CSE 421
Introduction to Algorithms

Lecture 3: Overview, Graph Search

O, o, Ω, Θ-notation intuition

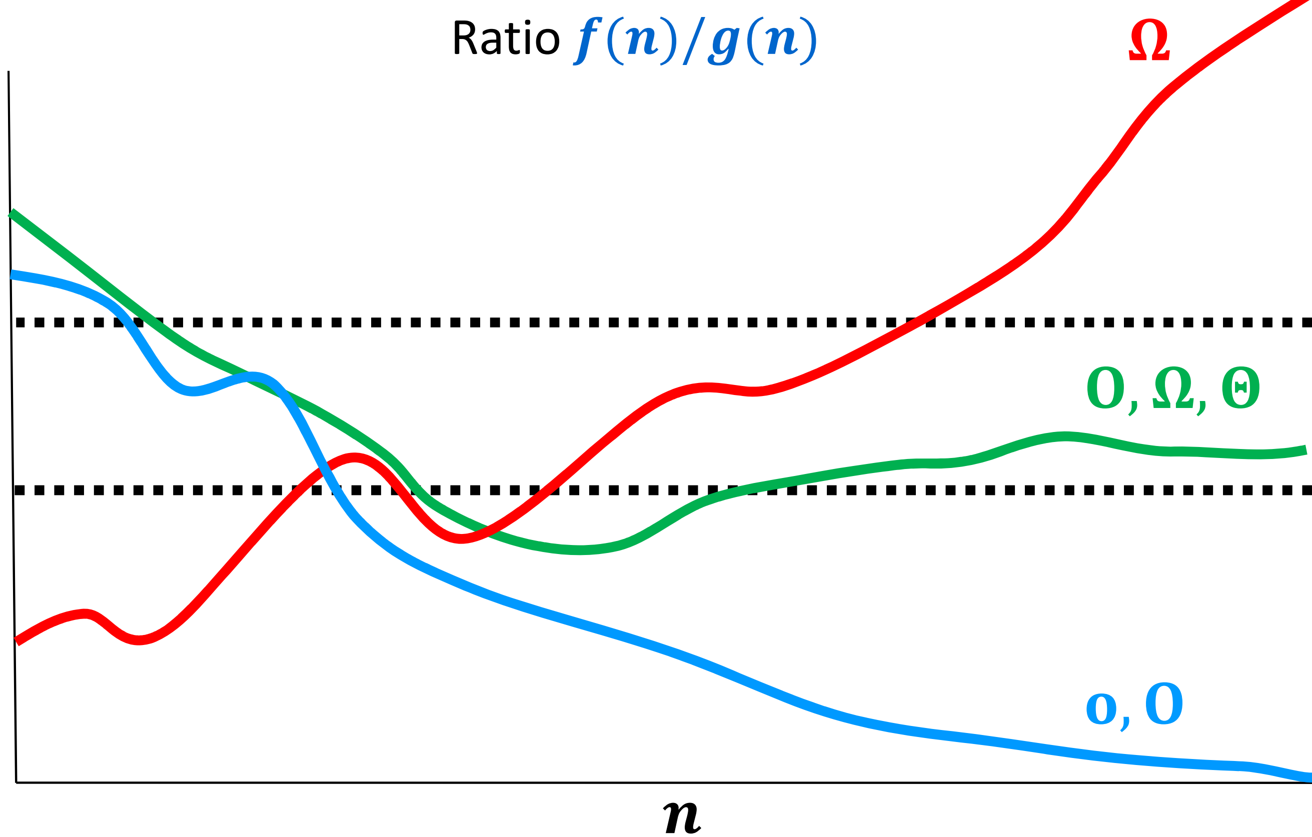
$f(n)$ is...

$O(g(n))$: ratio eventually below a line forever

$o(g(n))$: ratio goes to 0

$\Omega(g(n))$: ratio eventually above a line forever

$\Theta(g(n))$: both O and Ω



Introduction to Algorithms

- **Some representative problems**
  - Variety of techniques we'll cover
  - Seemingly small changes in a problem can require big changes in how we solve it

Some Representative Problems

Interval Scheduling:

- Single resource
- Reservation requests of form:

  "Can I reserve it from start time s to finish time f?"

$$s < f$$

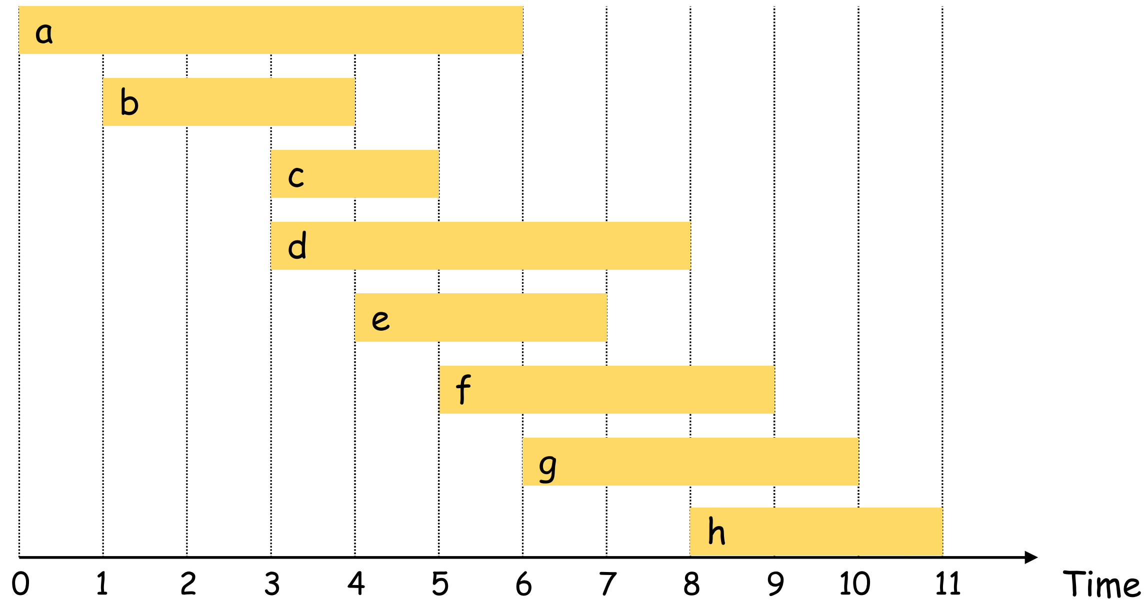
Interval Scheduling

Interval scheduling:

Input: set of jobs with start times and finish times

Goal: find maximum size subset of mutually compatible jobs.

jobs don't overlap



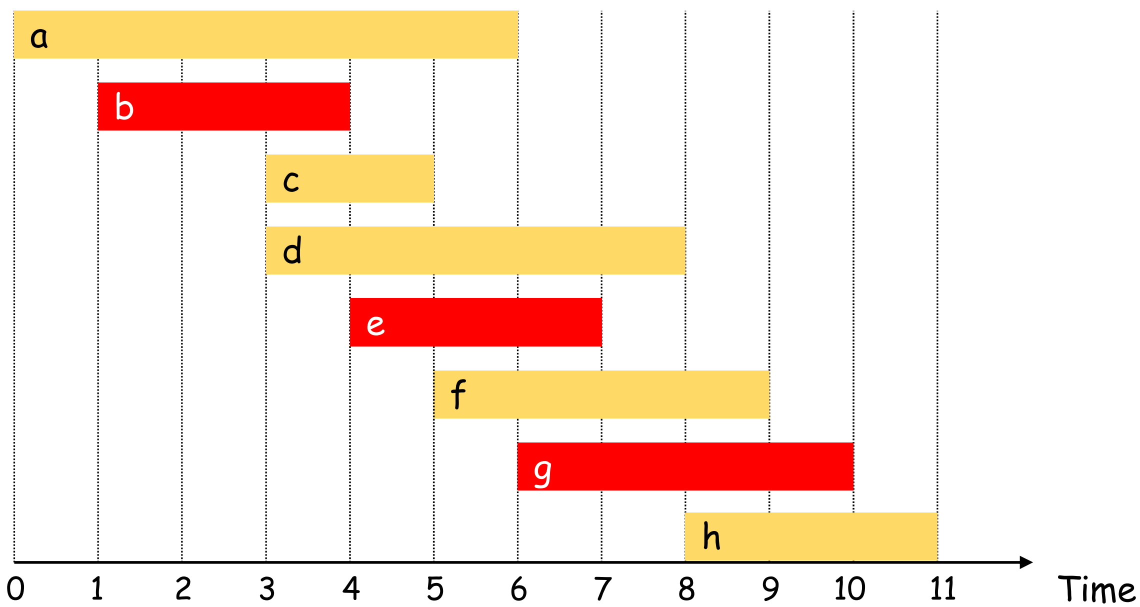
Interval Scheduling

Interval scheduling:

Input: set of jobs with start times and finish times

Goal: find maximum size subset of mutually compatible jobs.

jobs don't overlap



Interval Scheduling

- An optimal solution can be found using a “greedy algorithm”
  - Myopic kind of algorithm that seems to have no look-ahead
  - Greedy algorithms only work when the problem has a special kind of structure
  - When they do work they are typically very efficient

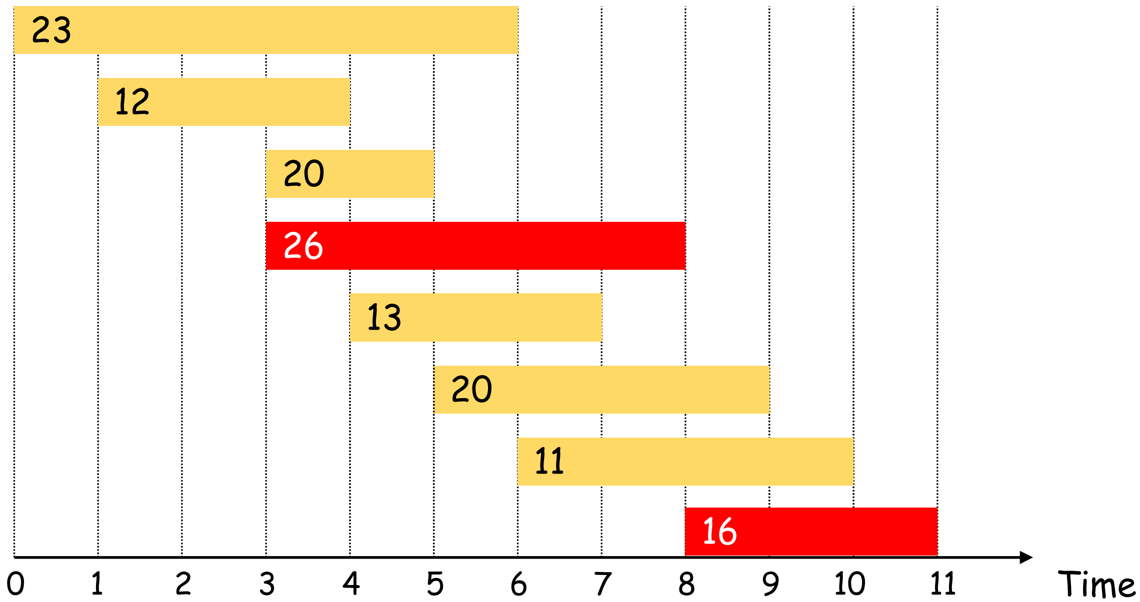
Weighted Interval Scheduling

- Same problem as interval scheduling except that each request i also has an associated value or weight w_i
  - w_i might be
    - amount of money we get from renting out the resource for that time period
    - amount of time the resource is being used

Weighted Interval Scheduling

Input: Set of jobs with start times, finish times, and weights

Goal: Find maximum weight subset of mutually compatible jobs.



Weighted Interval Scheduling

Ordinary interval scheduling is a special case of this problem

- Take all weights $\boldsymbol{w_i} = \mathbf{1}$

Problem is quite different though

- E.g. one weight might dwarf all others

"Greedy algorithms" don't work

Solution: "Dynamic Programming"

- builds up optimal solutions from a table of solutions to smaller problems

Bipartite Matching

A graph $G = (V, E)$ is bipartite iff

- Set V of vertices has two disjoint parts X and Y
- Every edge in E joins a vertex from X and a vertex from Y

Set $M \subseteq E$ is a matching in G iff no two edges in M share a vertex

Goal: Find a matching M in G of maximum size.

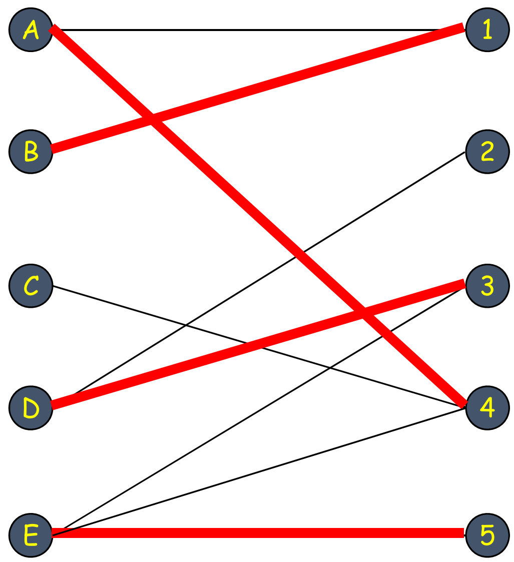
Differences from stable matching

- limited set of possible partners for each vertex
- sides may not be the same size
- no notion of stability; matching everything may be impossible.

Bipartite Matching

Input: Bipartite graph

Goal: Find maximum size matching.



Bipartite Matching

- Models assignment problems
  - $\boldsymbol{X}$ represents customers, $\boldsymbol{Y}$ represents salespeople
  - $\boldsymbol{X}$ represents professors, $\boldsymbol{Y}$ represents courses

- If $|\boldsymbol{X}| = |\boldsymbol{Y}| = \boldsymbol{n}$
  - $\boldsymbol{G}$ has perfect matching iff maximum matching has size $\boldsymbol{n}$

Solution: polynomial-time algorithm using “augmentation” technique

- Also used for solving more general class of network flow problems

Independent Set

Defn: For graph $G = (V, E)$ a set $I \subseteq V$ is independent iff no two nodes in I are joined by an edge

Input: Graph $G = (V, E)$

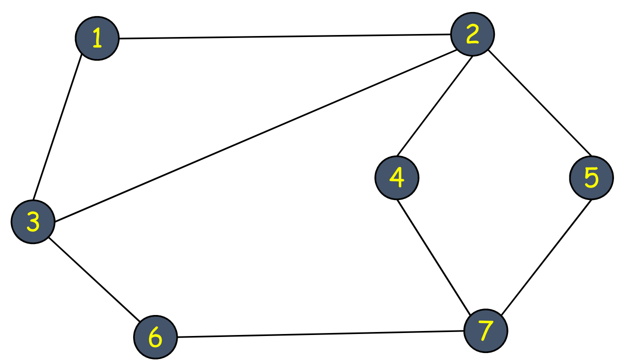
Goal: Find an independent set I in V of maximum possible size

- Models conflicts and mutual exclusion

Independent Set

Input: Graph.

Goal: Find a maximum size independent set.



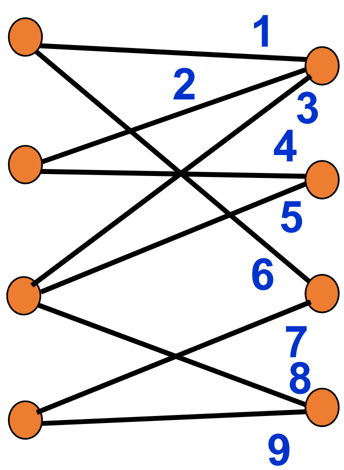
Independent Set

Generalizes

- **Interval Scheduling**
  - Vertices in the graph are the requests
  - Vertices are joined by an edge if they are **not** compatible

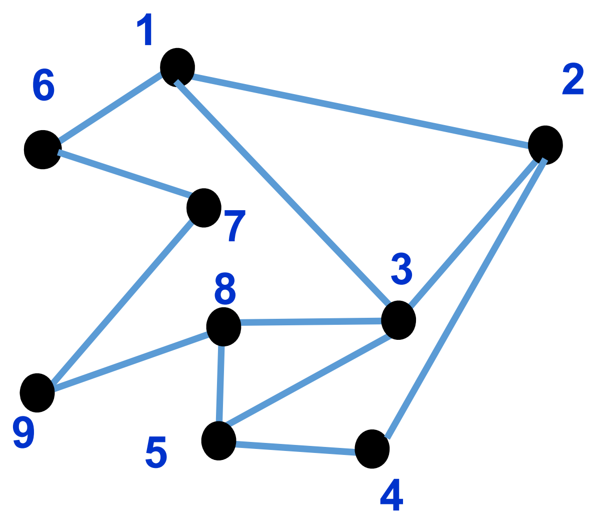
- **Bipartite Matching**
  - Given bipartite graph $\boldsymbol{G = (V, E)}$ create new graph $\boldsymbol{G' = (V', E')}$ (sometimes called the line-graph of $\boldsymbol{G}$) where
    - $\boldsymbol{V' = E}$
    - Two elements of $\boldsymbol{V'}$ (which are edges in $\boldsymbol{G}$) are joined iff they touch
  - Independent set $\boldsymbol{I}$ in $\boldsymbol{V'}$ ⇒ no edges in $\boldsymbol{I}$ touch ⇒ $\boldsymbol{I}$ is matching in $\boldsymbol{G}$

Bipartite Matching



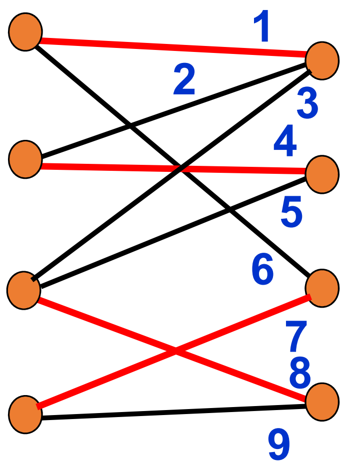
$$G = (V, E)$$

Independent Set



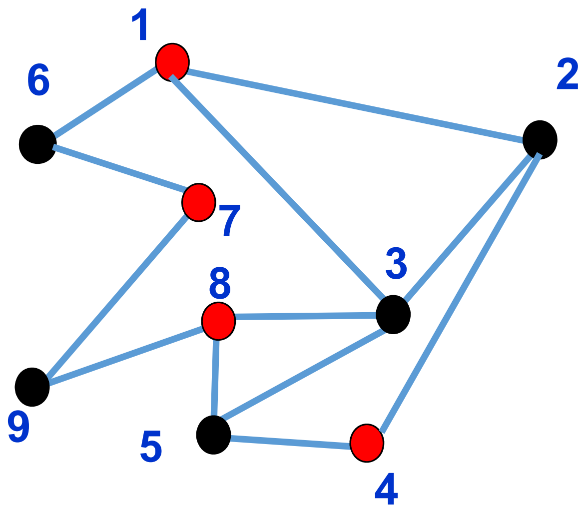
$$G' = (V', E')$$

Bipartite Matching



$G = (V, E)$

Independent Set



$G' = (V', E')$

Independent Set

No polynomial-time algorithm is known

- But to convince someone that there is a large independent set all you'd only need to tell them what the set is
  - they can easily convince themselves that the set is large enough and independent

- Convincing someone that there isn't such a set seems much harder

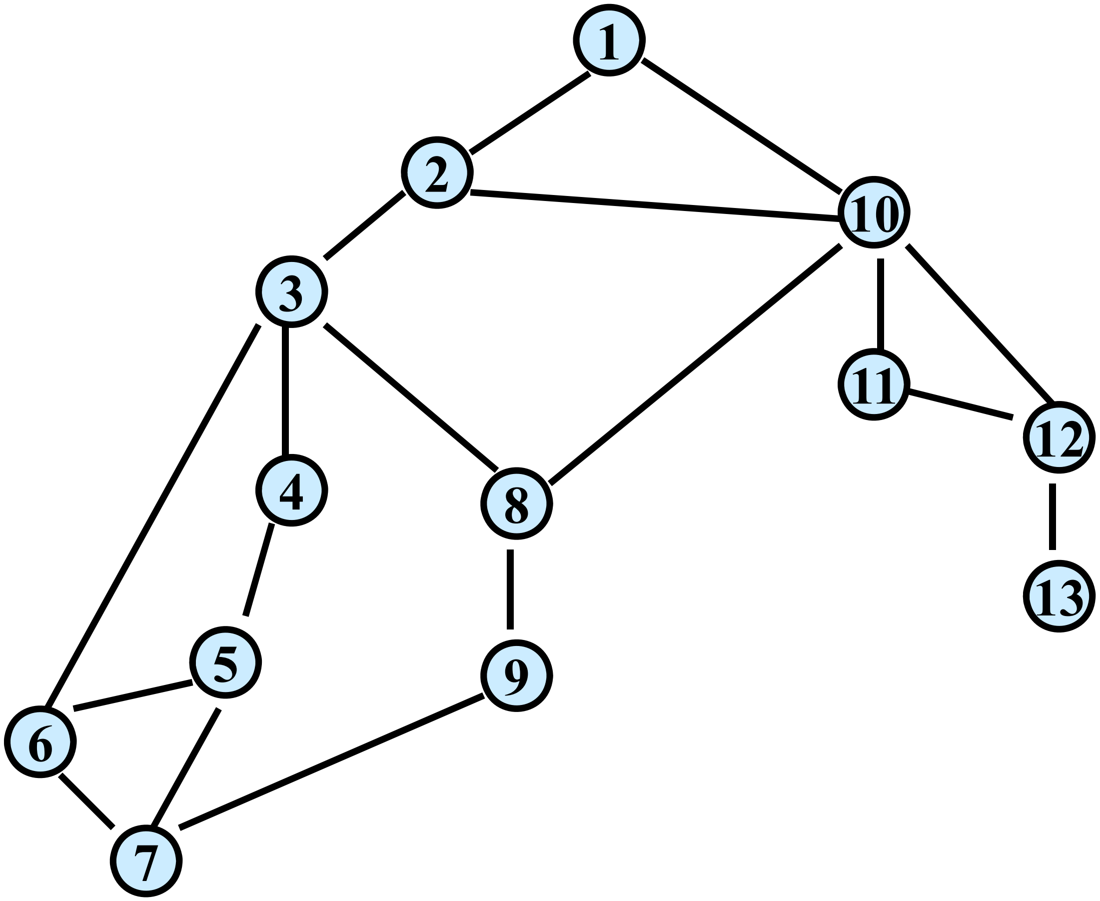
We will show that **Independent Set** is NP-complete

- Class of all the hardest problems that have the property above

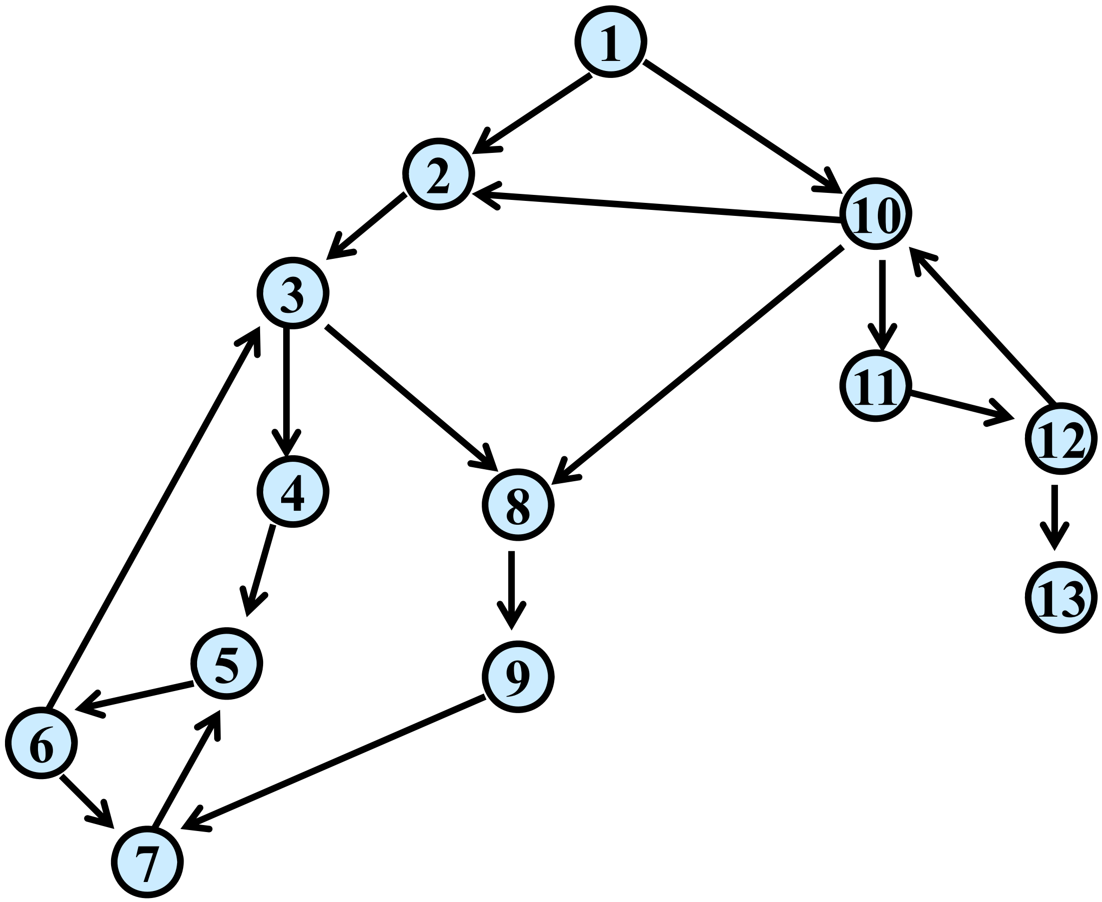
Introduction to Algorithms

- Graph Search/Traversal

Undirected Graph G = (V,E)



Directed Graph G = (V,E)



Graph Traversal

Learn the basic structure of a graph

Walk from a fixed starting vertex s to find all vertices reachable from s

Generic Graph Traversal Algorithm

Given: Graph graph $\boldsymbol{G = (V, E)}$ vertex $\boldsymbol{s \in V}$

Find: set $\boldsymbol{R}$ of vertices reachable from $\boldsymbol{s \in V}$

Reachable($\boldsymbol{s}$):

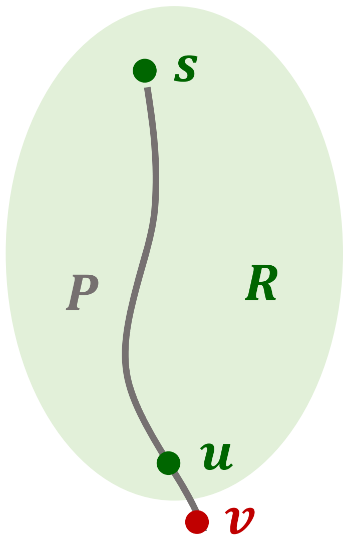
$\boldsymbol{R} \leftarrow \{\boldsymbol{s}\}$

while there is a $(\boldsymbol{u}, \boldsymbol{v}) \in \boldsymbol{E}$ where $\boldsymbol{u} \in \boldsymbol{R}$ and $\boldsymbol{v} \notin \boldsymbol{R}$

Add $\boldsymbol{v}$ to $\boldsymbol{R}$

return $\boldsymbol{R}$

Generic Traversal Always Works



Claim: At termination, R is the set of nodes reachable from s

Proof

$\subseteq$: For every node $v \in R$ there is a path from s to v

- Easy induction based on edges found.

$\supseteq$: Suppose there is a node $w \notin R$ reachable from s via a path P

- Take first node v on P such that $v \notin R$
- Predecessor u of v in P satisfies
  - $u \in R$
  - $(u, v) \in E$
- But this contradicts the fact that the algorithm exited the while loop. ■

Graph Traversal

Learn the basic structure of a graph

Walk from a fixed starting vertex s to find all vertices reachable from s

Three states of vertices

- **unvisited**
- **visited/discovered** (in R)
- **fully-explored** (in R and all neighbors have been visited)

Breadth-First Search

Completely explore the vertices in order of their distance from s

Naturally implemented using a queue

BFS(s)

Global initialization: mark all vertices "unvisited"

BFS(s)

mark s "visited"; $R \leftarrow \{s\}$; layer $L_0 \leftarrow \{s\}$; $i \leftarrow 0$

while L_i not empty

$L_{i+1} \leftarrow \emptyset$

for each $u \in L_i$

for each edge (u, v)

if (v is "unvisited")

mark v "visited"

Add v to set R and to layer L_{i+1}

mark u "fully-explored"

$i \leftarrow i + 1$

Properties of BFS

BFS(s) visits x iff there is a path in G from s to x.

Edges followed to undiscovered vertices define a

breadth first spanning tree of G

Layer i in this tree:

L_i = set of vertices u with shortest path in G from root s of length i.

Properties of BFS

Claim: For undirected graphs:
All edges join vertices on the same or adjacent layers of BFS tree

Proof: Suppose not...

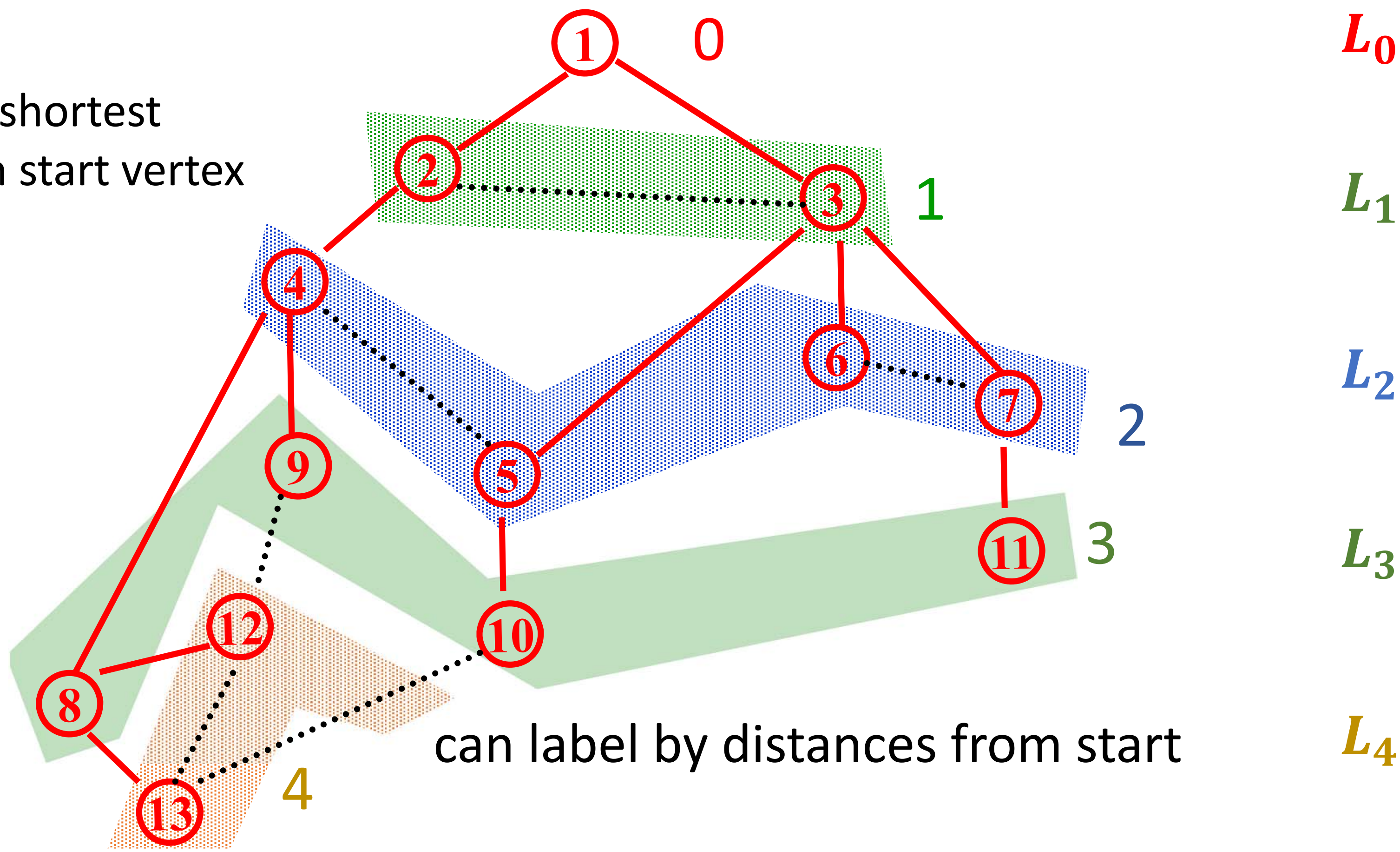
Then there would be vertices (x, y) s.t. $x \in L_i$ and $y \in L_j$ and $j > i + 1$.

Then, when vertices adjacent to x are considered in BFS,
y would be added to L_{i+1} and not to L_j.

Contradiction. ■

BFS Application: Shortest Paths

Tree gives shortest paths from start vertex



can label by distances from start