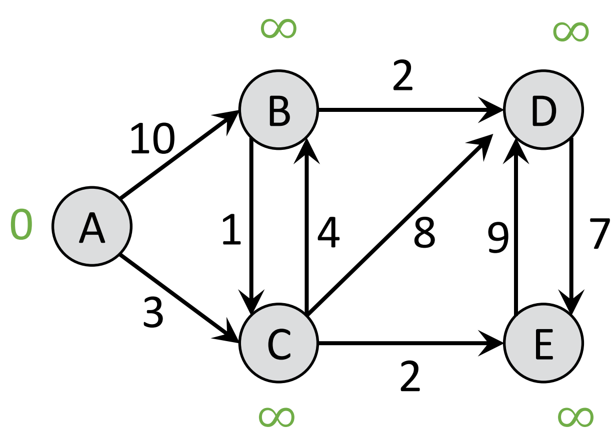
d(s) = 0
d(ow) = ∞

DIJKSTRA(G, w, s)

```
INITIALIZE-SINGLE-SOURCE(G, s)      # Init relaxation starting from s
S = Ø                               # Set of vertices that we know the shortest path
Q = G.V                             # All vertices
while Q ≠ Ø
    u = EXTRACT-MIN(Q)              { # 1. Select u by minimal d(u) (Greedy)
                                      # 2. Delete u from Q
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)              # Relax every vertices coming out of u
```

d(u) is best known (so far) distance to s

Will select s in its first run



Example 2: Shortest Path without negative weight (Dijkstra)

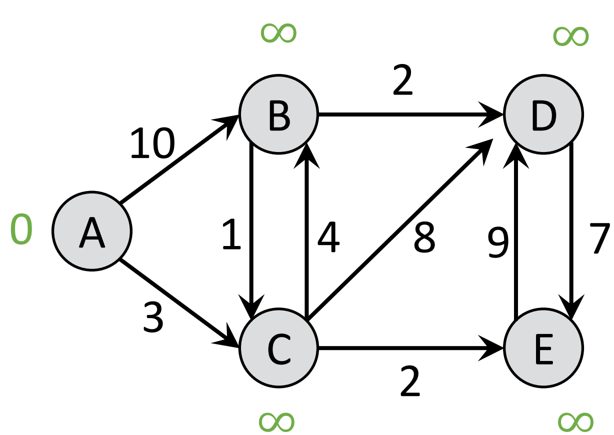
d(s) = 0
d(ow) = ∞

DIJKSTRA(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s) # Init relaxation starting from s
$S = \emptyset$ # Set of vertices that we know the shortest path
$Q = G.V$ # All vertices
while $Q \neq \emptyset$
    $u =$ EXTRACT-MIN(Q) { # 1. Select u by minimal d(u) (Greedy) # 2. Delete u from Q
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$
        RELAX(u, v, w) # Relax every vertices coming out of u

d(u) is best known (so far) distance to s

Will select s in its first run



Time complexity ?

Example 2: Shortest Path without negative weight (Dijkstra)

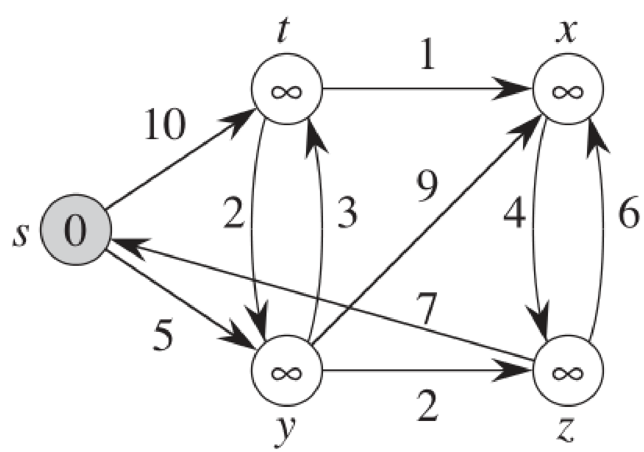
d(s) = 0
d(ow) = ∞

DIJKSTRA(G, w, s)

```
INITIALIZE-SINGLE-SOURCE(G, s)      # Init relaxation starting from s
S = ∅                               # Set of vertices that we know the shortest path
Q = G.V                             # All vertices
while Q ≠ ∅
    u = EXTRACT-MIN(Q)              # 1. Select u by minimal d(u) (Greedy)
                                    # 2. Delete u from Q
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)              # Relax every vertices coming out of u
```

d(u) is best known (so far) distance to s

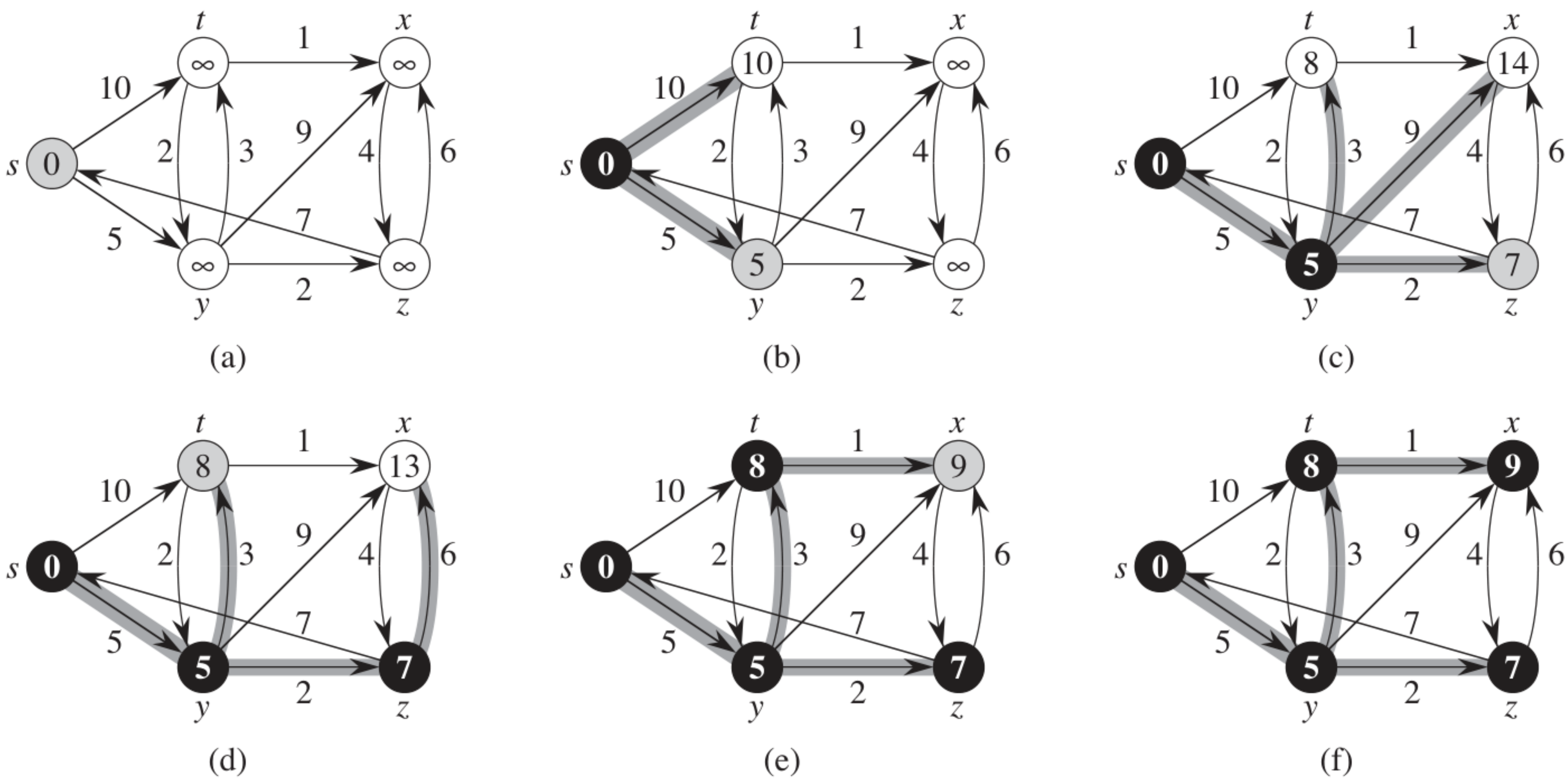
Will select s in its first run



Time complexity:

O(V * Extract-Min + E * Decrease-Key)

May use min-heap:
O(V lg V + E lg V)
Fibonacci heap:
O(V lg V + E)



Note: no negative edge →

vertex being relaxed won't change vertex already relaxed

More generally: With negative edges? Bellman-Ford

Use Dijkstra if possible. Use Bellman-Ford otherwise.

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)                # Just relax VE times: O(VE)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE                  # if it's still possible to relax,
return TRUE                           # report negative weight cycles
```

How this works? Roughly:

Assume no negative cycles, and assume

p = <v0, v1, v2, …, vn> is a shortest path with minimal number of edges.

Then first pass -> find <v0, v1>, second pass -> find <v0, v1, v2>, etc.

This algorithm is related to dynamic programming (more later)

Recap: Graph Related Algorithms

Visit: BFS, DFS

Minimal Spanning Tree

Shortest Path: for DAG, non-negative weight, general