Topic 18: Basic Graph Algorithms
(CLRS Appendix B.4-B.5, 22)

Fall 2001

1 Graph Problems

- During the next couple of weeks we will discuss graph algorithms.
- We start with a review of the basic definitions and a few fundamental graph algorithms.

1.1 Definitions

- A graph \( G = (V, E) \) consists of a finite set of vertices \( V \) and a finite set of edges \( E \).
  - Directed graph (DAG): \( E \) is a set of ordered pairs of vertices \( (u, v) \) where \( u, v \in V \)
    \[ V = \{1, 2, 3, 4, 5, 6\} \]
    \[ E = \{(1,2), (2,3), (2,5), (4,1), (4,5), (5,4), (6,3)\} \]
  - Undirected graph: \( E \) is a set of unordered pairs of vertices \( \{u, v\} \) where \( u, v \in V \)
    \[ V = \{1, 2, 3, 4, 5, 6\} \]
    \[ E = \{(1,2), (1,5), (2,5), (3,6)\} \]

- Edge \( (u, v) \) is incident to \( u \) and \( v \)

- Degree of vertex in an undirected graph is the number of edges incident to it.
- In (out) degree of a vertex in a directed graph is the number of edges entering (leaving) it.

- A path from \( u_1 \) to \( u_2 \) is a sequence of vertices \( (u_1 = v_0, v_1, v_2, \ldots, v_k = u_2) \) such that \( (v_i, v_{i+1}) \in E \) (or \( (v_i, v_{i+1}) \in E \))

- We say that \( u_2 \) is reachable from \( u_1 \)
- The length of the path is \( k \)
- It is a cycle if \( v_i = u_1 \)

- An undirected graph is connected if every pair of vertices are connected by a path
  - The connected components are the equivalence classes of the vertices under the “reachability” relation. (All connected pair of vertices are in the same connected component)
  - A directed graph is strongly connected if every pair of vertices are reachable from each other
    - The strongly connected components are the equivalence classes of the vertices under the “mutual reachability” relation.
    - In the DAG pictured earlier, there are three strongly connected components. The subgraph induced by vertices \( \{1, 2, 4, 5\} \) is strongly connected and it forms a strongly connected component. The other two strongly connected components consist of the single sets \( \{3\} \) and \( \{6\} \).

- Graphs appear all over the place in all kinds of applications, e.g:
  - Trees \( |E| - |V| - 1 \)
  - Connectivity/dependencies (house building plans, WWW page connections, …)
- Often the edges \( (u, v) \) in a graph have weights \( w(u, v) \), e.g.
  - Road networks (distances)
  - Cable networks (capacity)

1.2 Representation

- Adjacency list representation:
  - Array of \(|V|\) list of edges incident to each vertex

Examples:

- Graph adjacency list:
  - Array of \(|V|\) list of edges incident to each vertex

CPS 230

Topic 18: Basic Graph Algorithms
Note: For undirected graphs, every edge is stored twice. Hence, space is $O(|V| + 2|E|) = O(|V| + |E|)$.

If graph is weighted, a weight is stored with each edge.

**Adjacency matrix representation**

- $|V| \times |V|$ matrix $A$ where

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

Examples:

Note: For undirected graphs, the adjacency matrix is symmetric along the main diagonal (i.e., $A^T = A$).

If graph is weighted, weights are stored in the adjacency matrix instead of 1s.

**Comparison of matrix and list representation**

### 2 Graph traversal

- There are two standard (and simple) ways of traversing all vertices/edges in a graph in a systematic way:
  - **Breadth-first**
  - **Depth first**
- We can use them in many fundamental algorithms, e.g., finding cycles, connected components, ...

#### 2.1 Breadth-first search (BFS)

- **Main idea**
  - Start by visiting some source vertex $s$,
  - Then visit all vertices at distance 1,
  - Then visit all vertices at distance 2,
  - Then visit all vertices at distance 3,
  - ...
- **BFS** corresponds to computing shortest path distance (in terms of the number of edges) from $s$ to all other vertices.
- To control progress of our BFS algorithm, we think about **coloring** each vertex
  - **White** before we start,
  - **Gray** after we visit the vertex but before we have visited all its adjacent vertices,
  - **Black** after we have visited the vertex and all its adjacent vertices (all adjacent vertices are gray).
- We use a FIFO queue $Q$ to hold all gray vertices — vertices we have seen but are still not done with.
- We remember from which vertex a given vertex $v$ is colored gray (visit $v$).
• Algorithm:

BFS(s)

1. color[s] = gray
2. d[s] = 0
3. ENQUEUE(Q, s)
4. WHILE Q not empty DO
   5. DEQUEUE(Q, u)
   6. FOR (u, v) ∈ E DO
      7. IF color[v] = white THEN
          8. color[v] = gray
          9. d[v] = d[u] + 1
          10. ENQUEUE(Q, v)
   11. FI
12. FI
13. OD

• Algorithm runs in O(|V| + |E|) time

• Note
  - The edges (visit[v], v), for all v ∈ V form a tree called the BFS tree.
  - d[v] contains length of shortest path (in terms of the number of edges) from s to v.
  - We can use the visit array to find the shortest path from s to any given vertex v, by tracing the path backwards from v, visit[v], visit[visit[v]], ...

• If graph is not connected we have to try to start the traversal at all nodes.

FOR each vertex u ∈ V DO

1. IF color[u] = white THEN BFS(u)

OD

• Note: We can use algorithm to compute connected components in O(|V| + |E|) time.

2.2 Depth-first search (DFS)

• If we use a stack instead of a FIFO queue Q, we get another traversal order: depth-first search

  - We explore “as deeply as possible”.
  - Backtrack until we find an unexplored adjacent vertex.
  - Explore as deeply as possible, ...

• Often we are interested in “discovery time” and “finish time” of vertex u

  - Discovery time \(d[u]\): indicates at what “time” vertex u is first visited.
  - Finish time \(f[u]\): indicates at what “time” all adjacent vertices of vertex u have been visited.

• Instead of using a stack in a DFS algorithms, we can write a recursive procedure
DFS: How it works

- Initialize all vertices to white
- Reset global counter
- Check each vertex; visit each *white* vertex using DFS
- Each call to DFS(u) roots a new tree of depth-first forest at vertex u
- Vertex is *gray* if it has been discovered, but not all its edges have been explored!
- *gray* edges always form a linear chain!
- Vertex is *black* after all its edges are explored
- When DFS returns, every vertex u is assigned:
  1. a discovery time $d[u]$, and
  2. a finishing time $f[u]$

---

- We will color a vertex gray when we first meet it and black when we finish processing all adjacent vertices.

- Algorithm:

```c
DFS(u)
\begin{align*}
\text{color}[u] & \leftarrow \text{gray} \\
d[u] & \leftarrow \text{time} \\
\text{time} & \leftarrow \text{time} + 1 \\
\text{FOR } (u,v) \in E \text{ DO} \\
& \text{IF color}[v] = \text{white} \text{ THEN} \\
& \text{visit}[v] \leftarrow u \\
& \text{DFS}(v) \\
& \text{FI} \\
\text{color}[u] & \leftarrow \text{black} \\
d[u] & \leftarrow \text{time} \\
\text{time} & \leftarrow \text{time} + 1
\end{align*}
```

- Algorithm runs in $O(|V| + |E|)$ time

- As before we can extend algorithm to unconnected graphs and we can use it to find connected components in $O(|V| + |E|)$ time,

```c
\text{FOR each vertex } u \in V \text{ DO} \\
\text{IF color}[u] = \text{white} \text{ THEN DFS}(u) \\
\text{OD}
```

- As previously, the edges (visit[v], u), for all $v \in V$ form a tree called the *DFS tree*.
**DFS: Running time**

Running time $O(|V|^2)$, because
DFS called once per vertex
Each loop over $Adj$ runs $< |V|$ times.
But... can we show a better bound?

- **Amortized bookkeeping:** charge exploration of edge to the edge:
  
  Charge DFS loop body to edge (runs once per edge if directed graph, twice if undirected)
  Charge rest of DFS to vertex (runs once per vertex)

- Time = $O(|V| + |E|)$, which is *linear time*

$O(|V| + |E|)$ is considered linear time for graph because it is linear in size of adjacency-list representation!

**DFS Timestamping**

The procedure DFS records:

- discovery time of vertex $u$ in $d[u]$
- finishing time of vertex $u$ in $f[u]$

For every vertex $u$,

$$d[u] < f[u].$$
DFS Example

Inside each node above,

- each gray vertex is labeled by its discovery time, and
- each black vertex is labeled by both its discovery time and its finish time.
**DFS: Structure of colored vertices**

Vertex $u$ is:
- *white* before time $d[u]$
- *gray* between time $d[u]$ and time $f[u]$
- *black* thereafter.

Also notice structure throughout algorithm:
- *gray* vertices form a linear chain.
  - stack of recursive calls
    *(things started but not yet finished)*

**DFS: parenthesis theorem**

Discovery, finish times have **parenthesis structure**.

- represent discovery of $u$ with left parenthesis “$(u)$”
- represent finishing $u$ by right parenthesis “$u$)”
- history of discoveries and finishings makes a well-formed expression! (Parentheses are properly nested.)
- If $v$ is a descendant of $u$ in the DFS tree, then
  

Proof in CLRS (omitted here); intuition:
  Intervals either disjoint or enclosed, but never (otherwise) overlap
We’ll just look at example.
### DFS and Parenthesization

(a) Graph representation with vertices labeled and edges marked.

### Edge Classification

**Tree edge**: (gray to white)
- encounter new (white) vertex
- Form spanning forest (no cycles)

**Back edge**: (gray to gray)
- from descendant to ancestor

**Forward edge**: (gray to black)
- non-tree, from ancestor to descendant

**Cross edge**: (gray to black)
- remainder — between trees or subtrees
  - (if same tree, can’t go anc/desc, or desc/anc)
DFS: edge classification

Notes:
- ancestor/descendant is with respect to tree edges
- tree and back edges are important;
- most algorithms don’t distinguish between forward and cross edges

Exercise:
- How to distinguish forward, cross edges in DFS? (Hint: look at discovery times.)

DFS: Lemma

Theorem 22.10:

In a depth-first search of an undirected graph $G$, every edge of $G$ is either a tree edge or a back edge.

Sketch of proof:
**DFS: Lemma**

**Theorem 22.10:**

*Proof:*

▷ Suppose there’s a forward edge \( F' \) (at left)
But \( F' \) edge must actually be \( B \) because we must finish processing bottom vertex before resuming with top vertex.

▷ Suppose there’s a cross edge \( C' \) between subtrees (at right)
\( C' \) edge can’t be Cross edge:
It must be explored from its first endpoint to be explored, in which case the other endpoint isn’t yet explored, and the edge becomes a T edge instead of a C edge.
The search continues beyond the other endpoint, and the T edge coming out of the other endpoint changes to a B edge.
Exercise

Can use DFS to find cycles in undirected graphs!

An undirected graph is acyclic (i.e., a forest) iff a DFS yields no back edges.

• Proof that acyclic ⇒ no back edge:
  trivial (back edge ⇒ cycle)

• Proof that no back edges ⇒ acyclic:
  No back edges ⇒ only tree edges (by above lemma)
  ⇒ forest ⇒ acyclic

Exercise

We can thus run DFS: if find a back edge, then we can stop and report that there’s a cycle

• Time $O(|V|)$, [not $O(|V| + |E|)$ !]

If ever see $|V|$ distinct edges, must have seen a back edge, because in acyclic (undirected) forest,
$|E| \leq |V| - 1.$
Directed Acyclic Graphs (DAGs)

- No directed cycles
  example:

  ![Directed Acyclic Graph Example](image)

- Used in many applications to indicate precedences among events
- Example: parallel code execution
  - Topological Sort (induce a total ordering)

DAG: Theorem

Theorem: A directed graph $G$ is acyclic iff a DFS yields no back edges.

$\Rightarrow$: back edge $\Rightarrow$ cycle

$\Leftarrow$: Contrapositive: cycle $\Rightarrow$ back edge

Suppose $G$ has a cycle. Let $v$ have lowest discovery # on cycle, and let $u$ be predecessor on cycle.

$$
\begin{align*}
  u &\rightarrow v \\
  \uparrow \quad \cdots \quad \downarrow \\
  (v \text{ is first vertex visited})
\end{align*}
$$

When $v$ discovered, whole cycle is white.

Must visit everything reachable on a white path from $v$ before returning from DFS($v$).

Thus $(u, v)$ is a back edge. \(\blacksquare\)

- $O(|V| + |E|)$ time [Why not $O(|V|)$ as before?]
Topological Sort

The following algorithm topologically sorts a DAG:

**Topological-Sort(G)**
1. call DFS(G) to compute finishing times f[v] for each vertex v
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices

At end, linked list comprises total ordering!

If the graph has a cycle, then no linear ordering is possible!
**Topological Sort: Example**

Example: precedence relations (don $x$ before $y$)

Intuition: Can “schedule” task only when all of its follow-on tasks have been scheduled. The task is scheduled earlier than its follow-on tasks.

---

**Topological Sort: running time**

Running Time:

- depth-first search: takes $O(|V| + |E|)$ time
- insert each of the $|V|$ vertices onto the front of the linked list: takes $O(1)$

We can perform a topological sort in time $O(|V| + |E|)$. 
Topological Sort: correctness

Correctness proof for \textsc{Topological-Sort}(G)

Claim: \((u, v) \in E \Rightarrow f[u] > f[v]\)

When \((u, v)\) explored, \(u\) is \textit{gray}

If \(v = \text{gray}\)
\[
\Rightarrow (u, v) = \text{backedge (cycle, contradiction)}.
\]

If \(v = \text{white}\)
\[
\Rightarrow v \text{ becomes descendant of } u
\Rightarrow f[v] < f[u]
\]

If \(v = \text{black}\)
\[
\Rightarrow f[v] < f[u]
\]

Alternative algorithm for Topological Sort

Count the in degree of each vertex. Then repeat the following until there are no more vertices: Remove a vertex with in degree 0, remove all its outgoing edges, and update the in degrees of the neighboring vertices.

\[
\begin{aligned}
\text{FOR all vertices } v \text{ DO} \\
&\quad \text{degree}[v] = 0 \\
\text{OD} \\
\text{FOR all edges } (u, v) \in E \text{ DO} \\
&\quad \text{degree}[v] = \text{degree}[v] + 1 \\
&\quad \text{IF } \text{degree}[v] = 0 \text{ THEN ENQUEUE}(Q, v) \\
\text{OD} \\
&i = 0 \\
\text{WHILE } Q \neq \emptyset \text{ DO} \\
&\quad \text{DEQUEUE}(Q, u) \\
&\quad \text{Topsort}(u) = i \\
&\quad i = i + 1 \\
&\quad \text{FOR all edges } (u, v) \in E \text{ DO} \\
&\quad \quad \text{degree}[v] = \text{degree}[v] + 1 \\
&\quad \quad \text{IF } \text{degree}[v] = 0 \text{ THEN ENQUEUE}(Q, v) \\
\text{OD} \\
\text{OD}
\end{aligned}
\]
Strongly Connected Components (SCC)

A strongly connected component of a directed graph $G = (V, E)$ is:

a maximal set of vertices $U \subseteq V$ such that for every pair of vertices $u$ and $v$ in $U$, we have both

- $u \rightarrow \cdots \rightarrow v$

and

- $v \rightarrow \cdots \rightarrow u$

That is, $u$ and $v$ are reachable from each other!

in other words . . .

- $u \mathcal{R} v$ if $u$ and $v$ lie on a common cycle.
- $\mathcal{R}$ is an equivalence relation $(r, s, t)$.
- strongly connected components are a partition of graph $G$ under $\mathcal{R}$.
**SCC: Pseudocode**

(CLRS §22.5)
To compute SCC of directed graph \( G = (V, E) \), use two DFS’s, one on \( G \) and one on \( G^T \) (\( G \), with edges swapped):

**STRONGLY-CONNECTED-COMPONENTS(G)**
1. call DFS\((G)\) to compute finishing times \( f[u] \)
   for each vertex \( u \)
2. compute \( G^T \)
3. call DFS\((G^T)\), but in the main loop of DFS,
   consider the vertices in order of
   decreasing \( f[u] \) (as computed in line 1)
4. output vertices of each tree in the depth-first
   forest of step 3 as a separate SCC

Intuition: explore latest-finished vertices first
Running time \( \Theta(V + E) \) [Why?]

- Strongly-Connected-Components can be
  found in linear time.

---

**SCC: Lemmas and Theorems**

Lemma 22.13
- Let \( C \) and \( C' \) be two strongly connected components
  in directed graph \( G \). Let \( u, v \in C \) and \( u', v' \in C' \).
  If there is a path in \( G \) from \( u \) to \( u' \), then there cannot
  be a path in \( G \) from \( v' \) to \( v \).

Lemma 22.14
- Let \( C \) and \( C' \) be two strongly connected components
  in directed graph \( G \). Suppose there is an edge \( (u, v) \)
  in \( G \), where \( u \in C \) and \( v \in C' \).
  Then \( f(C) > f(C') \).

Corollary 22.15
- Let \( C \) and \( C' \) be two strongly connected components
  in directed graph \( G \). Suppose there is an edge \( (u, v) \)
  in \( G^T \), where \( u \in C \) and \( v \in C' \).
  Then \( f(C) < f(C') \).
SCC: Lemmas and Theorems

Theorem 22.16

- **Strongly-Connected-Components(G)** correctly computes the strongly connected components of a directed graph $G$.

See CLRS §22.5 for proofs and further explanations.