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 (DG): \( E \) is a set of ordered pairs of vertices \((u, v)\) where \( u, v \in V \)
    \[ V = \{1, 2, 3, 4, 5, 6\} \]
    \[ E = \{(1, 2), (2, 3), (1, 4), (2, 5), (3, 4), (5, 6)\} \]

- Undirected graph: \( E \) is a set of unordered pairs of vertices \( \{u, v\} \) where \( u, v \in V \)

- Denote an edge to \( u \) and \( v \) as \( (u, v) \).
- Degree of vertex in undirected graph is the number of edges incident to it.
- In (out) degree of a vertex in directed graph is the number of edges entering (leaving) it.

A path from \( u_0 \) to \( u_1 \) is a sequence of vertices \( (u_0, u_1, u_2, \ldots, u_n) \) such that \((u_i, u_{i+1}) \in E\) for \( i = 0, 1, \ldots, n-1 \).

2 Graph traversal

- There are two standard (and simple) ways of traversing all vertices/edges in a graph in a systematic way:
  - Depth-first
  - Breadth-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.

- BFS guarantees to hold all gray vertices—vertices we have seen but we still not done with.
  - We remember from which vertex a given vertex \( v \) is colored gray (visit \( v \)).
### BFS

**Algorithm**

```plaintext
BFS(s)
    color[s] = gray
    d[s] = 0
    ENQUEUE(Q, s)
    WHILE Q not empty DO
        Dequeue(Q, u)
        FOR (v, u) ∈ E DO
            IF color[v] = white THEN
                color[v] = gray
                d[v] = d[u] + 1
                VISIT(v) = u
                ENQUEUE(Q, v)
        FI
        color[u] = black
    OD
```

- Algorithm runs in $O(|V|+|E|)$ time
- Note:
  - The edges (visit[v], u), for all $v \in V$ from 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 $v$ to $s$.
- If graph is not connected we have to try to start the traversal at all nodes.

### DFS

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

```plaintext
DFS(v)
    color[v] = gray
    d[v] = time
    time = time + 1
    FOR (w, v) ∈ E DO
        IF color[w] = white THEN
            VISIT(w) = v
            DFS(w)
        FI
    OD
    color[v] = black
    f[v] = time
    time = time + 1
```

- 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.

### 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 unexplored adjacent vertex.
  - Explore as deeply as possible.
- Often we are interested in “discovery time” and “finishing time” of vertex $v$
  - Discovery time ($d[v]$): indicates at what “time” vertex $v$ is first visited.
  - Finishing time ($f[v]$): indicates at what “time” all adjacent vertices of vertex $v$ have been visited.
- Instead of using a stack in a DFS algorithm, 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 $f[u]$
**DFS: Running time**

Running time $O(|V|^2)$, because
DFS called once per vertex
Each loop over $\text{Adj}$ 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**

- white
- d = gray
- d f = black

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
  
  \[ d[u] < d[v] < f[v] < f[u]. \]

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)

(b)

(c)

**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)*
  - nontree, 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:

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.

DFS: Lemma

Theorem 22.10:

Proof:

- 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 \( \implies \) no back edge: trivial (back edge \( \implies \) cycle)
- Proof that no back edges \( \implies \) acyclic: No back edges \( \implies \) only tree edges (by above lemma) \( \implies \) forest \( \implies \) 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:

  ![Diagram of directed acyclic graph]

- 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.
\( \implies \): back edge \( \implies \) cycle
\( \Leftarrow \): Contrapositive: cycle \( \implies \) back edge

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

\[
\begin{array}{c}
u \\ \rightarrow v \\ \ldots \\
\end{array}
\]

\( (v \text{ is first vertex visited}) \)

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. 

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

Topological Sort

**Topological Sort** of a dag $G = (V, E)$ is a

- Linear ordering of all vertices of a dag

such that

- If $G$ contains an edge $(u, v)$, then $u$ appears before $v$ in the 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 TOPOLOGICAL-SORT(G)

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

When \((u, v)\) explored, \(u\) is gray
If \(v = \text{gray}\)
\(\Rightarrow (u, v) = \text{backedge (cycle, contradiction)}\).

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

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

Alternative algorithm for Topological Sort

Given 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.

```
FOR all vertices \(v\) DO
    degree\([v]\) = 0
END
FOR all edges \((u, v)\) \(\in E\) DO
    degree\([v]\) = degree\([v]\) + 1
    IF degree\([v]\) = 0 THEN Enqueue(Q, e)
END
i = 0
WHILE Q \(\neq\) \emptyset DO
    Enqueue(Q, e)
    Dequeue(Q, u)
    i = i + 1
    FOR all edges \((v, e)\) \(\in E\) DO
        degree\([v]\) = degree\([v]\) - 1
        IF degree\([v]\) = 0 THEN Enqueue(Q, e)
    END
END
```

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 ...

- \(u \sim R v\) if \(u\) and \(v\) lie on a common cycle.
- \(R\) is an equivalence relation \((r, s, t)\).
- strongly connected components are a partition of graph \(G\) under \(R\).

SCC: examples

![Graph diagrams illustrating strongly connected components](image-url)
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.