Parallel and Sequential Algorithms and Data Structures

LECTURE 8

Graphs and Graph Search

华中科技大学

HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Synopsis

- Graph and their Representation
- Graph Search
- Breadth-First Search
- Depth-First Search

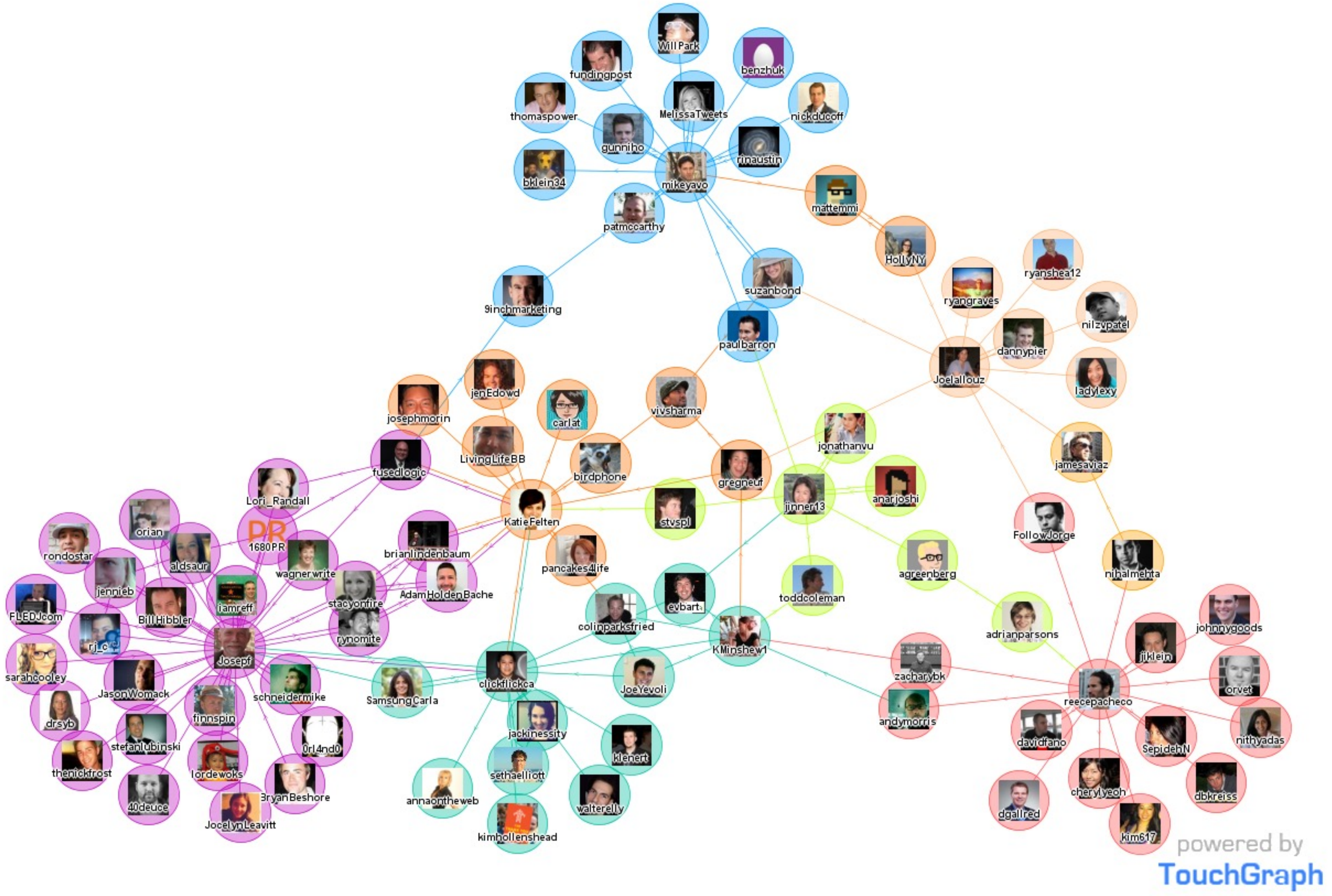
Graphs and Relations

- Graphs (sometimes referred to as networks) are **one of the most important abstractions in computer science**.
  - ➢ They are typically used to represent relationships between things from the most abstract to the most concrete,
    - ✓ mathematical objects
    - ✓ People
    - ✓ Events
    - ✓ molecules

计算的目标本质上来说是什么?

Social Networks - Questions

- Who is popular?
- What is the largest “clique”?
- Do I know somebody who knows X?
- What is the “diameter”?



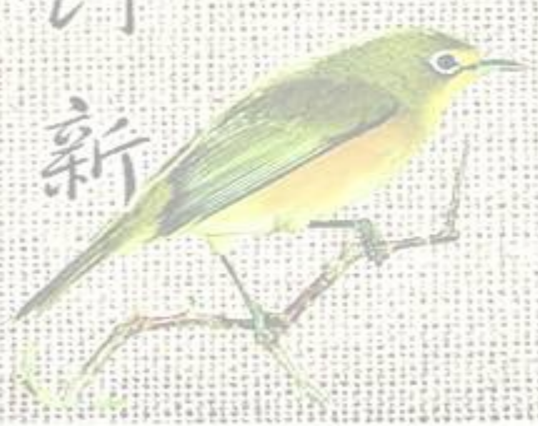
Transportation Networks -Questions

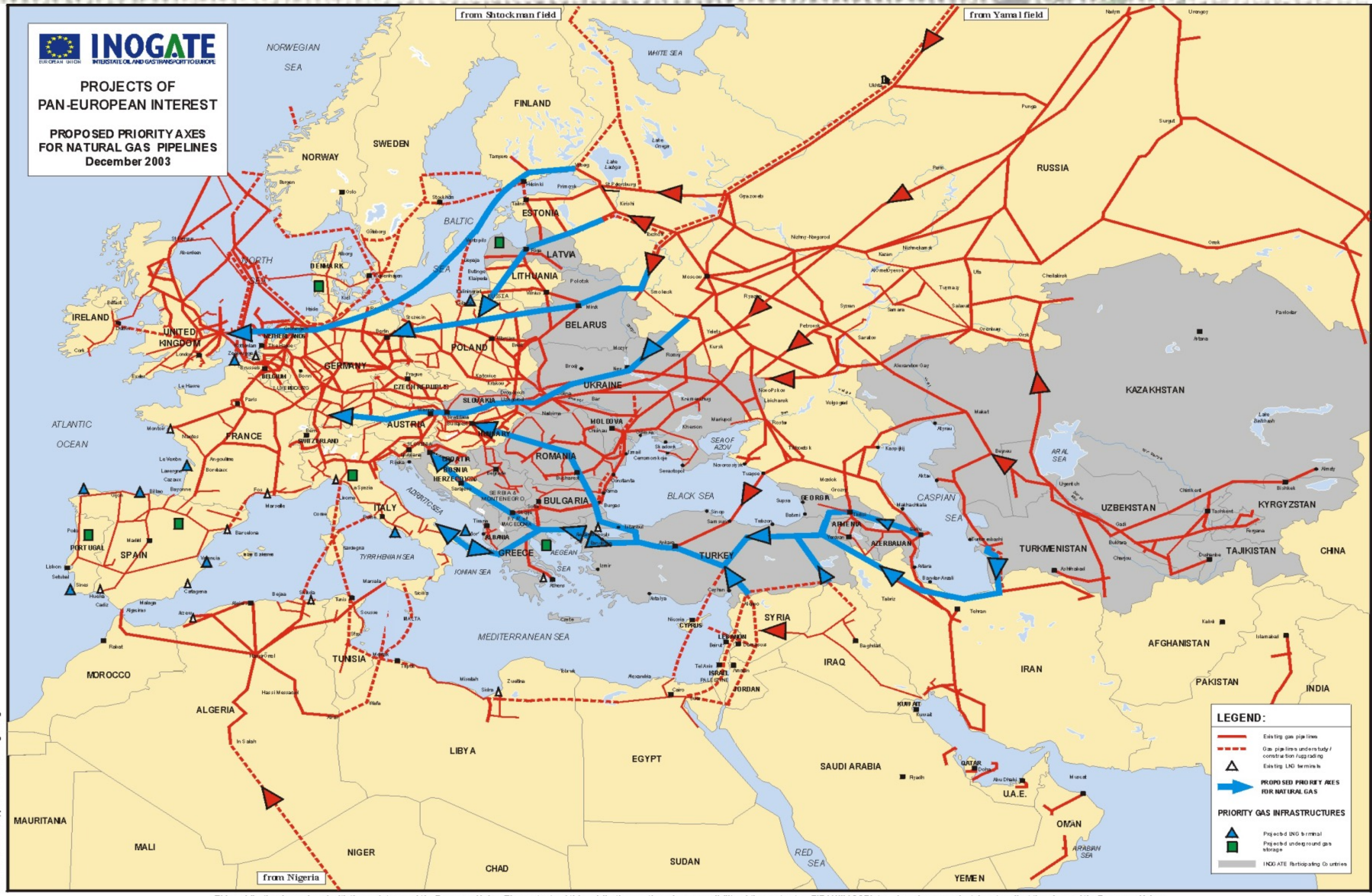
- What is the shortest route from NYC to Los Angeles?
  - ➢ without Toll Roads?
  - ➢ without any state roads?
- What is the expected driving time from Boston to Atlanta?
  - ➢ considering traffic congestion?

National Highway System
Canada
Mexico
The Bahamas
Alaska
Hawaii
Puerto Rico
U.S. Department of Transportation Federal Highway Administration FHWA: Effective October 1, 2012
Eisenhower Interstate System
Other NHS Routes
Non-Interstate STRAHNET Route
Major STRAHNET Connector
Intermodal Connector
Intermodal/STRAHNET Connector
Unbuilt NHS Routes
MAP-21 Principal Arterials
Census Urbanized Areas
Department of Defense
Water
Miles

Flow Networks - Questions

- Is it possible to send 1M cubic meters of gas to Paris daily?
- What is the maximum gas that can be pumped from Azerbaijan to Italy?





This publication is produced with the assistance of the European Union. The contents of this publication are the sole responsibility of the contractor, EIR / Hill / G&F int, and can in no way be taken to reflect the views of the European Union.

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Other Examples Of Graphs

- Course prerequisite relation graphs (directed-acyclic)
- Web-page linkage graph
- Protein-protein interaction graph
- Neural networks
- Semantic networks

Graph Representations

- Common operations a graph $G=(V,E)$ including
  - ➢ Map a function over the vertices $v \in V$
  - ➢ Map a function over the edges $(u,v) \in E$
  - ➢ Map a function over the (in or out) neighbors of a vertex $v \in V$
  - ➢ Return the degree of a vertex $v \in V$
  - ➢ Determine if the edge (u,v) is in E
  - ➢ Insert or delete an isolated vertex
  - ➢ Insert or delete an edge

Graphs Representations

- Different representations do <u>better on some operations and worse on others</u>
- <u>Cost can also depend on the *density* of the graph</u>, i.e. the relationship of the number of vertices and number of edges
- Assumption
  - ➢ consider a graph $G=(V,E)$ with n vertices and m edges
  - ➢ Throughout we assume that we <u>only delete isolated vertices</u>
    - ✓ If a vertex is incident on edges, then this means that we first have to delete the edges before deleting the vertex

Edge Sets

- The simplest representation of a graph is based on its definition as
  - a set of vertices V
  - a set of directed edges A ⊆ V×V

{
(Alice, Bob), (Alice, Arthur), (Bob, Alice), (Bob, Arthur),
(Arthur, Josefa), (Arthur, Bob), (Arthur, Alice), (Josefa, Arthur)
}.

Edge Sets

- Assuming we have a universe of possible vertices $\mathcal{V}$ (e.g., the integers, or character strings), we can represent directed graphs in that universe as

$$G = (\mathcal{V}\ set,\ (\mathcal{V} \times \mathcal{V})\ set).$$

-
  - The $(\mathcal{V}$ set) is the set of vertices and the $((\mathcal{V} \times \mathcal{V})$ set) is the set of directed edges
  - The sets could be represented with lists, arrays, trees, or hash tables.

Edge Sets

- the tree-based cost specification for sets
  - determine if an arc (u, v) is in the graph
    - ✓ W=log(m)
  - insert or delete an arc (u, v)
    - ✓ W=log(m)
  - identify the neighbors of a vertex v
    - ✓ W=O(m)
    - ✓ S=log(m)
    - ✓ Is it efficient?

how?

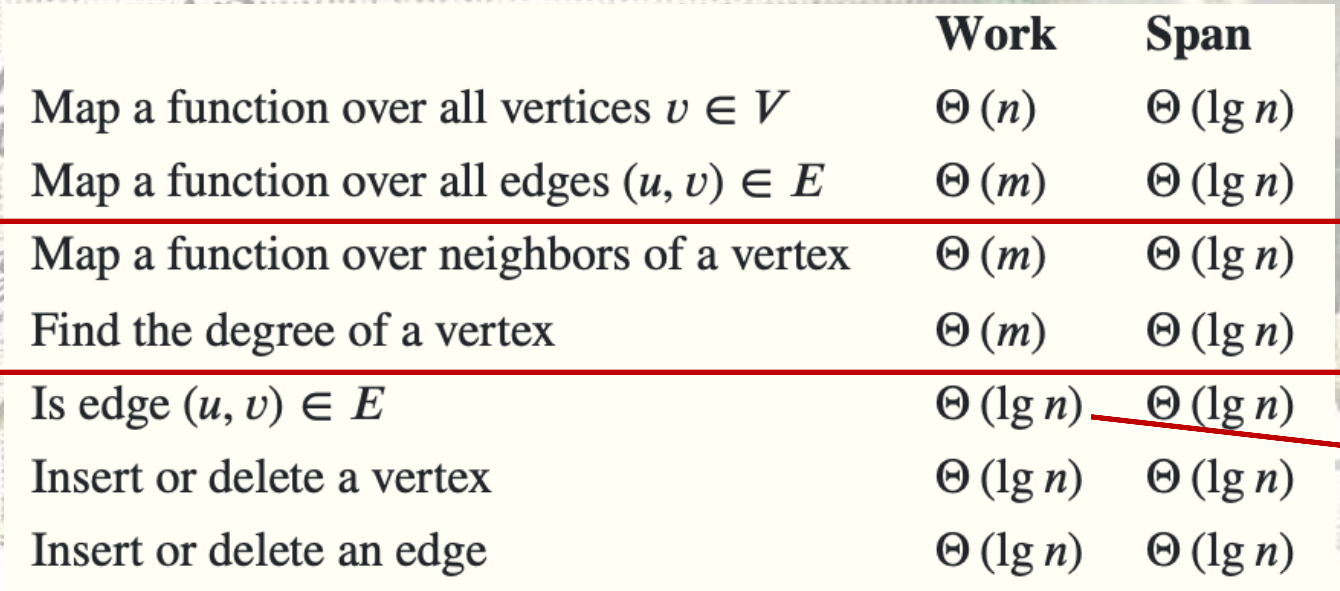
how?

華中科技大學

HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Edge Sets

- **Cost Specification (Edge Sets for Graphs)**
  - **For a graph represented as G=($\mathcal{V}$ set,($\mathcal{V}\times\mathcal{V}$) set) and assuming a tree-based cost model for sets**



	Work	**Span**
Map a function over all vertices $v \in V$	$\Theta(n)$	$\Theta(\lg n)$
Map a function over all edges $(u, v) \in E$	$\Theta(m)$	$\Theta(\lg n)$
Map a function over neighbors of a vertex	$\Theta(m)$	$\Theta(\lg n)$
Find the degree of a vertex	$\Theta(m)$	$\Theta(\lg n)$
Is edge $(u, v) \in E$	$\Theta(\lg n)$	$\Theta(\lg n)$
Insert or delete a vertex	$\Theta(\lg n)$	$\Theta(\lg n)$
Insert or delete an edge	$\Theta(\lg n)$	$\Theta(\lg n)$

how?

efficient?

Lg m?

Adjacency Tables

- Table items are *(key, value)* pairs
- Keys are vertex/node labels
- Values are either sets or tables
  - Sets: All neighbors node labels or out-neighbor node labels
  - Tables: All pairs of neighbors node labels and associated edge values

$$
\left\{
\begin{array}{l}
\text{Alice} \mapsto \{\text{Arthur}, \text{Bob}\}, \\
\text{Bob} \mapsto \{\text{Alice}, \text{Arthur}\}, \\
\text{Arthur} \mapsto \{\text{Alice}, \text{Josefa}\}, \\
\text{Josefa} \mapsto \{\text{Arthur}\}
\end{array}
\right\}
$$

Set or table?

Adjacency Tables

- **Definition (Adjacency Table Representation)**
  - The ***adjacency-table*** representation of a graph consists of a table mapping every vertex to the set of its out-neighbors and can be defined as

$$G = (\mathcal{V} \times (\mathcal{V}\ set))\ table.$$

  - Accessing neighbors needs log(*n*) work and span
  - finding, inserting or deleting an edge, w=log(*n*)
  - once the neighbor set has been pulled out, we can apply a constant work function over the neighbors in $O(d_G(v))$ work and $O(\log d_G(v))$ span

Adjacency Tables

- **Cost Specification(Adjacency Tables)**
  - For a graph represented as $G=(\mathcal{V} \times (\mathcal{V}\ \text{set}))$ table and assuming a tree-based cost model for sets and tables, we have that:

Operation	Work	Span
Map a function over all vertices $v \in V$	$\Theta(n)$	$\Theta(\lg n)$
Map a function over all edges $(u, v) \in E$	$\Theta(m)$	$\Theta(\lg n)$
Map a function over neighbors of a vertex	$\Theta(\lg n + d_g(v))$	$\Theta(\lg n)$
Find the degree of a vertex	$\Theta(\lg n)$	$\Theta(\lg n)$
Is edge $(u, v) \in E$	$\Theta(\lg n)$	$\Theta(\lg n)$
Insert or delete a vertex	$\Theta(\lg n)$	$\Theta(\lg n)$
Insert or delete an edge	$\Theta(\lg n)$	$\Theta(\lg n)$

how?

Adjacency Sequences

- Table items are *(key, value)* pairs
- Keys are vertex/node labels
- Values are adjacency sequences, this means use sequences to represent both tables and sets
  - ➢ Recall that a sequence is a table with a domain taken from {*0, ……, n-1*}
  - ➢ Sequences allow for fast random access, requiring only O(1) work to *access the ith element* rather than O(log n)

$$
\begin{array}{l}
\langle \\
\quad \langle 1, 2 \rangle, \\
\quad \langle 0, 2, 3 \rangle, \\
\quad \langle 0, 1 \rangle, \\
\quad \langle 1 \rangle \\
\rangle.
\end{array}
$$

$\boxed{\text{how?}}$

Adjacency Sequences

- **Definition (Adjacency Sequences for Enumerable Graphs)**
  - For enumerable graphs $G=(V,E)$, where $V=\{0\ldots(n-1)\}$, we can use sequences to improve the efficiency of the adjacency table representation. The type of a graph in this representation is thus

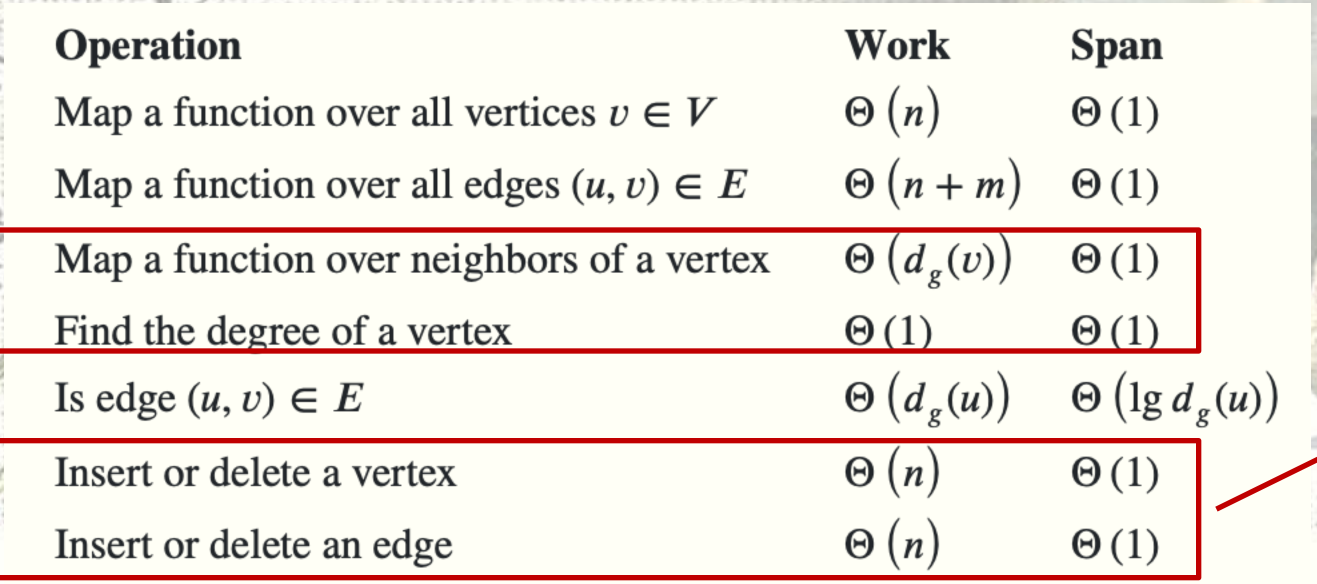
$$\mathbf{G} = (int\ seq)\ seq.$$

How about the big graph?

  - Here the length of the outer sequence in n, and the length of each inner sequences equals the degree of the corresponding vertex
  - This representation allows for fast random access, requiring only Θ(1) work to access the ith vertex.

Adjacency Sequences

- **Cost Specification (Adjacency Sequence)**
  - **Consider a graph with vertices V={0,…,n−1}, represented as G=(int seq) seq. Assuming an (persistent) array-sequence cost model**



Operation	Work	Span
Map a function over all vertices $v \in V$	$\Theta(n)$	$\Theta(1)$
Map a function over all edges $(u, v) \in E$	$\Theta(n+m)$	$\Theta(1)$
Map a function over neighbors of a vertex	$\Theta(d_g(v))$	$\Theta(1)$
Find the degree of a vertex	$\Theta(1)$	$\Theta(1)$
Is edge $(u, v) \in E$	$\Theta(d_g(u))$	$\Theta(\lg d_g(u))$
Insert or delete a vertex	$\Theta(n)$	$\Theta(1)$
Insert or delete an edge	$\Theta(n)$	$\Theta(1)$

How?

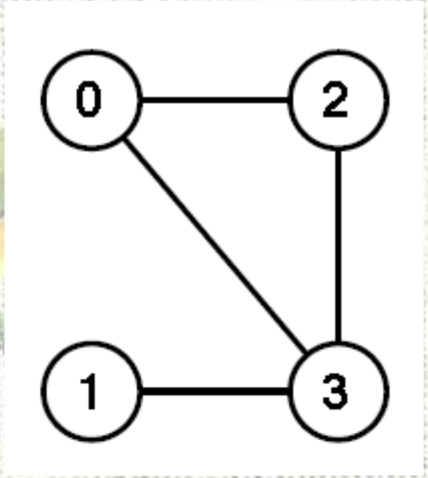
Adjacency Sequences

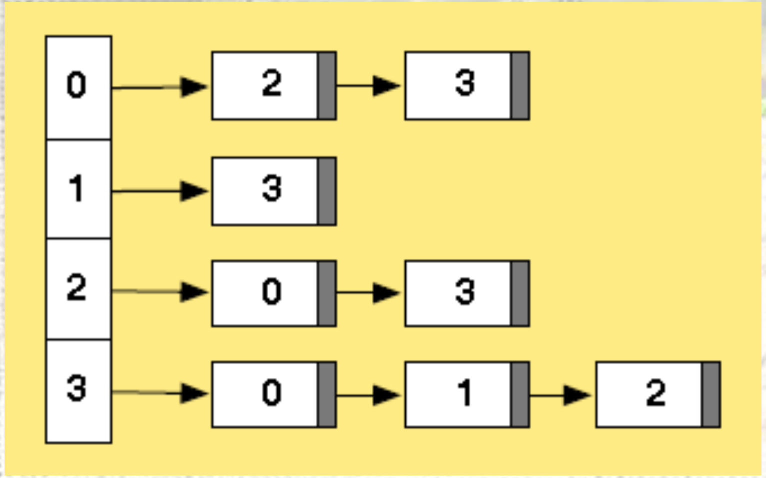
- **Adjacency List Representation**
  - ➢ In the adjacency sequence representation, we can represent the inner sequences (the out-neighbor sequence of each vertex) by using arrays or lists
  - ➢ If we use lists, then the resulting representation is the same as the classic ***adjacency list*** representation of graphs
    - ✓ This is a traditional representation used in sequential algorithms
    - ✓ It is not well suited for parallel algorithms since traversing the adjacency list of a vertex will take span proportional to its degree

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Adjacency List Representation

- Graph is represented by an array A of length n where each entry A[i] contains a pointer to a linked list of all the out-neighbors of vertex i
  - In an undirected graph edge {u, v} will appear in the adjacency list for both u and v (not always necessary!)





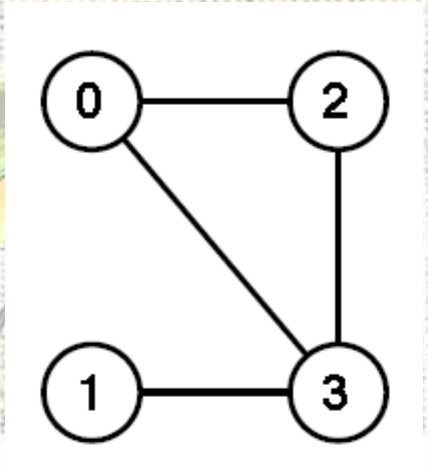
Adjacency Sequences

- **Mixed Adjacency Sequences and Tables**
  - ➢ It is possible to mix adjacency tables and adjacency sequences by having either the inner sets or the outer table be a sequence, but not both
  - ➢ Using sequences for the inner sets has the advantage that it defines an ordering over the edges of a vertex
  - ➢ This can be helpful in some algorithms

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Adjacency Matrices

- Assume vertices are numbered 1, 2, . . . , n (or 0, 1, . . . , n – 1)
- Graph is represented by an n $\times$ n matrix of binary values in which location (i, j) is 1 if (i, j) ∈ E and 0 otherwise
  - ➢ For undirected graphs, matrix is symmetric and has 0's along the diagonal



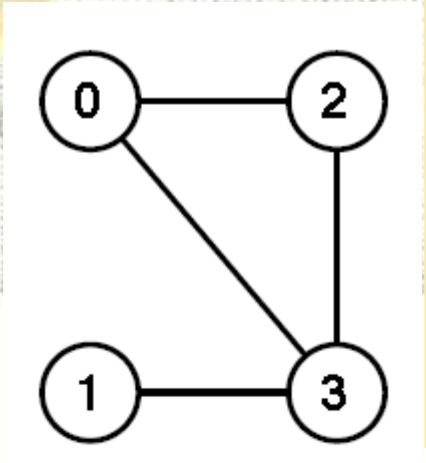
$$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Adjacency Matrices

- A matrix can be represented as a sequence of sequences of booleans (or zeros and ones), for which the type of the representation is:

$$G = (bool\ seq)\ seq$$

- For a graph with n vertices the outer sequence and all the inner sequences have equal length n



$$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Adjacency Matrices

- **Cost Specification(Adjacency Matrix)**
  - ➢ **For a graph represented as an adjacency matrix with** V={0,…,n−1} **and** G=(bool seq) seq**, and assuming the an array-sequence cost model**

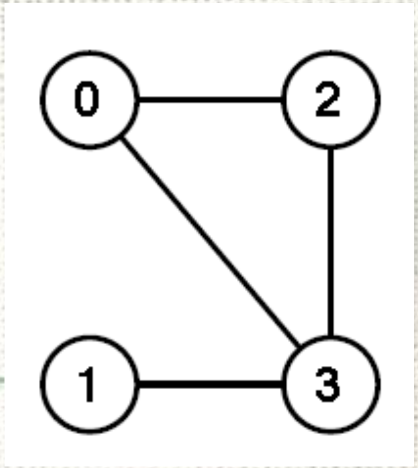
Operation	Work	Span
Map a function over all vertices $v \in V$	$\Theta(n)$	$\Theta(1)$
Map a function over all edges $(u, v) \in E$	$\Theta(n^2)$	$\Theta(1)$
Map a function over neighbors of a vertex	$\Theta(n)$	$\Theta(1)$
Find the degree of a vertex	$\Theta(n)$	$\Theta(\lg n)$
Is edge $(u, v) \in E$	$\Theta(1)$	$\Theta(1)$
Insert or delete a vertex	$\Theta(n^2)$	$\Theta(1)$
Insert or delete an edge	$\Theta(n)$	$\Theta(1)$

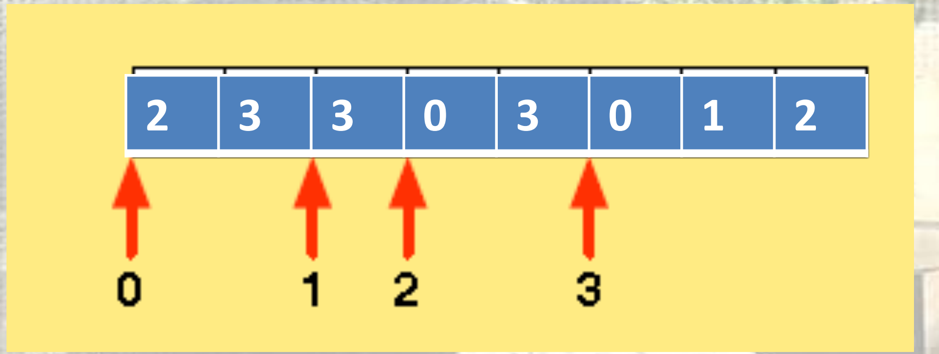
How?

Other Representations

- **Adjacency Array**

question?





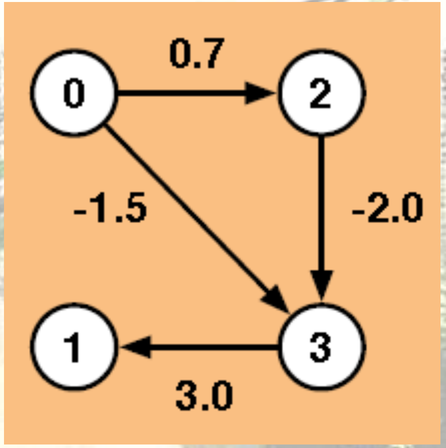
- **Edge List**

((0,2), (0, 3), (1, 3), (2,0), (2, 3), (3, 0), (3, 1), (3, 2))

Representing Weighted Graphs

- **Weighted and Edge-Labeled Graphs**
  - **An edge-labeled graph or a weighted graph is a triple G = (E, V, w) where w: E$\rightarrow$ L is a function mapping edges or directed edges to their labels (weights) , and L is the set of possible labels (weights)**
- **three different representations of graphs suitable for parallel algorithms**
  - **Edge sets**
  - **adjacency tables**
  - **adjacency sequences**
  - **How to extend to support edge values?**

Representing Weighted Graphs



- **edge table**

$$W = \{(0,2) \mapsto 0.7,\ (0,3) \mapsto -1.5,\ (2,3) \mapsto -2.0,\ (3,1) \mapsto 3.0\}$$

- **adjacency table**

$$G = \{1 \mapsto \{2 \mapsto 0.7, 3 \mapsto -1.5\}, 3 \mapsto \{3 \mapsto -2.0\}, 4 \mapsto \{1 \mapsto 3.0\}\}$$

- **adjacency sequence**

$$G = \langle \langle (2, 0.7),\ (3, -1.5) \rangle, \langle \rangle, \langle (3, -2.0) \rangle, \langle (1, 3.0) \rangle \rangle$$

Synopsis

- Graph and their Representation
- Graph Search
- Breadth-First Search
- Depth-First Search

Generic Graph Search

- **Definition(Source)**
  - A graph search usually starts at a specific ***source*** vertex $s \in V$ or a set of vertices ***sources*** . Graph search then searches out from the source(s) and iteratively ***visits*** the unvisited neighbors of vertices that have already been visited.
- **Definition(Visited Vertices)**
  - Graph search algorithms keep track of the ***visited vertices*** , which have already been visited, and avoid re-visiting them. We typically denote the set of visited vertices with the variable X.

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Generic Graph Search

- **Definition(Frontier and Discovered Vertices)**
  - For a graph $G=(V,E)$ and a visited set $X \subset V$, the ***frontier set*** or simply the ***frontier*** is the set of un-visited out-neighbors of X, i.e., the set $N^{+}_{G}(X) \setminus X$
    - ✓ We often denote the frontier set with the variable F
    - ✓ We refer to each vertex in the frontier as a ***discovered vertex***
- **Reminder**
  - Recall that $N^{+}_{G}(v)$ are the out-neighbors of the vertex v in the graph G, and $N^{+}_{G}(U) = \bigcup_{v \in U} N^{+}_{G}(v)$ (i.e., the union of all out-neighbors of U).

Generic Graph Search

- Algorithm(Graph Search (Single Source))
  - The generic graph-search algorithm that starts at a single source vertex s is given

```
graphSearch$(G, s)$ =
  let
    explore $X$ $F$ =
      if $(|F| = 0)$ then $X$
      else
        let
          choose $U \subseteq F$ such that $|U| \geq 1$
          visit $U$
          $X = X \cup U$
          $F = N_G^+(X) \setminus X$
        in explore $X$ $F$ end
  in
    explore $\{\}$ $\{s\}$
  end
```

Generic Graph Search

- Graph Search is Generic
  - ➢ Since the function `graphSearch` is not specific about the set of vertices to be visited next, it can be used to describe many different graph-search algorithms
    - ✓ Selecting all of the vertices in the frontier leads to breadth-first search (BFS).
    - ✓ Selecting the single most recent vertex added to the frontier leads to depth-first search (DFS)
    - ✓ Selecting the highest-priority vertex (or vertices) in the frontier, by some definition of priority, leads to priority-first search (PFS)

Reachability

- **Definition(Reachability)**
  - We say that a vertex v is ***reachable*** from u in the graph G (directed or undirected) if there is a path from u to v in G
- **Problem(The Graph Reachability Problem)**
  - For a graph $G=(V,E)$ and a vertex $v \in V$, return all vertices $U \subseteq V$ that are reachable from v
- **Theorem(Graph Search Solves Reachability)**
  - The function `graphSearch` $(G=(V,E))$ s returns exactly the set of vertices that are reachable in G from $s \in V$, and does so in at most $|V|$ rounds, and for any selection of U on each round

$\boxed{\textbf{Can you prove?}}$

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Graph-Search Tree

- Definition(Graph-Search Tree)
  - Let $G=(V,E)$ be a graph. A *graph-search tree* for an execution of the graph search algorithm of G is a rooted tree over the visited vertices $X \subseteq V$ and the edges $E' \subseteq E$ such that every vertex $v \in X \setminus \{s\}$ has a parent u that is in X when v is visited and the $(u,v) \in E$
  - **u is v's "blame" vertex**
  - The source s is the root of the tree (and has no parent)

Priority-First Search (PFS)

- Many graph-search algorithms can be viewed as visiting vertices in some priority order
  - ➢ The idea is to assign a priority to all vertices in the frontier, allowing the priority of a vertex to change whenever an in-neighbor of that vertex is visited.
  - ➢ When picking the set of vertices U to visit, we have several options:
    - ✓the highest priority vertex
    - ✓all highest priority vertices
    - ✓all vertices close to being the highest priority, perhaps the top k
- This specialization of generic graph search is called *Priority-First Search* or *PFS* for short.

華中科技大学

HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Synopsis

- Graph and their Representation
- Graph Search
- Breadth-First Search
- Depth-First Search

Breadth-first Search

- Applicable to a variety of problems
  - Connectedness
  - Reachability
  - Shortest path
  - Diameter
  - Bipartedness
- Applicable to both directed and undirected graphs
  - For digraphs, we only consider outgoing arcs

BFS and Distances

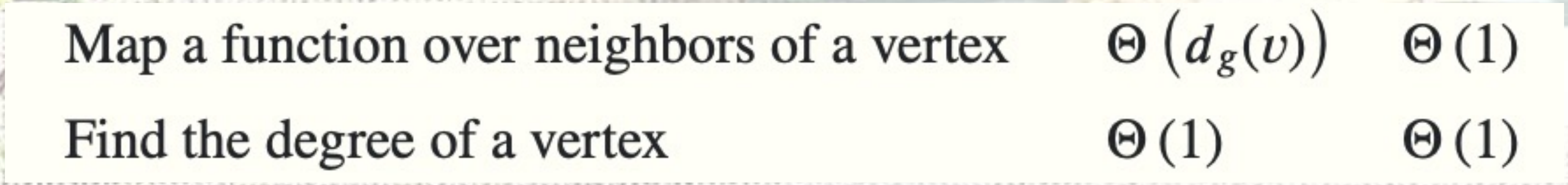
- **Definition(Distance of a Vertex)**
  - To understand how BFS operates, consider a graph $G=(V,E)$ and a source vertex $s \in V$
  - For a given source vertex s, define the ***distance*** of a vertex $v \in V$ from s as the shortest distance from s to v, that is the number of edges on the shortest path connecting s to v in G, denoted as $\delta_G(s,v)$
- **Definition(Breadth First Search)**
  - Breadth First Search (BFS) is a graph search that explores a given graph "outward" in all directions in increasing order of distances.
  - More precisely, for all distances $i<j$ in the graph, a vertex at distance i is visited before a vertex at distance j

华中科技大学

HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

BFS and Distances

- **Reminder(Representing Enumerable Graphs)**
  - Consider an enumerable graph $G=(V,E)$ where $V=\{0,1,\ldots,n-1\}$
  - As discussed earlier in adjacency sequence representations we can represent enumerable graphs as an adjacency sequence—that is, a sequence of sequences, where the ith inner sequence contains the out-neighbors of vertex i
  - This representation allows for finding the out-neighbors of a vertex in constant-work (and span)



Map a function over neighbors of a vertex	$\Theta\left(d_g(v)\right)$	$\Theta(1)$
Find the degree of a vertex	$\Theta(1)$	$\Theta(1)$

Sequential BFS

- **Algorithm (Sequential BFS: Reachability)**
  - To ensure that vertices are visited in distance order, <u>the frontier keeps the distance for each vertex</u>

```
BFSReach (G = (V, E)) s =
let explore X F i =
        if (|F| = 0) then (X, i)
        else let
            $(u, j) = \text{argmin}_{(v,k) \in F}(k)$
            $X = X \cup \{u\}$
            $F = F \setminus \{(u, j)\}$
            $F = F \cup \{(v, j+1) : v \in N_G^+(u) \mid v \notin X \land (v, \_) \notin F\}$
        in explore X F j end
in explore {} {(s, 0)} 0 end
```

How to realize?

Cost of Sequential BFS

- To start with, note that the algorithm uses the visited set X and the frontier F in a linear fashion
  - ➢ Representation of the Visited Set
    - ✓ We can use a boolean sequence of size $|V|$ to represent the visited set
      - – The value in the ith position indicates whether the vertex i is visited or not
    - ✓ We initialize the sequence with all `false`'s, indicating that none of the vertices are visited
    - ✓ When we visit a vertex, we update the corresponding element to `true`
      - – Because the visited set is used linearly, we can represent it with an ephemeral array, where update and sub (lookup) operations require constant work and span

Cost of Sequential BFS

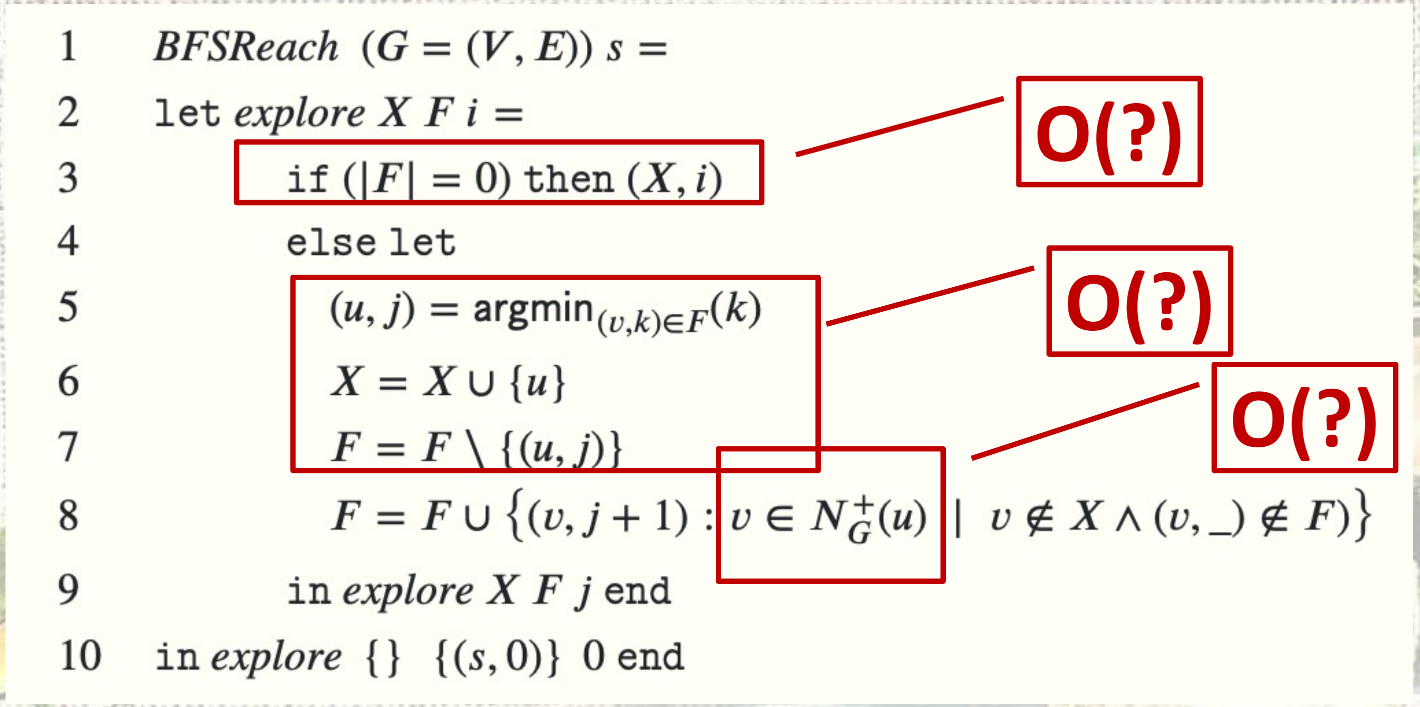
- To start with, note that the algorithm uses the visited set X and the frontier F in a linear fashion
  - ➢ Representation of the Frontier
    - ✓The frontier needs support several operations, including
      - – checking that a vertex is not in the frontier
      - – removing the vertex with the smallest distance
      - – adding the neighbors of a vertex into the frontier

How to represent F？

Cost of Sequential BFS

- **Representation of the Frontier**
  - **use a standard priority queue data structure:**
    - ✓ support insert and remove operations in logarithmic work and span
    - ✓ Because BFS visits the vertices only in increasing order of their distances, we only insert into the frontier vertices whose distances are no smaller than those in the frontier. In other words, the priorities increase monotonically as the algorithm proceeds
  - **use simpler priority queue data structure, i.e., just a simple ephemeral queue**
    - ✓ requires constant work and span to push to the tail of the queue and to pop from the head of the queue
    - ✓ In each round, we simply pop the vertex u at the head of the queue and visit u by marking the visited sequence. After the visit, we take all of u' s out-neighbors and check for each if they are visited. We push each unvisited out-neighbor into the tail of the queue. This implementation maintains the invariant that if a vertex has distance smaller than another, then it is closer to the head of the queue

Cost of Sequential BFS



```
BFSReach $(G = (V, E))\ s =$
let explore $X\ F\ i$ =
    if $(|F| = 0)$ then $(X, i)$
    else let
        $(u, j) = \text{argmin}_{(v,k) \in F}(k)$
        $X = X \cup \{u\}$
        $F = F \setminus \{(u, j)\}$
        $F = F \cup \{(v, j + 1) : v \in N_G^+(u) \mid v \notin X \land (v, \_) \notin F)\}$
    in explore $X\ F\ j$ end
in explore $\{\}\ \{(s, 0)\}\ 0$ end
```

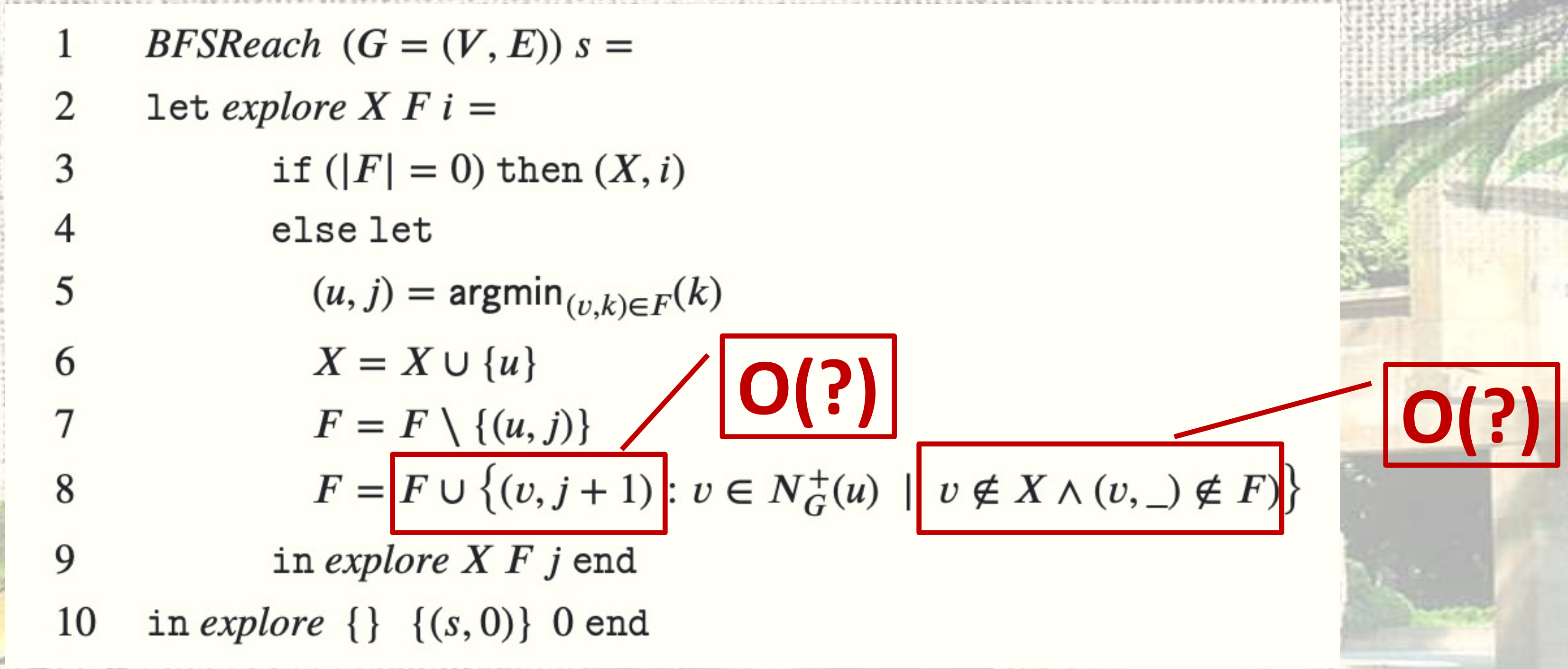
O(?)

O(?)

O(?)

- checking whether the frontier (queue) is empty or not, which is constant work
- If not empty, then we pop a vertex in constant work, and then mark it visited, also in constant work
- We then find all the out-neighbors of the vertex, which requires constant work by using the array-sequence based representation

Cost of Sequential BFS



```
$BFSReach\ (G = (V, E))\ s =$
let $explore\ X\ F\ i =$
        if $(|F| = 0)$ then $(X, i)$
        else let
            $(u, j) = \text{argmin}_{(v,k) \in F}(k)$
            $X = X \cup \{u\}$
            $F = F \setminus \{(u, j)\}$
            $F = \boxed{F \cup \{(v, j+1)}: v \in N_G^+(u) \mid \boxed{v \notin X \wedge (v, \_) \notin F}\}$
        in $explore\ X\ F\ j$ end
in $explore\ \{\}\ \{(s, 0)\}\ 0$ end
```

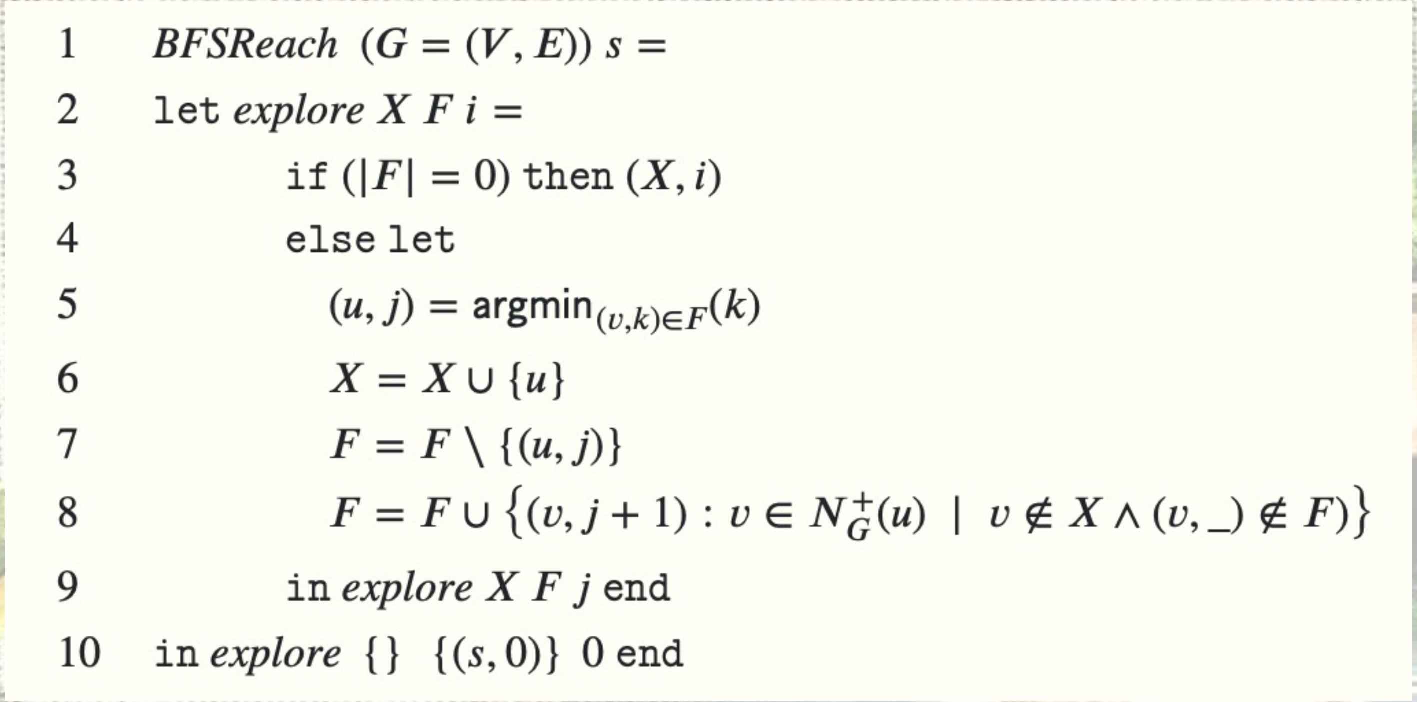
- We then check for each neighbor whether it is visited and if not, push it to the tail of the queue, which requires constant work per out-neighbor
- each vertex is pushed onto the queue at most once, the total number of push operations is bounded by the number of vertices and their work cost is $O(|V|)$

Cost of Sequential BFS

```
BFSReach $(G = (V, E))\ s =$
let explore $X\ F\ i$ =
        if $(|F| = 0)$ then $(X, i)$
        else let
          $(u, j) = \text{argmin}_{(v,k) \in F}(k)$
          $X = X \cup \{u\}$
          $F = F \setminus \{(u, j)\}$
          $F = F \cup \{(v, j+1) \mid \boxed{v \in N_G^+(u) \mid v \notin X \land (v, \_) \notin F)\}}$
        in explore $X\ F\ j$ end
in explore $\{\}\ \{(s, 0)\}\ 0$ end
```

- checking that each neighbor is visited, note that each such check corresponds exactly to one edge in the graph, and thus their number is bounded by m, and their total work cost is $O(|E|)$

Cost of Sequential BFS



```
$BFSReach\ (G = (V, E))\ s =$
let $explore\ X\ F\ i =$
    if $(|F| = 0)$ then $(X, i)$
    else let
      $(u, j) = \text{argmin}_{(v,k) \in F}(k)$
      $X = X \cup \{u\}$
      $F = F \setminus \{(u, j)\}$
      $F = F \cup \{(v, j+1) : v \in N_G^+(u) \mid v \notin X \wedge (v, \_) \notin F)\}$
    in $explore\ X\ F\ j$ end
in $explore\ \{\}\ \{(s, 0)\}\ 0$ end
```

- Because creating the initial sequence keeping track of visited vertices requires $\Theta(|V|)$ work, the total work of sequential BFS is $O(|V|+|E|)$

Parallel BFS

- **Algorithm(Parallel BFS: Reachability)**
  - The parallel BFS algorithm is an instance of Algorithm Graph Search (Single Source) where **all** frontier vertices are visited in each round

```
$BFSReach\ (G = (V, E), s) =$
let $explore\ X\ F\ i$ =
        if $(|F| = 0)$ then $(X, i - 1)$
        else let
            $X = X \cup F$
            $F = N_G^+(F) \setminus X$
        in $explore\ X\ F\ (i + 1)$ end
in $explore\ \{\}\ \{s\}\ 0$ end
```

Determine the neighbors of the frontier

Visit the frontier

Remove vertices that have been visited

Graph Search

- For all graph search methods vertices can be partitioned into three sets at any time during the search
  - vertices already visited (X ⊆ V)
  - the unvisited neighbors of the visited vertices, called the frontier (F)
  - the rest, unseen vertices
- The search essential goes as follows

```
while vertices remain
    -visit some unvisited neighbors of the visited set
```

- Web navigation analogy

BREADTH-FIRST SEARCH

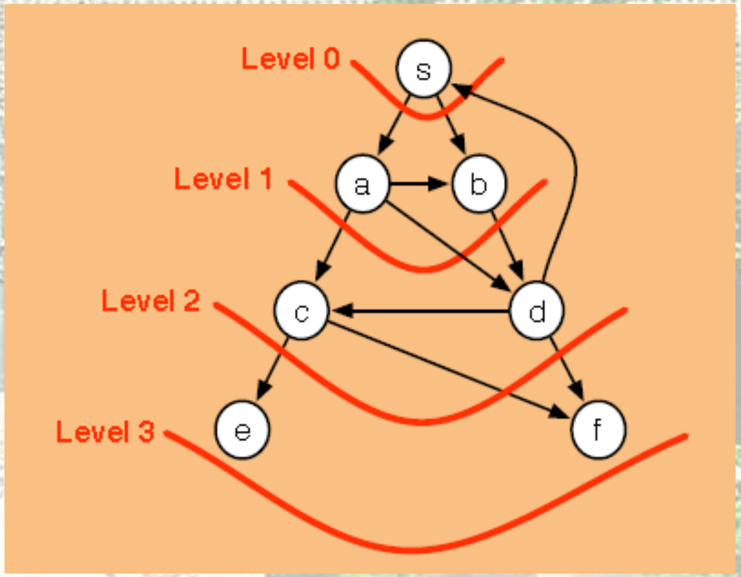
- Input : a graph G = (V, E) & a source vertex s ∈ V
- Output
  - BFS starts at the given source vertex s and explores the graph outward in all directions level by level

how?

- level
  - a vertex v∈V as the shortest distance (number of edges) from s to v

Breadth-first Search

- Starting from a source vertex s
  - Visit all vertices that are (out-)neighbors of s (at distance 1)
  - Visit all vertices at distance 2 from s
  - Visit all vertices at distance 3 from s
  - ……
- A vertex at distance i + 1 must have a (in-) neighbor at distance i



Parallel BFS and Distances

- **Lemma (Parallel BFS and Distances)**
  - In BFSReach $(G=(V,E))$ s, at the beginning of every invocation of explore (line 3), we have

$$\begin{array}{lclcl} X & = & X_i & = & \{v \in V \mid \delta_G(s,v) < i\}, \text{ and} \\ F & = & F_i & = & \{v \in V \mid \delta_G(s,v) = i\} \end{array}$$

```
1  BFSReach (G = (V, E), s) =
2  let explore X F i =
3          if (|F| = 0) then (X, i − 1)
4          else let
5             X = X ∪ F
6             F = N⁺_G(F) \ X
7          in explore X F (i + 1) end
8  in explore {} {s} 0 end
```

  - Prove
- **<u>By induction on levels i</u>**
- **For base case (i = 0) X_0 = {}, F_0 = {s}**
  - **Only s has distance 0 from s**
  - **No vertex has distance < 0 from s**
- **So base case is true!**

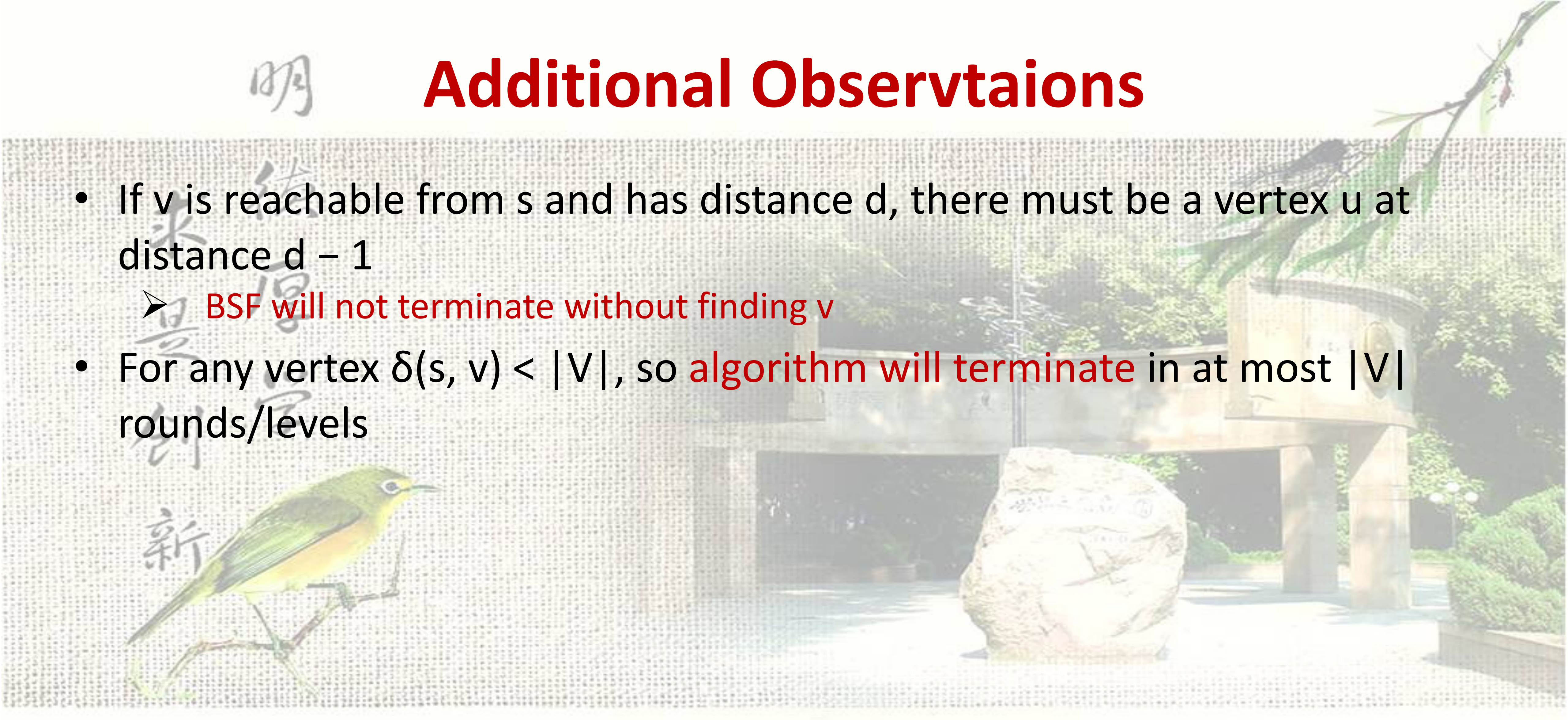
華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Proving BFS Correct

$$X = X_i = \{v \in V \mid \delta_G(s,v) < i\}, \text{ and}$$
$$F = F_i = \{v \in V \mid \delta_G(s,v) = i\}$$

```
1  BFSReach $(G = (V, E), s)$ =
2  let explore $X$ $F$ $i$ =
3        if $(|F| = 0)$ then $(X, i - 1)$
4        else let
5           $X = X \cup F$
6           $F = N_G^+(F) \setminus X$
7        in explore $X$ $F$ $(i + 1)$ end
8  in explore $\{\}$ $\{s\}$ $0$ end
```

- Assume claims are true for i, show for i + 1
- X_{i+1} must have all vertices at distance $< i + 1$
  - ➢ X_{i+1} is the union of
    - ✓ X_i : all vertices at distance < i
    - ✓ F_i : all vertices at distance = i
- $F_{i+1} = N_G(F_i) \setminus X_{i+1}$
  - ➢ Vertices in Fi have distance exactly i
  - ➢ Vertices in $N_G(F_i)$ have distance no more than i + 1
  - ➢ Vertices in $N_G(F_i)$ are reachable from a vertex at distance i+1
  - ➢ When we remove X_{i+1} from $N_G(F_i)$ only unvisited vertices at distance exactly i + 1 remain



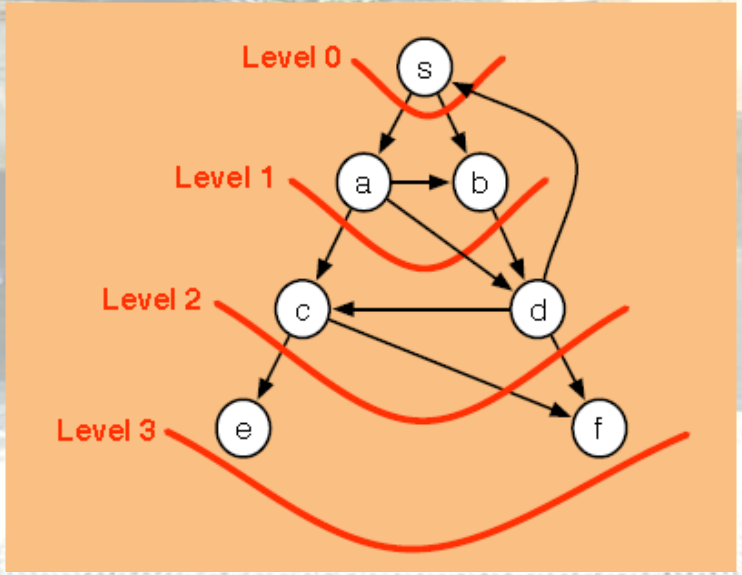
Additional Observtaions

- If v is reachable from s and has distance d, there must be a vertex u at distance d – 1
  - BSF will not terminate without finding v
- For any vertex δ(s, v) < |V|, so algorithm will terminate in at most |V| rounds/levels

Cost of Parallel BFS

- We analyze the cost the BFS variant for reachability from source.
  - ➢ represent the main data structures by using **tree-based sets and tables**.
    - ✓ vertices accept a comparison (total-order) operation

```
BFSReach $(G = (V, E), s) =$
let $explore\ X\ F\ i =$
        if $(|F| = 0)$ then $(X, i - 1)$
        else let
            $X = X \cup F$
            $F = N_G^+(F) \setminus X$
        in $explore\ X\ F\ (i + 1)$ end
in $explore\ \{\}\ \{s\}\ 0$ end
```



Cost of Parallel BFS

- BFS works in a sequence rounds (one per level)
- We can add up work and span in each round
  - ➢ But work at a level depends on number of outgoing edges from the frontier!
- Take a more global view
  - ➢ Each vertex appears exactly once in some frontier
  - ➢ All their (out-)edges are processed once
- $W_{BFS}(n,m) = W_v n + W_e m$
  - ➢ n = |V| and m = |E|

right?

HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Costs Per Vertex And Edge

- Nontrivial operations are
  - ➢ $X' = X \cup F$
  - ➢ $N = N_G(F)$
  - ➢ $F' = N \setminus X'$
- These all depend on size of F and number of outgoing edges from F
- Let $||F|| = \sum_{v \in F} (1 + d_G^+(v))$
  - ➢ denote the number of out-edges for a frontier plus the size of the frontier

Costs Per Vertex And Edge

	Work	Span
$X \cup F$	$O(\lvert F\rvert \log n)$	$O(\log n)$
$N \setminus X'$	$O(\lvert\lvert F\rvert\rvert \log n)$	$O(\log n)$

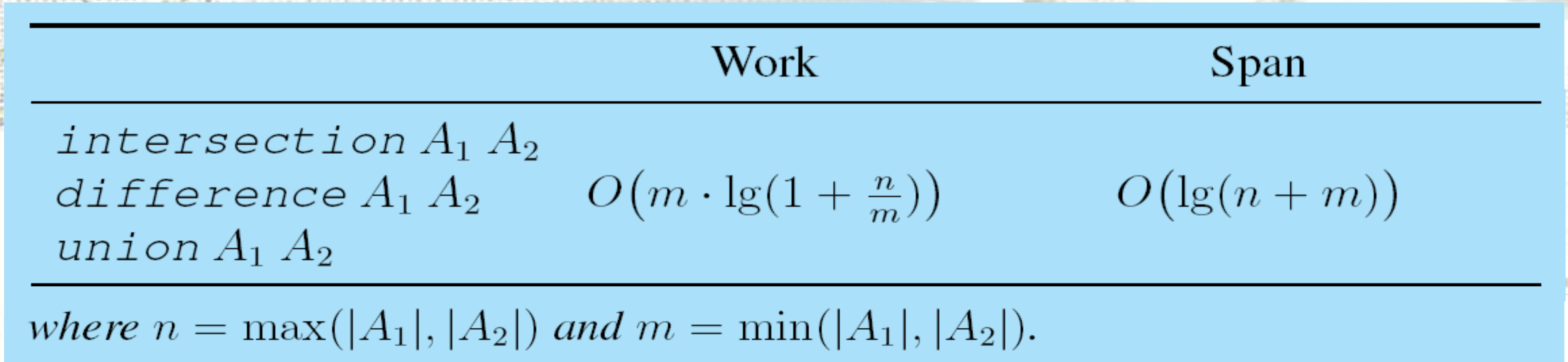
Why?

- **These come from the tree-based cost specification for the set ADT**

Gone with wind?

Work $= O(\boxed{W_c} \cdot |F| \log(1 + n/|F|)) = O(|F| \log n)$ Why?

Span $= O(\boxed{S_c \cdot} \log(n + |F|)) = O(\log n)$ Why?



	Work	Span
`intersection` A_1 A_2 `difference` A_1 A_2 `union` A_1 A_2	$O\left(m \cdot \lg(1 + \frac{n}{m})\right)$	$O\left(\lg(n + m)\right)$

where $n = \max(|A_1|, |A_2|)$ *and* $m = \min(|A_1|, |A_2|)$.

Costs Per Vertex And Edge

	Work	Span
$N_G(F)$	$O(\|\|F\|\| \log n)$	$O(\log^2 n)$

- Graph is represented as **a table mapping vertices to a set of their outneigbors**

let $N_G^+(F)$ = `Table.reduce Set.Union {} (Table.restrict` G F`)`

How about this?

- Let G_F = Table.restrict(G,F).
- The work to find G_F is $O(|F| \log n)$

Why?

Digression – Back To Reduce!

let $N^+_G(F)$ = `Table.reduce Set.Union {} (Table.restrict` G F`)`

R(reduce f || S) = {all function applications f (a, b) in the reduction tree**}**

$$W(reduce\ f\ \Vert\ S) = O(n + \sum_{f(a,b)\in R(f\ \Vert\ S)} W(f(a,b)))$$

$$S(reduce\ f\ \Vert\ S) = O(\log n \max_{f(a,b)\in R(f\ \Vert\ S)} S(f(a,b)))$$

Digression – Back To Reduce!

LEMMA

- For any combine function $f : \alpha \times \alpha \rightarrow \alpha$ and a monotone size measure $s : \alpha \rightarrow R_+$, if for any x, y,
  - $s(f(x, y)) \le s(x) + s(y)$ and
  - $W(f(x, y)) \le c_f (s(x) + s(y))$ for some universal constant c_f depending on the function f , then

$$W(reduce\ f\ \ \text{II}\ S) = O(\log |S| \sum_{x \in S} (1 + s(x)))$$

Back To Costs

- **In our case α is the set type, f is** Set.union **, s the size of a set**
  - **Size of the union ≤ sum of the sizes**
  - **Work of a union ≤ is at most proportional to size of the sets!**
- **So** Set.union **satisfies the conditions of the lemma**
- **G_F =** Table.restrict G **F**
  - **G_F is a set of neighbor sets**

$$W\,(\texttt{Table.reduce})\ \texttt{union}\ \{\}\ G_F) = O\left(\log|G_F| \sum_{v\mapsto N(v)\in G_F} (1+|N(v)|)\right)$$
$$= O\left(\log n \cdot \|F\|\right)$$

Back To Costs

- Focusing on a single round, we can see that the cost per vertex and edge visited in that round is O(log n)
- the cost per edge We and per vertex Wv over the algorithm is the same as the cost per round
- So at level i, W = O(||Fi || · log n) and each edge is processed once, ⇒
  - ➢ work per edge is O(log n)
- Wv = We = O(log n)
- $W_{BFS}(n,m) = O(n \log n + m\log n)$
  $= O(m\log n)$ (Why?)

Why?

Back To Costs

$S(\text{reduce union } \{\}\ G_F) = O(\log^2 n)$

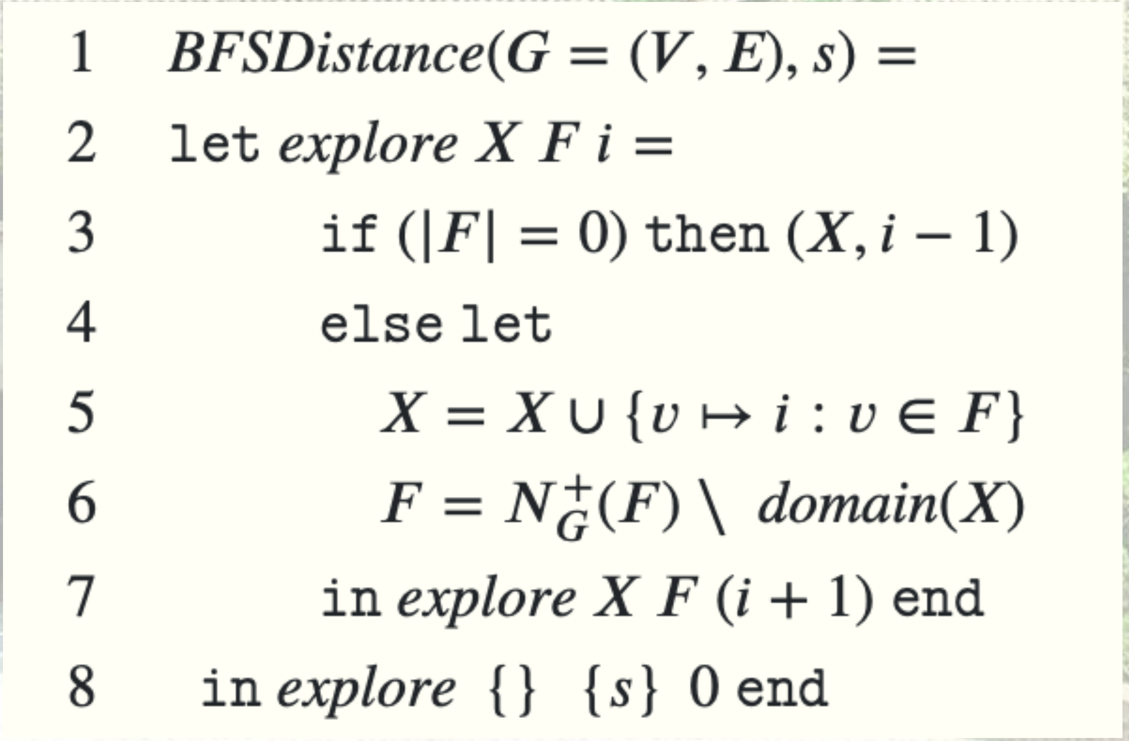
$\boxed{\text{Why?}}$

- Each union has span $O(\log n)$
- The reduction tree is bounded by $\log n$ depth

- Span depends on d
- $(S_{BFS}(n,m,d) = O(d \log^2 n))$
  - In worst $d \in O(n) \Rightarrow$ BFS is sequential.

Shortest Paths and Shortest-Path Trees

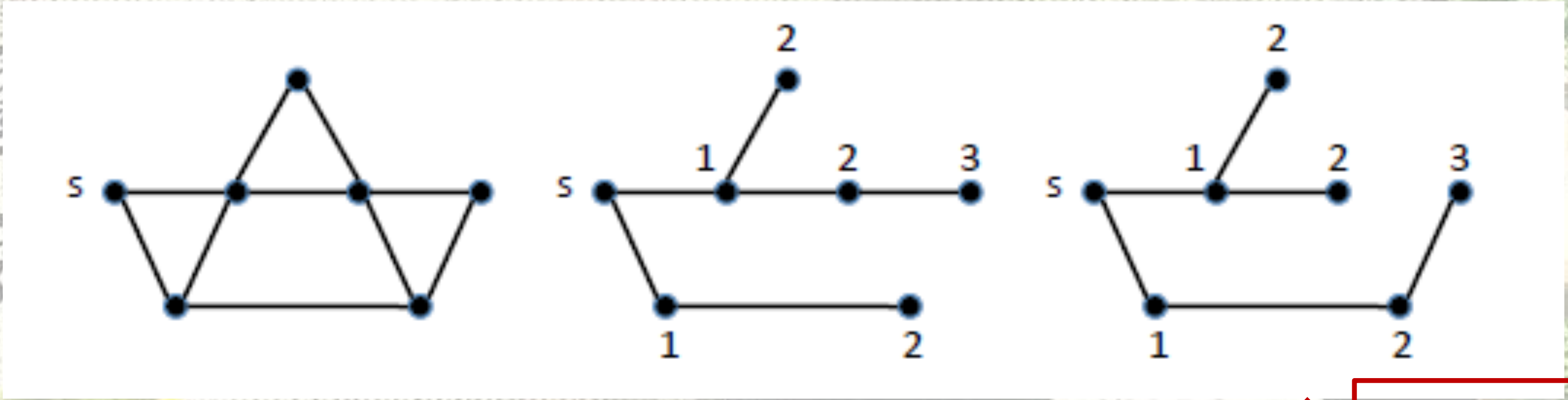
- Algorithm(Unweighted Shortest Paths)
  - The following variant of the Parallel BFS algorithm takes a graph and a source and returns a table mapping every reachable vertex v to $\delta_G(s,v)$
- Theorem(BFS Tree Gives Shortest Paths)
  - Given a graph-search tree for BFS, the path from any vertex v to the source s in the tree when reversed is a shortest path from s to v in G.



```
$BFSDistance(G = (V, E), s) =$
let $explore\ X\ F\ i =$
        if $(|F| = 0)$ then $(X, i - 1)$
        else let
            $X = X \cup \{v \mapsto i : v \in F\}$
            $F = N_G^+(F) \setminus domain(X)$
        in $explore\ X\ F\ (i + 1)$ end
  in $explore\ \{\}\ \{s\}\ 0$ end
```

Extensions To BFS

- Finding BFS trees

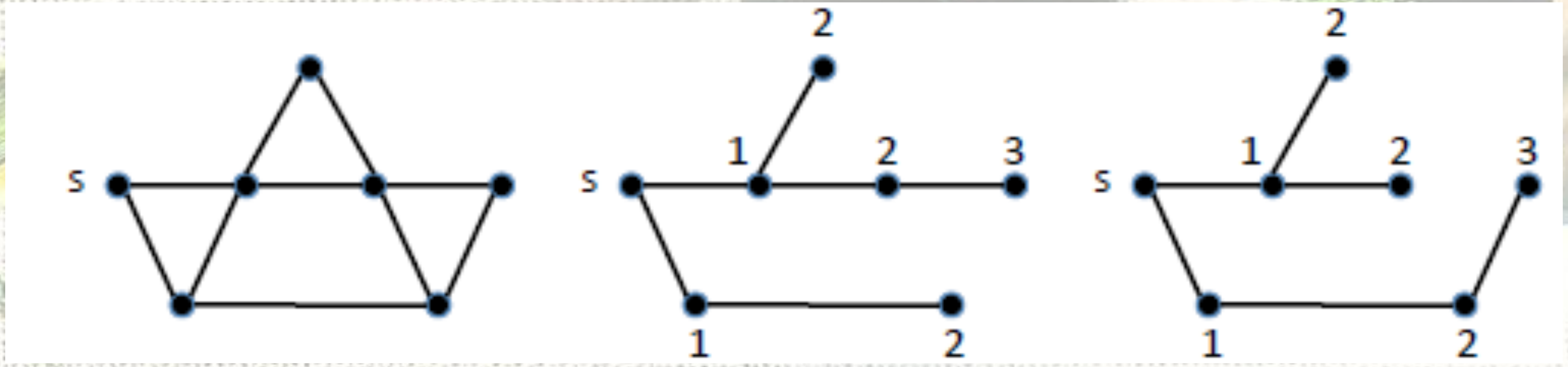


- There could be multiple BFS trees
- in un-weighted graphs, a BFS Tree is the same as a shortest path tree
  - ➢ shortest path tree contains the shortest path from the source to each vertex

How to prove?

Finding BFS Trees

- What do we need to keep for each vertex?
- Record a parent
    - ➢ If v is in a frontier, then there should be one or more visited vertices u such that (u, v) ∈ E
    - ➢ Any of those could be the parent of v



Identifying Parents

- Post-process the BFS distance table
  - ➢ Identify one (in-)neighbor vertex in $N^{-}(v)$ whose distance is one less
- Another way is to keep a table of vertices mapping to parents
  - ➢ For each $v \in F$, generate a table $\{u \rightarrow v : u \in N(v)\}$
  - ➢ Maps each neighbor of v back to v
- Merge these tables to X
  - ➢ Choose one if you have multiple parents

Shortest Paths and Shortest-Path Trees

- Algorithm(BFS Tree with Sequences)
  - We present a variant of BFS that computes the shortest-path tree from the source by using sequence-based data structures
  - The algorithm returns a shortest-paths tree as a sequence mapping each vertex (position) to its parent in the shortest-paths tree
  - The source points to itself and unvisited vertices contain None

```
$BFSTree\ (G, s) =$
let
  $explore(X, F) =$
    if $(|F| = 0)$ then $X$
    else let
      (* {Visit F} *)
      $f(v) = \langle (u, \mathrm{Some}\ v) : u \in G[v] \mid X[u] = \mathrm{None} \rangle$
      $N = Seq.flatten\ \langle f(v) : v \in F \rangle$
      $X = Seq.inject\ X\ N$
      $F = \langle u : (u, v) \in N \mid X[u] = v \rangle$
    in $explore(X, F)$ end
  $X = \langle \mathrm{None} : v \in \langle 0, \ldots, |G| - 1 \rangle \rangle$
  $X = Seq.update\ X\ (s, \mathrm{Some}\ s)$
in $explore(X, \{s\})$ end
```

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

BFS With ST Sequences

- BFS Costs revisited

$W_{BFS}(n,m) = O(m\log n)$

$S_{BFS}(n,m, d) = O(d \log^2 n)$

- Using single-threaded sequences reduces costs to

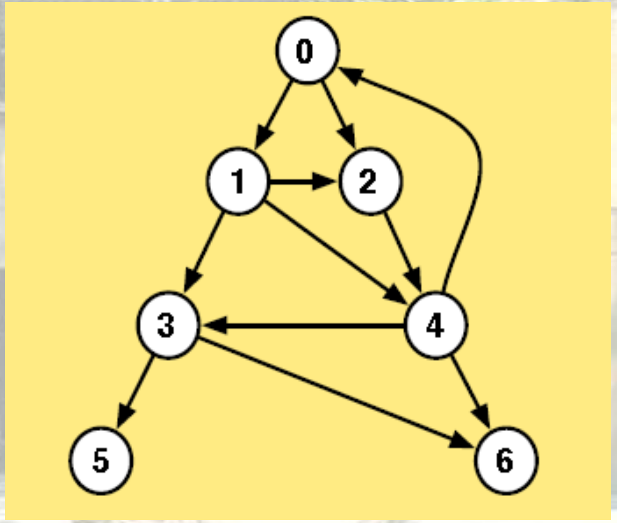
$W_{BFS}(n,m) = O(n+m)$

$S_{BFS}(n,m, d) = O(d \log n)$

BFS With ST Sequences

- Enumerated graph G=(V, E),
  - vertices are labeled with integers V = {0, 1, . . . , n − 1}
  - (int seq) seq

$\langle \langle 1,2 \rangle , \langle 2,3,4 \rangle , \langle 4 \rangle , \langle 5,6 \rangle , \langle 3,6 \rangle , \langle\ \rangle , \langle\ \rangle \rangle$



  - The representation supports constant-work lookup operations for finding the out-edges (or out-neighbors) of a vertex

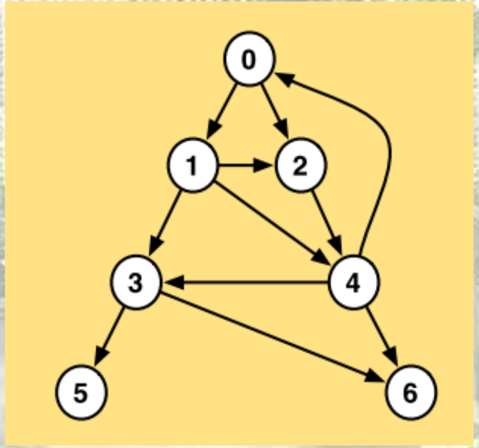
BFS With ST Sequences

- Since the graph does not change during BFS, this representation suffices for implementing BFS
- No change？
  - ➢ whether a vertex is visited or not by using the visited set X
  - ➢ use a single-threaded sequence of length |V| to mark which vertices have been visited
  - ➢ By using inject, we can mark vertices in constant work per update
- For each vertex, we can use either a Boolean flag to indicate its status, or the label of the parent vertex (if any)
  - ➢ The latter representation can help up construct a BFS tree

BFS Tree with Sequences

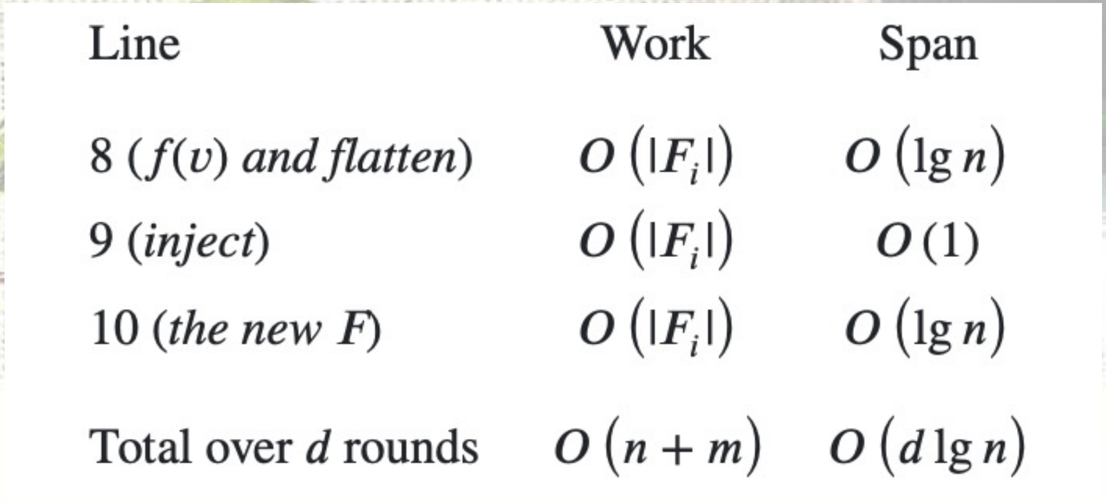
- Recall： Algorithm(BFS Tree with Sequences)

```
BFSTree (G, s) =
let
  explore(X, F) =
    if (|F| = 0) then X
    else let
      (* {Visit F} *)
      $f(v) = \langle (u, \text{Some } v) : u \in G[v] \mid X[u] = \text{None} \rangle$
      $N = Seq.flatten\ \langle f(v) : v \in F \rangle$
      $X = Seq.inject\ X\ N$
      $F = \langle u : (u, v) \in N \mid X[u] = v \rangle$
    in explore(X, F) end
  $X = \langle \text{None} : v \in \langle 0, \ldots, |G| - 1 \rangle \rangle$
  $X = Seq.update\ X\ (s, \text{Some } s)$
in explore(X, {s}) end
```



Cost

- **Cost Analysis**
  - **The cost of the algorithm is dominated by** flatten, inject, **and by the construction of the next frontier F in Lines 8, 9, and 10**
  - **The following table gives costs for each round, and then the total across rounds including also the $\Theta(n)$ initialization cost**



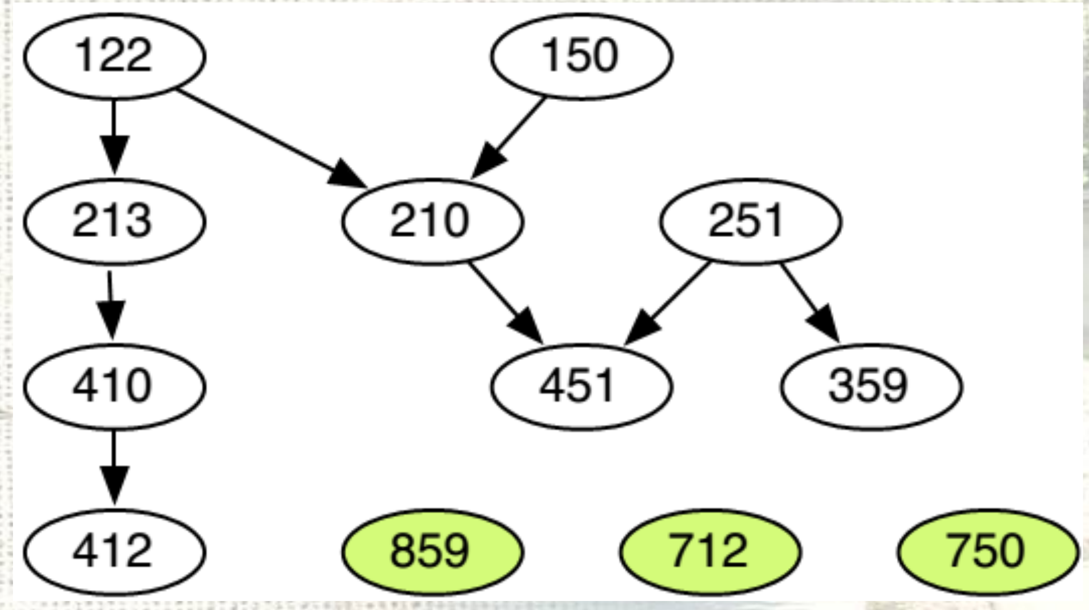
Line	Work	Span
8 (*$f(v)$ and flatten*)	$O(\lvert F_i \rvert)$	$O(\lg n)$
9 (*inject*)	$O(\lvert F_i \rvert)$	$O(1)$
10 (*the new F*)	$O(\lvert F_i \rvert)$	$O(\lg n)$
Total over d rounds	$O(n+m)$	$O(d \lg n)$

Synopsis

- Graph and their Representation
- Graph Search
- Breadth-First Search
- Depth-First Search

Taking CS Courses

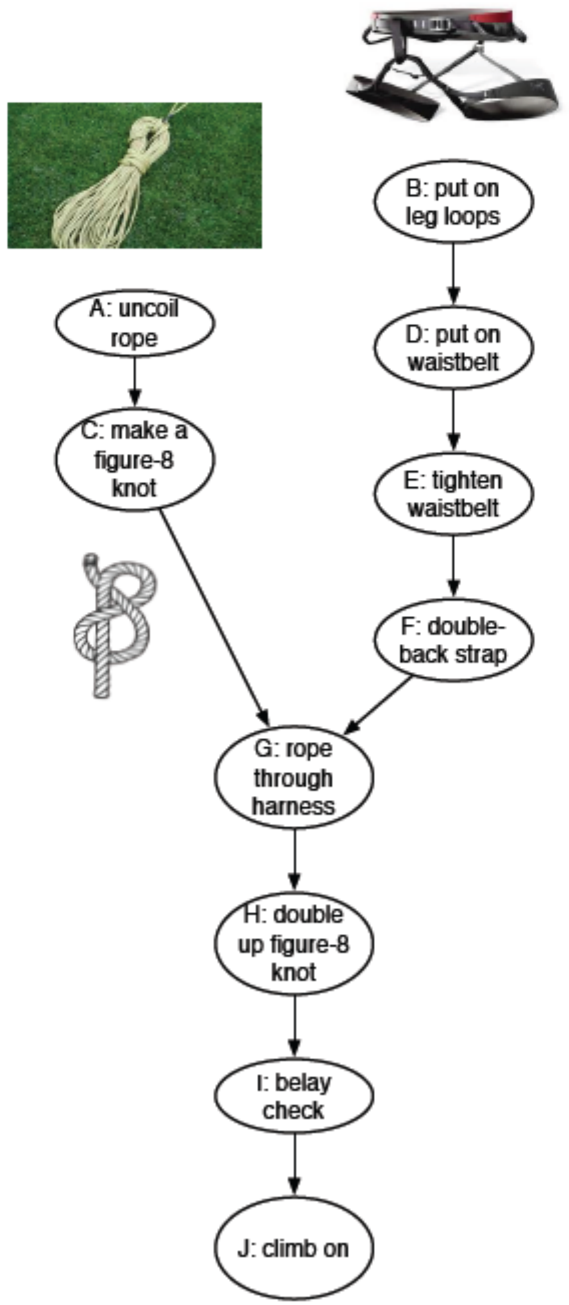
- Take the following courses – but one per semester


$$\begin{tikzcd}
122 \arrow[d] \arrow[rr] & & 210 \arrow[dr] & 150 \arrow[l] & 251 \arrow[dl] \arrow[dr] & \\
213 \arrow[d] & & & 451 & & 359 \\
410 \arrow[d] & & & & & \\
412 & 859 & & 712 & & 750
\end{tikzcd}$$

- What are some possible orders?

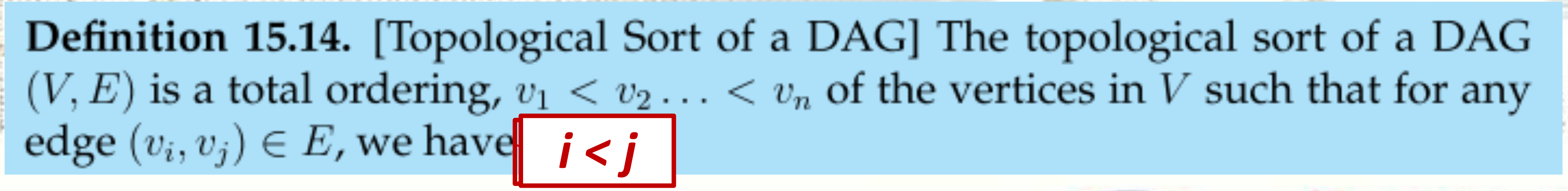
Climber’s Harness

- Since a climber can only perform one of these tasks at a time, her actions are naturally ordered.



Topological Sort

- Directed Acylic Graph (DAG)
  - ➢ A directed acyclic graph is a directed graph with no cycles
- Topological Sort of a DAG
  - ➢ Put vertices in a linear order that respects the graph precedence relationships
  - ➢ The topological sort a DAG (V,E) is a total ordering, $v_1 < v_2 \dots < v_n$ of the vertices in V such that for any edge $(v_i,v_j) \in E$, $i < j$ holds

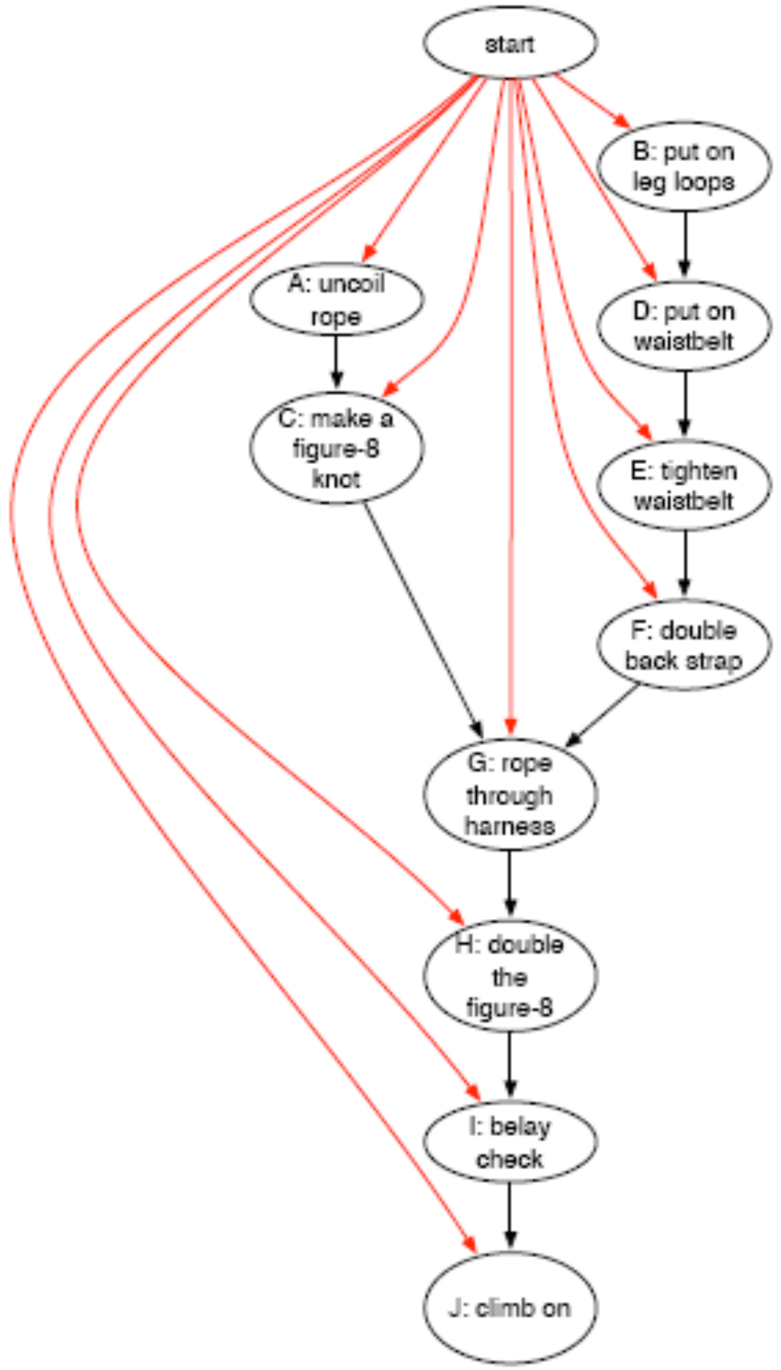


Definition 15.14. [Topological Sort of a DAG] The topological sort of a DAG (V, E) is a total ordering, $v_1 < v_2 \ldots < v_n$ of the vertices in V such that for any edge $(v_i, v_j) \in E$, we have $\boxed{i < j}$

Topological Sort

- When considering the topological sort of a graph, it is often helpful to insert a "start" vertex and connect it to all the other vertices
- Adding a start vertex does not change the set of topological sorts

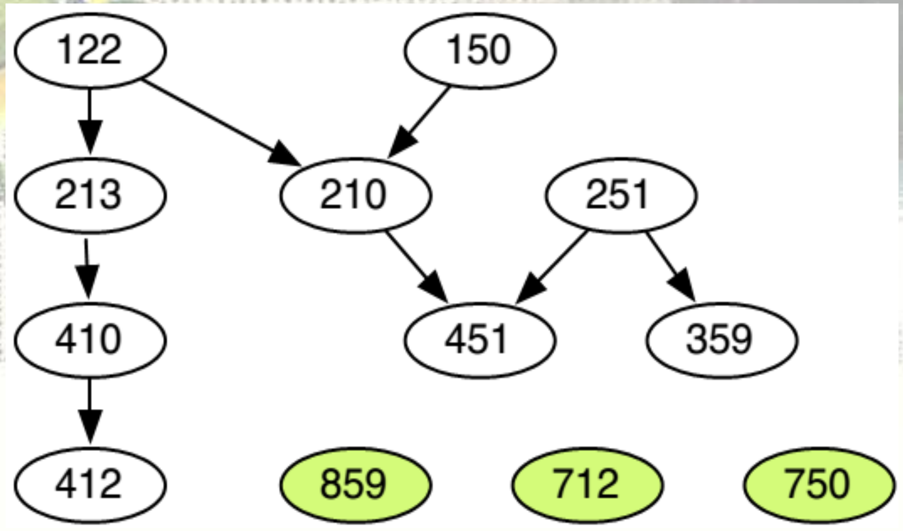
why?



Topological Sort

- How can we know if a schedule is even possible?
  - There should be no cycles!
- Can these problems be solved by Graph Search?
  - BFS is not an effective way to implement topological sort
  - depth-first search (DFS) ?

why?



DFS vs BFS

BFS

- Explores vertices one level at a time
  - ➢ Increases breadth
  - ➢ No backtracking
- Can solve/generate
  - ➢ reachability
  - ➢ connectedness
  - ➢ spanning tree
- Not suitable for topological sort

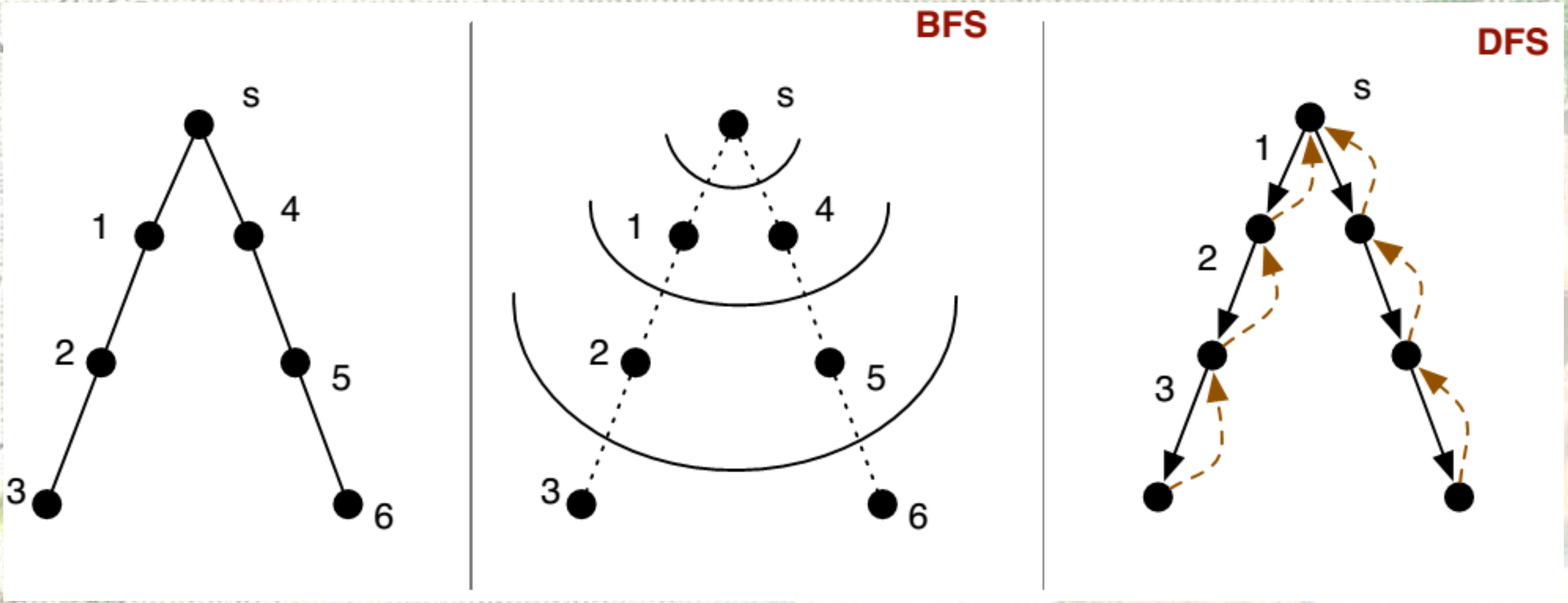
DFS

- Explores vertices one vertex at a time
  - – Increases depth
  - – Backtracking when it can’t go deeper
- Can solve/generate
  - – reachability
  - – connectedness
  - – spanning tree
- Not suitable for shortest unweighted path

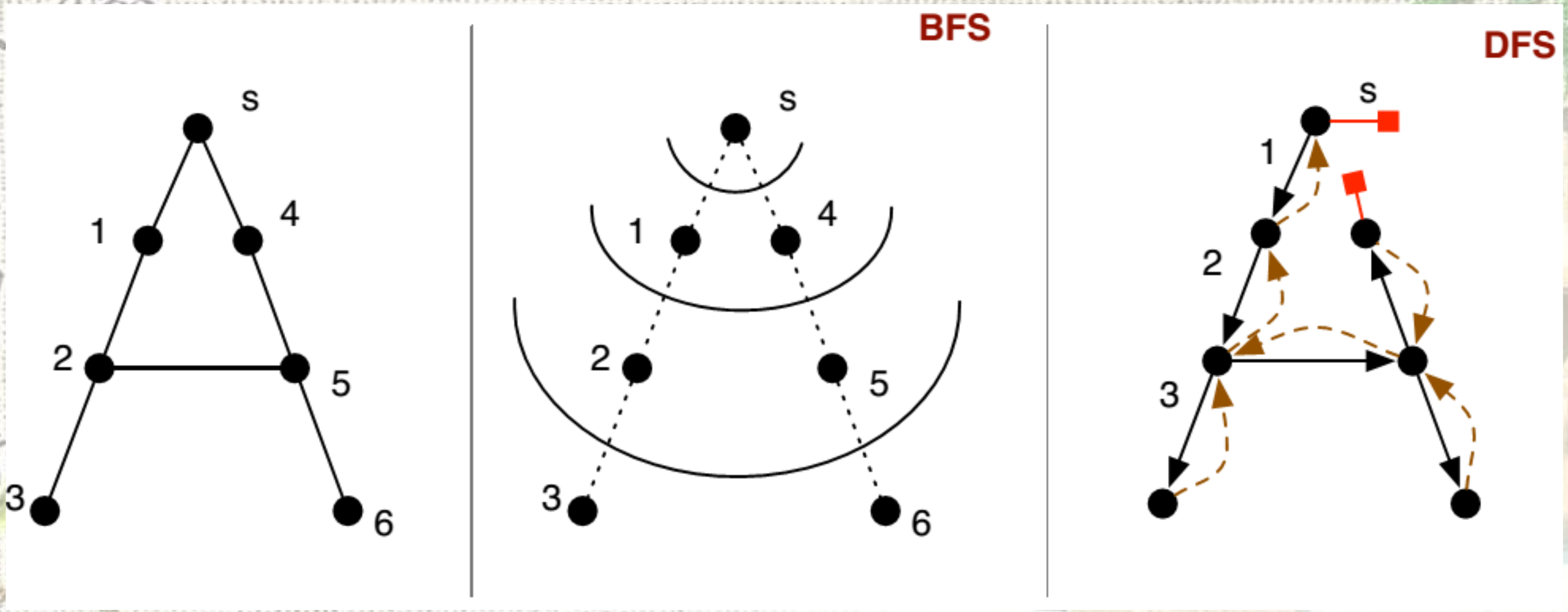
華中科技大學

HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

DFS vs BFS

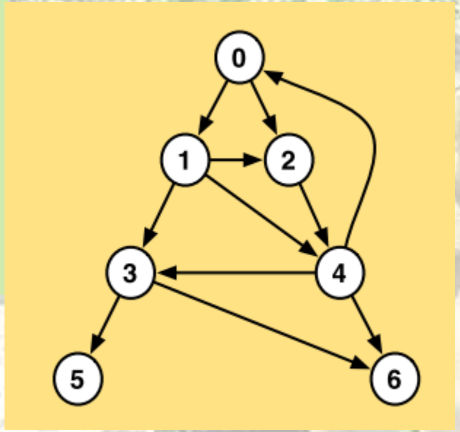


DFS vs BFS



Depth-First Search (DFS)

```
reachability $(G, s)$ =
    let $DFS\ (X, v)$ =
            if $v \in X$ then $X$
            else iterate $DFS\ (X \cup \{v\})\ (N_G(v))$
    in $DFS\ (\{\}, s)$ end
```



- **For example?**

Some Observations

- iter **goes sequentially**
  - **Sets are unordered, ordering depends on implementation!**
- **When a vertex *v* is entered (ENTER *v*) in code**
  - **it picks the "first" outgoing edge *(v,w*$_1$*)***
  - **through** iter **calls** DFS'(**X∪{v}, w_1**)
- **When** DFS'(**X∪{v}, w_1**) **returns**
  - **All vertices reachable from w_1 are explored**
  - **Vertex set returned is**

    $\underline{X_1 = X \cup \{v\} \cup \{\text{All vertices reachable from } w_1\}}$
- iter **picks next edge (v,w_2)and continues**
- **When** iter **is done**

  $\underline{X'' = X \cup \{v\} \cup \{\text{All vertices reachable from } v\}}$

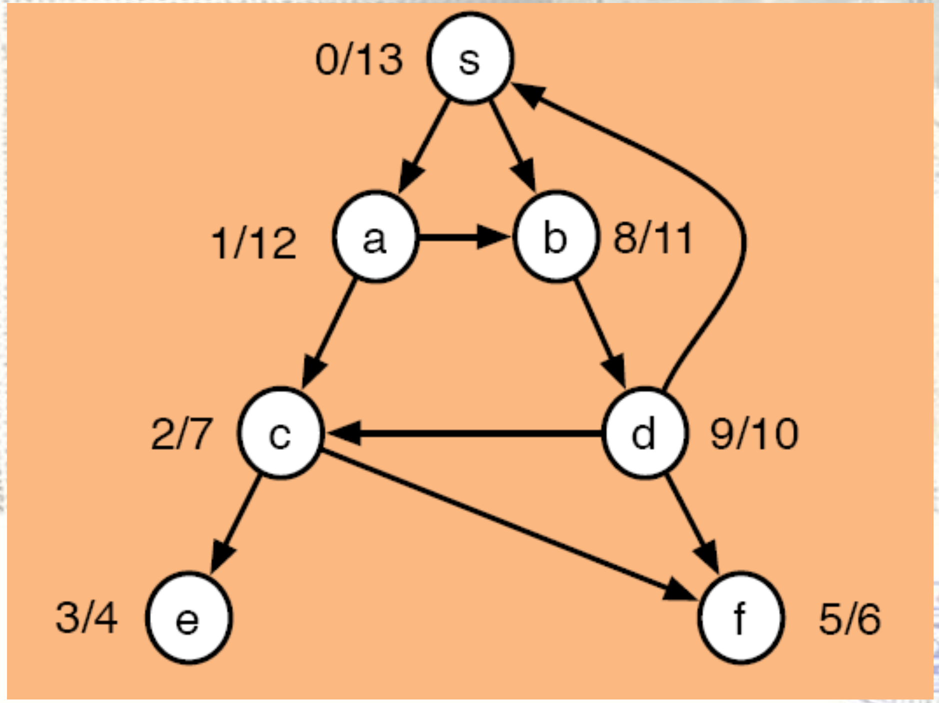
IDEA OF PEBBLING

- **Ones use (white) pebbles to find their way home when left in the middle of a forest by their struggling parents**
  - **When discover one vertex, use white pebble to indicate that has been visited**
  - **The red pebble indicates you are done searching that vertex—i.e. there are no more neighbors who have not been visited**

Why must use two colors pebble?

Timestamp in DFS

- **DFS numbers (t1/t2)**
  - t1: when the vertex gets its white pebble (discovered)
  - t2: when the vertex gets its red pebble (finished)

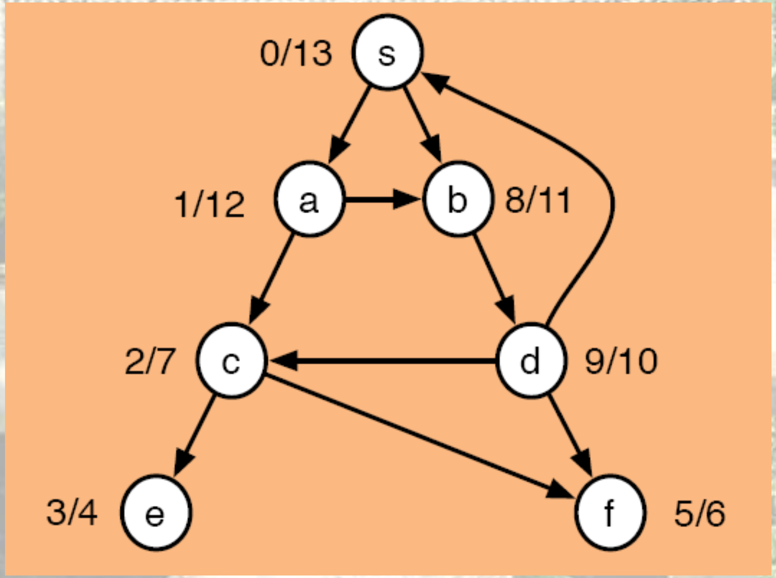


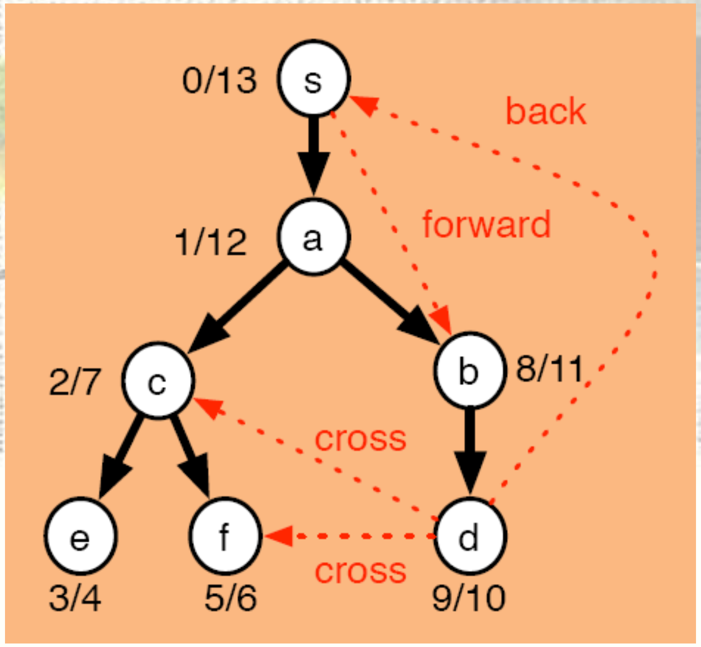
DFS tree

- Tree edges define the DFS tree
  - We call an edge (u, v) a tree edge if v receives its white pebble when the edge (u, v) was traversed
  - The rest of the edges in the graph, which are non-tree edges, can further be classified as back edges, forward edges, and cross edges
    - ✓ A non-tree edge (u, v) is a back edge if v is an ancestor of u in the DFS tree
    - ✓ A non-tree edge (u, v) is a forward edge if v is a descendant of u in the DFS tree
    - ✓ A non-tree edge (u, v) is a cross edge if v is neither an ancestor nor a descendant of u in the DFS tree

DFS tree

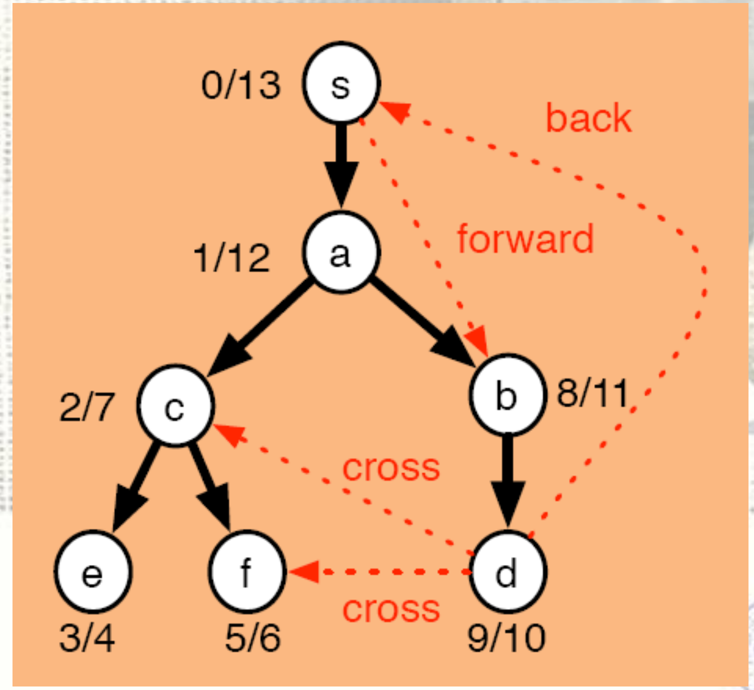
- DFS tree?
- Then, back edges, forward edges, and cross edges?





DFS tree

- How can you determine by just using the DFS numbers of the endpoints of an edge whether it is a cross edge, forward edge, or backward edge?



DFS with discover, finish and revisit

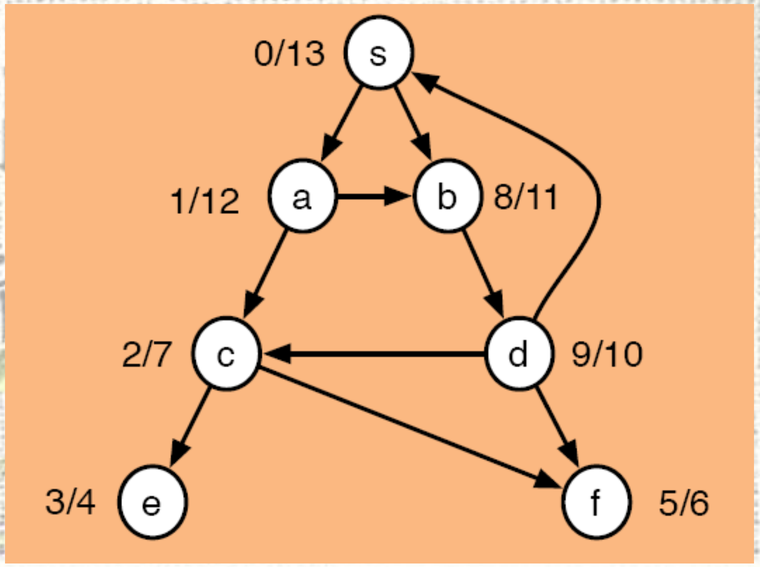
- A version of DFS that corresponds closely to the **DFS numbers**, making explicit the points at which a vertex is **discovered**, **finished**, and **revisited** (if any)

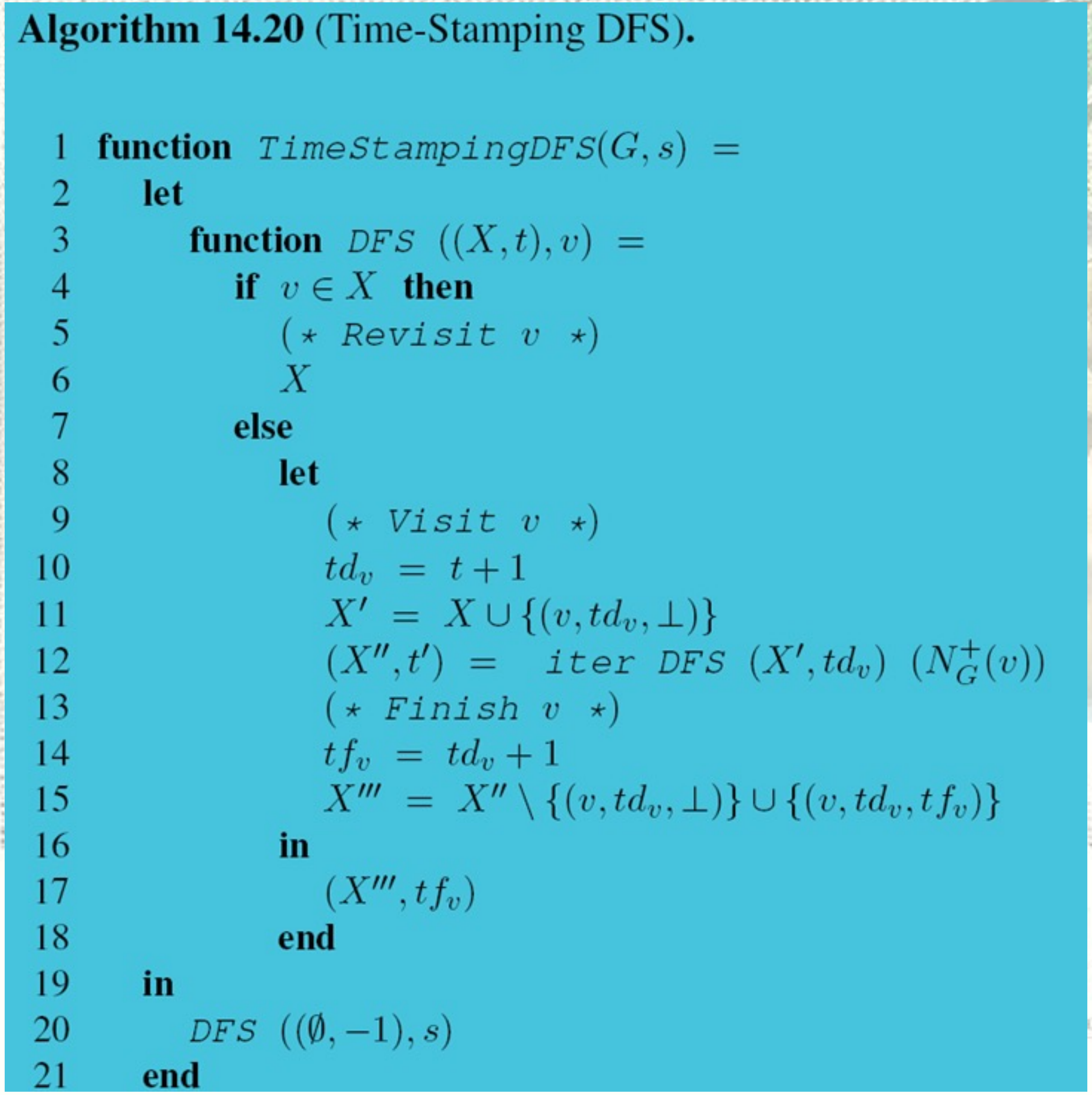
```
$reachability\ (G, s)$ =
    let $DFS\ (X, v)$ =
        if $v \in X$ then $X$    % revisit $v$
        else let
            $X' = X \cup \{v\}$    % discover and visit $v$
            $X'' = iterate\ DFS\ (X \cup \{v\})\ (N_G(v))$
        in $X''$    % finish $v$
    in $DFS\ (\{\}, s)$ end
```

Why not do \X?

Revisiting DFS

- **Can this algorithm get the DFS number?**

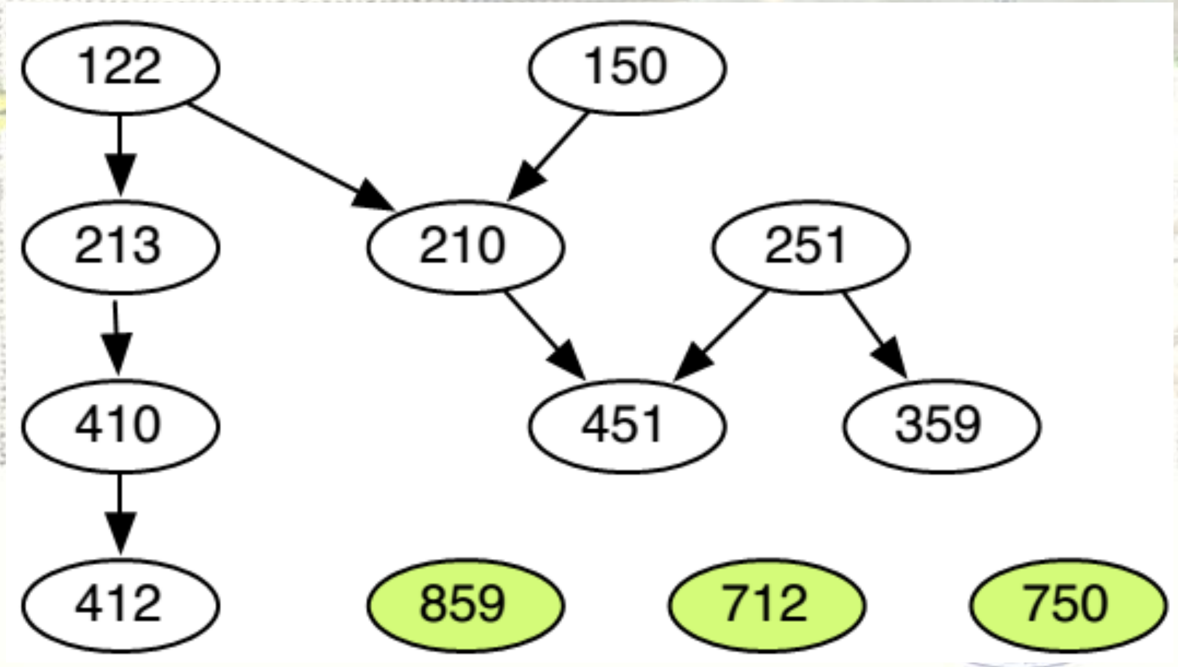



Algorithm 14.20 (Time-Stamping DFS).

```
function $\mathtt{TimeStampingDFS}(G, s)$ =
   let
      function $\mathtt{DFS}\ ((X, t), v)$ =
         if $v \in X$ then
            $(\ast\ \mathtt{Revisit}\ v\ \ast)$
            $X$
         else
            let
               $(\ast\ \mathtt{Visit}\ v\ \ast)$
               $td_v = t + 1$
               $X' = X \cup \{(v, td_v, \bot)\}$
               $(X'', t') = \mathtt{iter\ DFS}\ (X', td_v)\ (N^+_G(v))$
               $(\ast\ \mathtt{Finish}\ v\ \ast)$
               $tf_v = td_v + 1$
               $X''' = X'' \setminus \{(v, td_v, \bot)\} \cup \{(v, td_v, tf_v)\}$
            in
               $(X''', tf_v)$
            end
   in
      $\mathtt{DFS}\ ((\emptyset, -1), s)$
   end
```

Topological Sorting

- **Order the vertices so that the ordering respects reacheability**
  - **If *u* is reachable from *v*, *v* must come earlier in the ordering**



Topological Sort with DFS

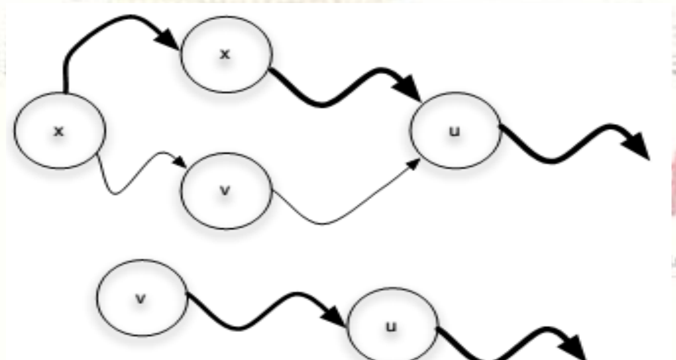
- The DFS numbers have many interesting properties
- possible to use DFS for topological sorting

Lemma

When running DFS on a DAG, if a vertex u is reachable from v then u will finish before v finishes

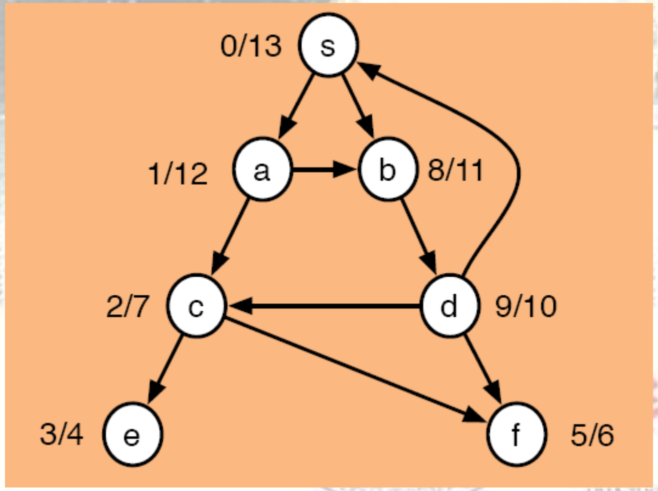
How to prove?

- Assume *u* is reachable from *v*
  - ➢ *u* is entered before *v*. *u* must exit before *v* is entered (otherwise there is a cycle!)
  - ➢ *v* is entered before *u*. *u* will exit first



Topological Sort with DFS

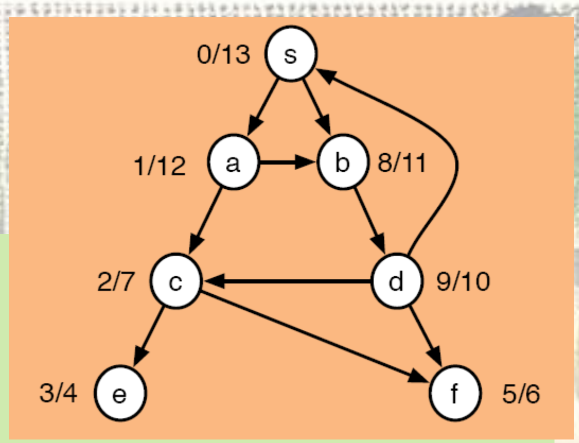
- if we order the vertices by finishing time (latest first), then all vertices reachable from a vertex v will appear after v in the ordering, since they must finish before v finishes
- This is exactly the property we require from a topological sort



Topological Sort With DFS

- Ex: X S v

```
topSort $(G = (V, E))$ =
    let $DFS$ $((X, \underline{S}), v)$ =
        if $v \in X$ then
            $(X, \underline{S})$    % Revisit $v$
        else
            let
                $X' = X \cup \{v\}$    % Discover $v$
                $(X'', \underline{S'})$ = iterate $DFS$ $(X', \underline{S})$ $(N_G^+(v))$
            in $(X'', \underline{cons(v, S')})$ end    % Finish $v$

    in second (iterate $DFS$ $(\{\}, \underline{\langle\,\rangle})$ $V$) end
```



華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Topological Sort With DFS

- Augment with a new source vertex *s*

$$G=(V,E) \rightarrow G'=(V \cup \{s\}, E \cup \{(s,v): v \in V\})$$

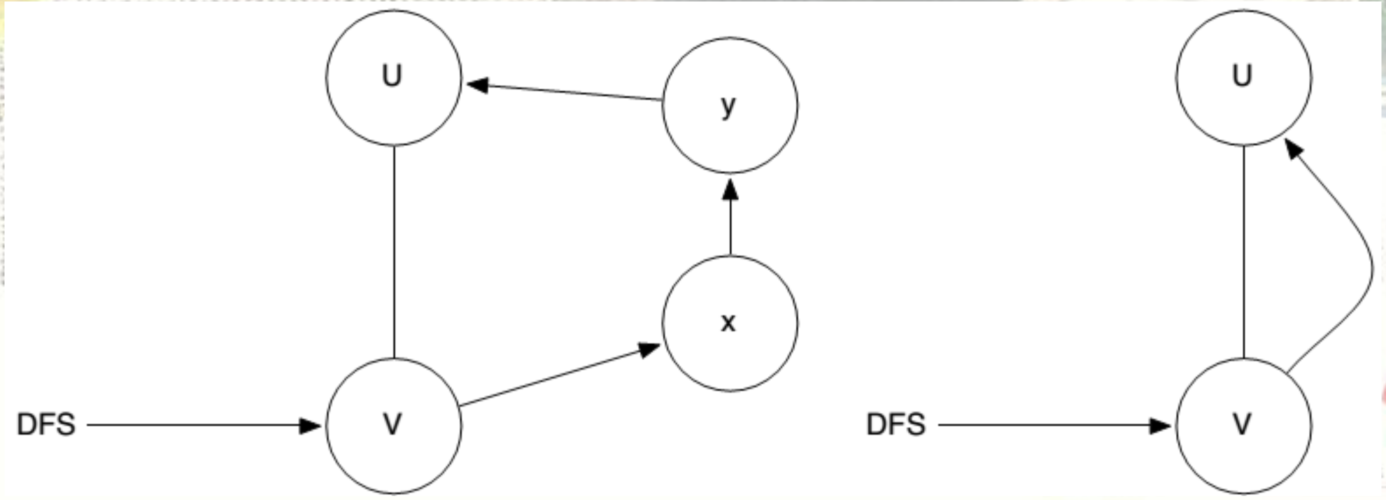
- Why do we need to do this?

CYCLE DETECTION PROBLEM

- Given a graph G = (V, E) cycle detection problem is to determine if there are any cycles in the graph
- The problem is different depending on whether the graph is directed or undirected

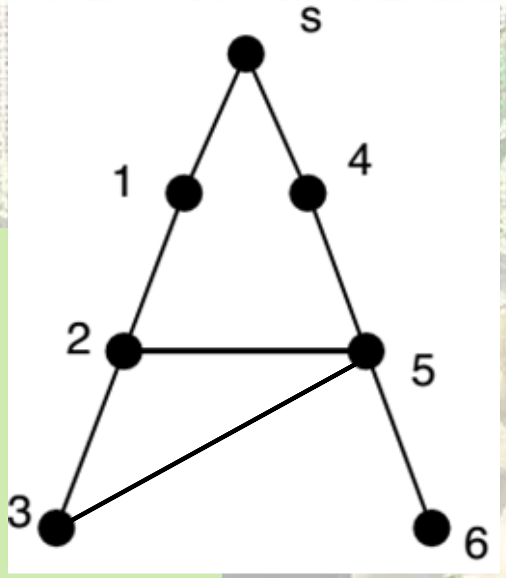
Cycle Detection In Undirected Graphs

- DFS' arrives at a vertex v a second time and the second visit is coming from another vertex u (via the edge (u, v)), what can we conclude?
  - ➢ There must be two paths between *u* and *v* (Why?)
- Not really! In undirected graphs cycles should have length at least 3



Cycle Detection In Undirected Graphs

- **Ex: p X C v**
- **What is p? What is C?**



```
undirectedCycle $(G = (V, E))$ =
  let
    $s$ = a new vertex                      % used as top level parent

    DFS $\underline{p}$ $((X, \underline{C}), v)$ =
      if $(v \in X)$ then
        $(X, \underline{true})$                       % revisit $v$
      else
        let
          $X' = X \cup \{v\}$                % discover $v$
          $(X'', \underline{C'})$ = iterate (DFS $\underline{v}$) $(X', \underline{C})$ $(N_G(v)\underline{\setminus\{p\}})$
        in $(X'', \underline{C'})$ end                  % finish $v$

  in second (iterate (DFS $\underline{s}$) $(\{\}, \underline{false})$ $V$) end
```

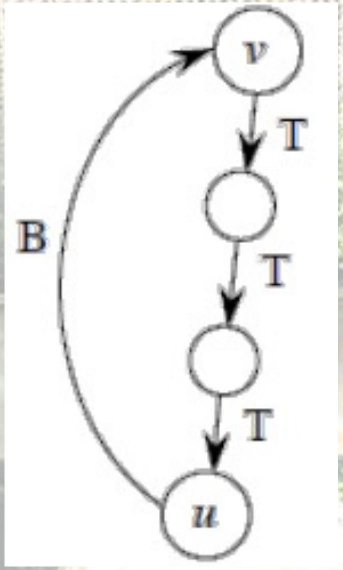
Cycle Detection In Undirected Graphs

```
undirectedCycle $(G = (V, E))$ =
  let
    s = a new vertex          % used as top level parent

    DFS $\underline{p}$ $((X, \underline{C}), v)$ =
      if $(v \in X)$ then
        $(X, \underline{true})$                    % revisit $v$
      else
        let
          $X' = X \cup \{v\}$              % discover $v$
          $(X'', \underline{C'})$ = iterate (DFS $\underline{v}$) $(X', \underline{C})$ $(N_G(v)\underline{\setminus\{p\}})$
        in $(X'', \underline{C'})$ end                % finish $v$

  in second (iterate (DFS $\underline{s}$) $(\{\}, \underline{false})$ $V$) end
```

- *C* keeps tracks of cycles
- *p* is the parent – removed from neighbors and curried!

Back Edges In A DFS Search

- A **back edge** goes from a vertex *u* to an ancestor *v* in the DFS tree



Theorem

A directed graph *G=(V,E)* has a cycle if and only if for *G'* = (*V*∪{*s*}, *E*∪{(*s*,*v*): *v*∈*V*}) a DFS from *s* has a back edge

Cycle Detection In Directed Graphs

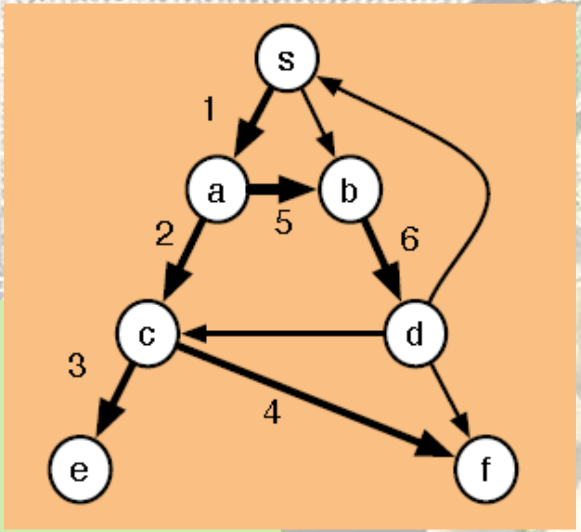
- **Important preprocessing step in Topological Sort**
  - **Topological sort will return garbage when graph has cycles**
- **We augment the graph with a node *s* with an edge to every other vertex the graph**
  - **This can not add cycles. Nothing comes into *s***

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Cycle Detection In Directed Graphs

- **Ex: X Y C v**
- **What is Y?**

```
directedCycle $(G = (V, E))$ =
    let DFS $\underline{Y}$ $((X, \underline{C}), v)$ =
        if $(v \in X)$ then
            $(X, \underline{C \vee Y[v]})$                % revisit $v$
        else let
            $X' = X \cup \{v\}$                % discover $v$
            $\underline{Y' = Y \cup \{v\}}$
            $\underline{(X'', C')}$ = iterate (DFS $Y'$) $(X', \underline{C})$ $(N_{G'}(v))$
        in $(X'', \underline{C'})$ end                % finish $v$

    in second(iterate (DFS {}) ({}, $\underline{false}$) $V$) end
```

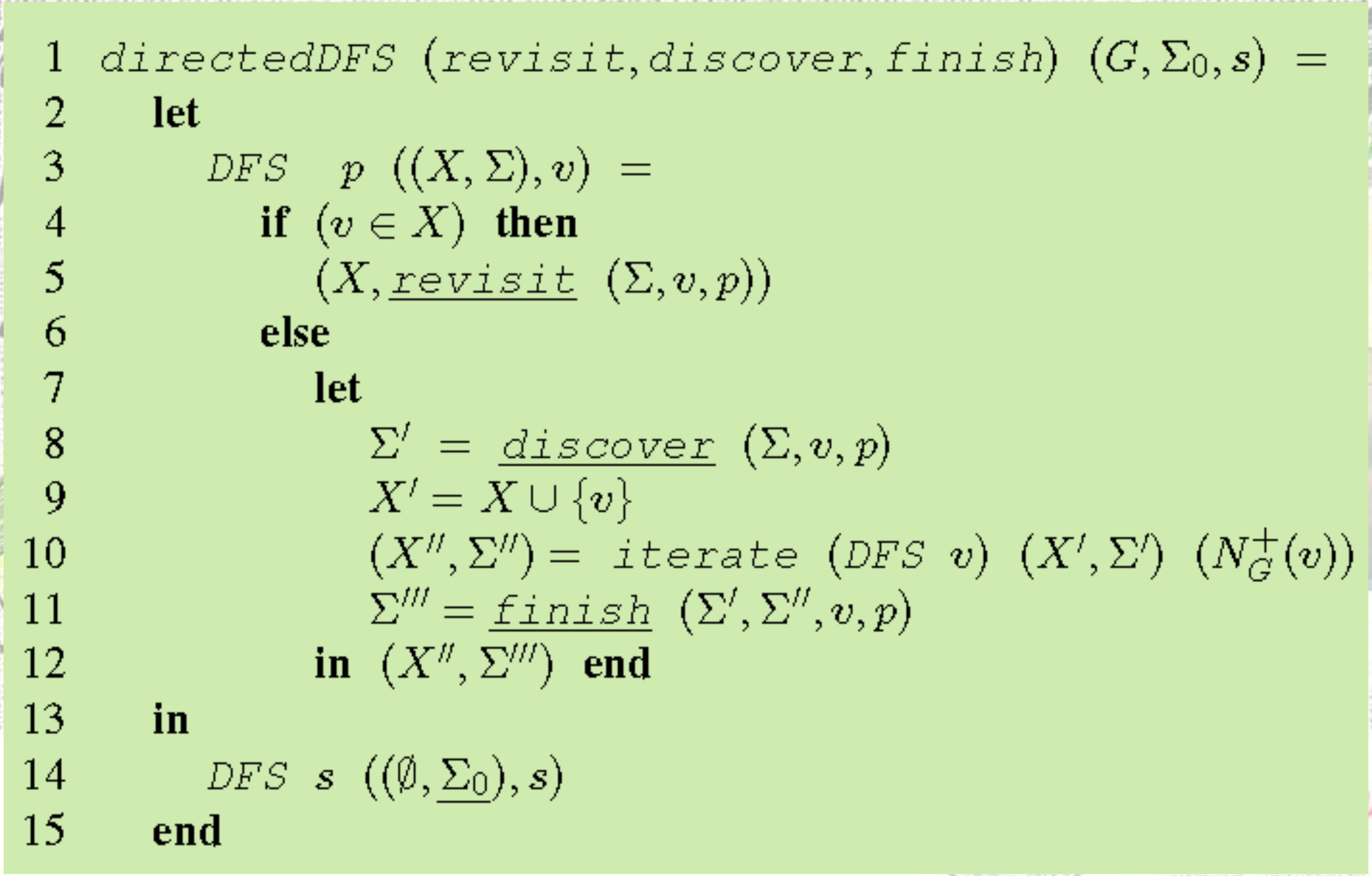


EX： X Y C v

Generalizing DFS

- All DFS code seem very much alike
- They do work on
  - Discovering
  - Finishing
  - Revisiting
- We need to keep some state σ around
  - and update it appropriately!
- each function
  - the current vertex v
  - the parent vertex p in the DFS tree
  - an updated state

Generalized directed DFS



```
$\mathtt{directedDFS}\ (\mathtt{revisit}, \mathtt{discover}, \mathtt{finish})\ (G, \Sigma_0, s)\ =$
    let
        $\mathtt{DFS}\ \ p\ ((X, \Sigma), v)\ =$
            if $(v \in X)$ then
                $(X, \underline{\mathtt{revisit}}\ (\Sigma, v, p))$
            else
                let
                    $\Sigma'\ =\ \underline{\mathtt{discover}}\ (\Sigma, v, p)$
                    $X' = X \cup \{v\}$
                    $(X'', \Sigma'') =\ \mathtt{iterate}\ (\mathtt{DFS}\ v)\ (X', \Sigma')\ (N_G^+(v))$
                    $\Sigma''' = \underline{\mathtt{finish}}\ (\Sigma', \Sigma'', v, p)$
                in $(X'', \Sigma''')$ end
    in
        $\mathtt{DFS}\ s\ ((\emptyset, \underline{\Sigma_0}), s)$
    end
```

Undirected Cycle Detection

```
$\mathtt{directedDFS}\ (\mathtt{revisit}, \mathtt{discover}, \mathtt{finish})\ (G, \Sigma_0, s) =$
    let
        $\mathtt{DFS}\ \ p\ ((X, \Sigma), v)\ =$
            if $(v \in X)$ then
                $(X, \underline{\mathtt{revisit}}\ (\Sigma, v, p))$
            else
                let
                    $\Sigma' = \underline{\mathtt{discover}}\ (\Sigma, v, p)$
                    $X' = X \cup \{v\}$
                    $(X'', \Sigma'') = \mathtt{iterate}\ (\mathtt{DFS}\ v)\ (X', \Sigma')\ (N^+_G(v))$
                    $\Sigma''' = \underline{\mathtt{finish}}\ (\Sigma', \Sigma'', v, p)$
                in $(X'', \Sigma''')$ end
    in
        $\mathtt{DFS}\ s\ ((\emptyset, \underline{\Sigma_0}), s)$
    end
```

```
$\Sigma_0 = \mathtt{false} : \mathtt{bool}$
$\mathtt{revisit}(\_) = \mathtt{true}$
$\mathtt{discover}(\mathtt{fl}, \_, \_) = \mathtt{fl}$
$\mathtt{finish}(\_, \mathtt{fl}, \_, \_) = \mathtt{fl}$
```

Topological Sort

```
$\mathtt{directedDFS}\ (\mathtt{revisit}, \mathtt{discover}, \mathtt{finish})\ (G, \Sigma_0, s) =$
    let
        $\mathtt{DFS}\ \ p\ ((X, \Sigma), v) =$
            if $(v \in X)$ then
                $(X, \underline{\mathtt{revisit}}\ (\Sigma, v, p))$
            else
                let
                    $\Sigma' = \underline{\mathtt{discover}}\ (\Sigma, v, p)$
                    $X' = X \cup \{v\}$
                    $(X'', \Sigma'') = \mathtt{iterate}\ (\mathtt{DFS}\ \ v)\ (X', \Sigma')\ (N_G^+(v))$
                    $\Sigma''' = \underline{\mathtt{finish}}\ (\Sigma', \Sigma'', v, p)$
                in $(X'', \Sigma''')$ end
    in
        $\mathtt{DFS}\ s\ ((\emptyset, \underline{\Sigma_0}), s)$
    end
```

$\Sigma_0 = [\ ] : \mathtt{vertex\ list}$
$\mathtt{revisit}(L, \_, \_) = L$
$\mathtt{discover}(L, \_, \_) = L$
$\mathtt{finish}(\_, L, v, \_) = v :: L$

Directed Cycle Detection

```
$\mathtt{directedDFS}\ (\mathtt{revisit}, \mathtt{discover}, \mathtt{finish})\ (G, \Sigma_0, s)\ =$
  let
    $\mathtt{DFS}\ \ p\ ((X, \Sigma), v)\ =$
      if $(v \in X)$ then
        $(X, \underline{\mathtt{revisit}}\ (\Sigma, v, p))$
      else
        let
          $\Sigma' \ = \ \underline{\mathtt{discover}}\ (\Sigma, v, p)$
          $X' = X \cup \{v\}$
          $(X'', \Sigma'') = \ \mathtt{iterate}\ (\mathtt{DFS}\ \ v)\ (X', \Sigma')\ (N_G^+(v))$
          $\Sigma''' = \underline{\mathtt{finish}}\ (\Sigma', \Sigma'', v, p)$
        in $(X'', \Sigma''')$ end
  in
    $\mathtt{DFS}\ \ s\ ((\emptyset, \underline{\Sigma_0}), s)$
  end
```

$\Sigma_0 = (\{\}\,, \mathtt{false}) : \mathit{Set} \times \mathtt{bool}$

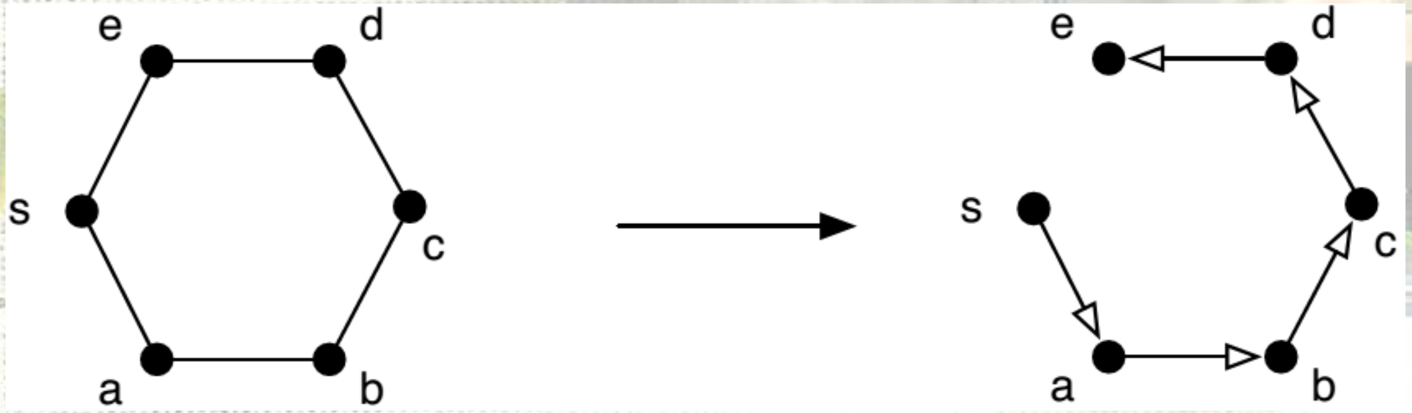
$\mathtt{revisit}\ \ ((S, \mathtt{fl}), v, \_)\ =\ (S,\ \ \mathtt{fl} \vee (S[v]))$

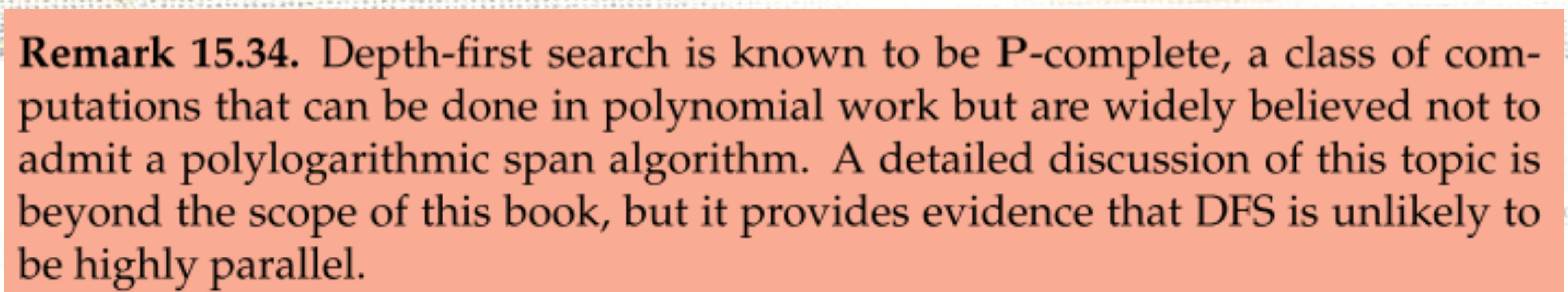
$\mathtt{discover}\ \ ((S, \mathtt{fl}), v, \_)\ =\ (S \cup \{v\}\,, \mathtt{fl})$

$\mathtt{finish}\ \ ((S, \_), (\_, \mathtt{fl}), v, \_)\ =\ (S, \mathtt{fl})$

DFS With Parallelism

- Can we do all outgoing edges in parallel?
  - Yes - if parallel searches never meet up (then we really have a tree!)
  - No - otherwise





Remark 15.34. Depth-first search is known to be P-complete, a class of computations that can be done in polynomial work but are widely believed not to admit a polylogarithmic span algorithm. A detailed discussion of this topic is beyond the scope of this book, but it provides evidence that DFS is unlikely to be highly parallel.

Cost of DFS

Lemma

For a graph *G=(V,E)* with *m* out edges and *n* vertices:

- DFS in Algorithms 15.15 and 15.24 will be called at most n + m times
- a vertex will be discovered (and finished) at most n times

why?

- **every vertex can only be discovered once (add to X)**
- **every out edge will only be traversed once, invoking a call to DFS**
- **In topological sort, an additional n initial calls are made starting at each vertex**

Cost of DFS

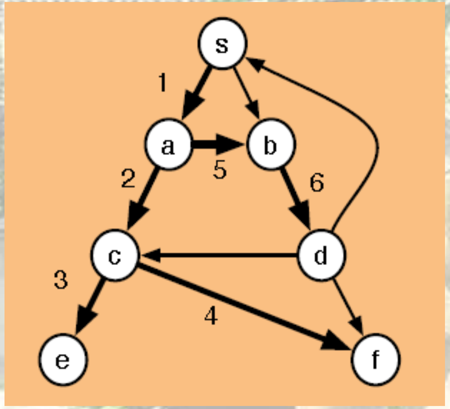
Corollary

The DFS algorithm on a graph with m out edges, and n vertices, and using the tree-based cost specification for sets runs in O((m+n) log n) work and span

- **Using ST sequences reduces work and span to *O(m+n)***

DFS With ST Sequences

```
directedDFS $(G:(int\ seq)\ seq, \Sigma_0 : \alpha,\ s:int)$ =
  let
    DFS p $((X : bool\ stseq,\ \Sigma : \alpha,\ v:int)$ =
      if $(X[v])$ then
        $(X, revisit\ (\Sigma, v, p))$
      else
        let
          $X' = STSeq.update\ X\ (v, true)$
          $\Sigma' = discover\ (\Sigma, v, p)$
          $(X'', \Sigma'') = iterate\ (DFS\ v)\ (X', \Sigma')\ (G[v])$
          $\Sigma''' = finish\ (\Sigma', \Sigma'', v, p)$
        in $(X'', \Sigma''')$ end
    $X_{init} = STSeq.fromSeq\langle\ false : v \in \langle 0, \ldots, |G| - 1\rangle\rangle$
  in
    DFS s $((X_{init}, \Sigma_0), s)$
  end
```



- $O(m)$ work and span, **why?**
- Use regular sequence, $O(n^2)$ work, $O(m)$ span, **why?**

Exercises

- Prove that for a graph with n vertices and m edges, $O(\lg m) = O(\lg n)$
- What is the cost of deleting a vertex with out-degree d?
- What is the cost of finding the out-neighbors of a vertex?
- Why does the cost of mapping over all edges require $\Omega(n)$ work?
- Give a constant-span algorithm for deleting an edge from a graph.
- Give a constant-span algorithm for computing the complement of a graph.
- Does the algorithm visit all the vertices in the graph?
- Present a multi-source version of the single-source graph search algorithm.
- The computation of the new F is not quite defined in the same way as in the generic graph search. Prove that the technique used here is consistent with that of the generic algorithm.
- Prove that the algorithm is correct, i.e., visits all reachable vertices from the source in the order of their distances to the source

Exercises

- Prove that the queue based implementation of sequential BFS is correct.
- In general, from which frontiers could the vertices in $N_G(F_i)$ come when the graph is undirected? What if the graph is directed?