1 Basic Definitions

- Graphs are useful models for reasoning about relations among objects and combinatorial problems. Many real-life problems can be solved by converting them to graphs. Proper application of graph theory ideas can drastically reduce the solution time for some important problems.
- A graph has a set vertices $V$, often labeled $v_1, v_2, \ldots$, and a set of edges $E$, labeled $e_1, e_2, \ldots$.
- Each edge $(u, v)$ “joins” two nodes $u$ and $v$.
- We write $G = (V, E)$ for the graph with vertex set $V$ and edge set $E$.
- In applications, where pair $(u, v)$ is distinct from pair $(v, u)$, the graph is directed. Otherwise, the graph is undirected. We can convert an undirected graph to a directed one by duplicating edges, and orienting them both ways.
- When $(u, v)$ is an edge, we say $v$ is adjacent to (or, neighbor of) $u$. A loop is an edge with both endpoints being the same.
- In undirected graphs, the degree of a node equals its number of neighbors. In directed graphs, we have the out-degree and the in-degree.
- In some applications, the edges can be associated with weights or costs.

2 Examples of Graphs

- Transportation Networks. The map of routes served by an airline carrier forms a graph, whose nodes are the airports, and we have an edge $(u, v)$ whenever airline has a non-stop flight from $u$ to $v$. Typically, airline edges are undirected—flight $(u, v)$ also means a flight $(v, u)$. 
Other transportation networks: rail networks, road networks.

- Communication Networks. Internet is essentially a collection of computers connected by communication links. Nodes are computers, and edges are physical links.
  Wireless networks: devices, and wireless connections.

- Information networks. WWW has web pages as nodes, and hyperlinks as edges.

- Social networks.

- Dependency graphs: nodes = courses, and edges = prereqs;

3 Representations of Graphs

- **Adjacency Matrix:** a 2-dim array $V \times V$. For each edge $(u, v)$, set $A[u, v] = 1$, or equal to cost, etc. Use infinity or 0 for non-edges.

  - Pros: easy to check if $(u, v)$ an edge in $G$.

  - Cons: Takes $V^2$ space even if graph has very few edges; e.g. street map, which typically has $O(V)$ edges. Infeasible space when $V$ is millions of nodes.

- **Adjacency List:** An array of (header cells for) adjacency lists. The $i$th cell points to a linked list of all vertices adjacent to vertex $v_i$.

  - Example:

    | 1 : | 2 4 3 |
    | 2 : | 4 5  |
    | 3 : | 6    |
    | 4 : | 6 7 3 |
    | 5 : | 4 7  |
    | 6 : |      |
    | 7 : | 6    |

  - Space is $O(E)$; each directed edge stored just once. Thus, if $G$ is undirected $(u, v)$ appears in lists of both $u$ and $v$.

  - Pros. Linear space. Easy to list out all vertices adjacent to $u$.

  - Cons: Checking if $(u, v)$ is an edge can take $O(n)$ time.
4 Paths and Connectivity

• One of the fundamental operations in graphs is that of traversing a sequence of nodes (and edges). Such a traversal could correspond to user browsing web pages by following hyper links, rumor passing by word of mouth, or travel route of an airline passenger, email passing through a chain of routers, etc.

• A path is sequence of vertices \( w_1, w_2, \ldots, w_k \) such that each pair \( (w_i, w_{i+1}) \) is an edge of \( G \). The length of a path is the number of edges in it, or total weight if each edge has a weight associated with it.

• A simple path has no repeated vertex, except first and last can be the same; in that case, the path is a cycle.

• An undirected graph is connected if there is a path between any two vertices. A directed graph with this property is strongly connected. A weakly connected graph—underlying graph connected but the directed graph may not have directed path between all pairs.

• Trees: an undirected graph is a tree if it is connected and does not contain a cycle.

• Trees are one of the simplest type of graphs. Any tree on \( n \) nodes has \( n - 1 \) edges, and therefore the deletion of a single node or edge disconnects it.

• Often, it is useful to root the tree at a particular node \( r \), and then orient all edges away from \( r \). EXAMPLE.

• In a rooted tree, each node (except root) has a parent, and if \( u \) is the parent of \( w \), then \( w \) is called a child of \( u \).

• More generally, \( w \) is called a descenedant of \( u \), and \( u \) an ancestor of \( w \), if \( u \) lies on the path from \( w \) to the root.

5 Graph Connectivity and Graph Traversals

• We start with one of the most basic questions regarding graphs. Given a graph \( G = (V, E) \), and two nodes \( s \) and \( t \), is there a path joining \( s \) and \( t \)?

• This is called the \( st \)-connectivity problem. (This is also the classical Maze problem.) In small graphs, one can decide this by visual inspection, but quickly becomes challenging in large graphs.

• More generally, given a start node \( s \), what are all the nodes reachable from \( s \)? This set is called the connected component of \( G \) containing \( s \).
• There are two simply algorithms for $st$-connectivity

5.1 Breadth First Search

• The simplest algorithm for $st$-connectivity is the following. We start at $s$, and explore outward in all possible directions.

• We just have to make sure we don’t get stuck in a loop, so we use markers to keep track of nodes we have already visited.

• Each node will get a layer number (also called level). Initially, we have only $s$, which is layer 0. The next iteration adds previously unreached nodes that have an edge to an already reached node. More specifically,

• We initialize Layer $L_0 = \{s\}$; i.e. layer 0 containing just $s$. Layer $L_1$ consists of all neighbors of $s$.

• Assuming we have layers $L_0, L_1, \ldots, L_i$, define

  $$L_{i+1} = \text{nodes not yet encountered who have an edge to some node in layer } L_i$$

• Example with 3 connected components.

• The layer by layer exploration of $G$ produces a tree-like structure, which is called the BFS tree of $G$.

• For each $j \geq 1$, layer $L_j$ consists of all nodes at distance exactly $j$ from $s$. There is a path from $s$ to $t$ if and only if $s$ appears in some layer of BSF from $s$.

• Let $T$ be a BFS tree, and let $x$ and $y$ be nodes in $T$ belonging to different layers $L_i$ and $L_j$. If $(x, y)$ is an edge of $G$, then $|i - j| \leq 1$.

• Proof. For contradiction, assume $i < j - 1$. When $x$ is scanned in layer $i$, the edge $(x, y)$ will add $y$ to layer $L_{i+1}$, ensuring $j \leq i + 1$.

• BFS can be constructed in $O(m + n)$ time, using Adj List representation of $G$.

• The set of nodes discovered by the BFS is precisely those reachable from $s$. We refer to this set $R$ as the connected component of $G$ containing $s$.

• Once we have $R$, we can simply check if $t$ belongs to $R$, and if so we have $st$ connectivity.

• BFS however is only one way to discover $R$. Another, and a very different, method is depth first search.
6 Depth First Search

- Depth-First-Search (DFS) uses a method similar to the exploration of mazes:
- Starting at \( s \), we take the first edge out of \( s \), and continue recursively until we reach a dead end— a node for which all neighbors have already been explored.
- We then backtrack until we get to a node with at least one explored neighbor.
- This is called DFS search, because it explores \( G \) by going as deeply as possible and only retreating when necessary.

\[
\text{DFS}(u) \\
\quad \text{Mark } u \text{ as Explored and add } u \text{ to } R \\
\quad \text{For each edge } (u, v) \text{ incident to } u \\
\quad \quad \text{if } v \text{ is not marked Explored, then} \\
\quad \quad \quad \text{recursively call } \text{DFS}(v) \\
\quad \quad \text{endif} \\
\quad \text{endfor.}
\]

- We can also implement DFS non-recursively.

Stack Implementation of DFS:

\[
\text{DFS}(s) \\
\quad \text{Init } S \text{ to be a stack with one item } s \\
\quad \text{While } S \text{ not empty} \\
\quad \quad \text{Take a node } u \text{ from } S \\
\quad \quad \quad \text{If } \text{Explored}[u] = \text{False} \text{ then} \\
\quad \quad \quad \quad \text{Set } \text{Explored}[u] = \text{True} \\
\quad \quad \quad \quad \text{For each edge } (u,v) \text{ incident to } u \\
\quad \quad \quad \quad \quad \text{Add } v \text{ to stack } S \\
\quad \quad \quad \text{endfor} \\
\quad \quad \text{endif} \\
\quad \text{endwhile}
\]
• Example from Kleinberg-Tardos.

• DFS also runs in $O(m + n)$ time, where $n = |V|$ and $m = |E|$.

• Although the DFS tree looks very different from the BFS tree of $G$, we can make strong claims about how non-tree edges connect the nodes of DFS.

• **Fact 1.** For a recursive call $DFS(u)$, all nodes that are marked *explored* between the invocation and the end of the recursive call are descendants of $u$ in $T$.

• **Fact 2.** Let $T$ be a DFS tree, let $x, y$ be two nodes in $T$ that have an edge between them in $G$, but $(x, y)$ is not an edge of $T$. Then, one of $x$ or $y$ is an ancestor of the other.

Suppose not, and assume that $x$ is reached first in DFS. When the edge $(x, y)$ is examined during the execution of $DFS(x)$, it is not added to $T$ because $y$ is marked Explored. Since $y$ was not marked Explored when $DFS(x)$ was first invoked, it must have been discovered during the recursive call. Thus, by Fact 1, $y$ must be a descendant of $x$.

• **Connected Components Fact.** For any two nodes $s$ and $t$ in $G$, their connected components are either identical or disjoint.

### 7 Applications of BFS and DFS

• **Testing Bipartititeness.** A graph $G$ is bipartite if its vertex set $V$ can be partitioned into sets $X$ and $Y$ in such a way that every edge of $G$ has one end in $X$ and the other in $Y$.

• Often we use colors red and blue (or 0 and 1) to represent $X$ and $Y$.

• A triangle is not bipartite: any partition will contain two nodes on the same side with an edge between them. The same argument also holds if $G$ is an odd-length cycle.

• Turns out however that odd cycles are the only obstacle for $G$ to be bipartite: that is, $G$ is bipartite if and only if it does not contain an odd cycle.

• In fact, one can use BFS to decide whether $G$ is bipartite, and in the end either discover the sets $X$ and $Y$, or detect an odd-cycle, thereby showing that $G$ is not bipartite.

• We can easily assume that $G$ is connected. Otherwise, we can apply the algorithm to each connected component separately.
• The algorithm begins by picking any arbitrary vertex \( s \), and color it 0.
• Now all neighbors of \( s \) must be colored 1, and these are precisely the nodes of layer 1.
• We alternate between colors: the nodes at layer \( i \) are colored 0 if \( i \) is even, and colored 1 if \( i \) is odd.
• At the end of the algorithm, we simply go back and check if the endpoints of each edge of \( G \) are colored differently. If not, that edge \((x, y)\) together with the path in the BFS from \( x \) to \( y \) is an odd cycle.
• Therefore, bipartiteness of a graph \( G \) can be decided in \( O(m + n) \) time.

8 Bi-Connectivity

• An undirected graph \( G \) is bi-connected if the deletion of a single node keeps it connected. That is, one must delete at least two nodes (and their incident edges) to disconnect \( G \).
• Another classical application of DFS is a linear-time algorithm (due to Hopcroft and Tarjan) to find bi-connected components of \( G \).
• Articulation point is a node \( v \) whose removal disconnects \( G \). Thus, \( G \) is bi-connected if and only if there is no articulation point.
• The main idea is to run a DFS while maintaining the following information for each vertex \( v \) of the DFS tree \( T \):
  1. the depth of \( v \) (once it gets visited), and
  2. the lowest depth among the neighbors of all descendants of \( v \), called the lowpoint
• More specifically, let \( d(v) \) be the depth (DFS number) of node \( v \). Define

\[
\text{low}(v) = \min\{d(v), \{d(w) : (u, w) \text{ is a back edge for some descendant } u \text{ of } v\}\}
\]

• The low() values of all the nodes can be computed in linear time, by performing a post-order traversal of \( T \).
• Example.
• One we have these computed, detecting articulation points is easy: the root is an articulation point, if it has more than one child; any non-root node \( v \) is an articulation point if it has a child \( w \) with \( \text{low}(w) \geq d(v) \).
• For proof, notice that if $v$ is an articulation point then none of the nodes explored during the recursive call at $v$ have an edge that goes to the other component, and thus the $\text{low}()$ value for all these points is $\geq d(v)$.

9 Topological Sort

• Suppose you have a set of tasks, which are subject to a set of precedence constraints: some jobs cannot be done before others. How shall you schedule the jobs without violating any prec constraint?

• Model as a directed graph where jobs are nodes and precedence relations are edges.

• Clearly, if there is a cycle in the graph, no feasible schedule.

• When there is no cycle, topological sorting is an ordering of vertices such if there is a path from $v_i$ to $v_j$, then $v_i$ appears before $v_j$ in the schedule.

Algorithm:
Find a vertex $v$ with zero in-degree (must exist!)
Print $v$, delete $v$, and its outgoing edges;
Repeat.

Improved Topological Sort

Compute all vertices’ indegs
Enqueue all those with zero indeg
Pick a vertex $w$ from the queue;
put $w$ next in schedule
for each vertex $v$ adj to $w$
    decrement $v$’s indeg
    add $v$ to queue if its indeg = 0

• This code only looks at each edge once, so $O(E)$ time.

• Example.

• One can use DFS to also perform topological sorting. How?
10 Strong Bi-Connectivity

- DFS and BFS algorithms work on directed graphs, without any significant change: while visiting a vertex \( v \), we just scan \( v \)'s out neighbors.

- In directed graphs, however, we need a stronger definition of a connected components. We put two vertices \( u \) and \( v \) in the same component only if we have a directed path from \( u \) to \( v \) and a path from \( v \) to \( u \).

- Example.

- We can also find strong connected components of \( G \) also in \( O(|V| + |E|) \) time, by using DFS, but in a more careful way.

- Historically, the first linear time algorithm dates back to 70s by Hopcroft and Tarjan.

- A simpler algorithm is by Koraraju-Sharir. It performs two DFS once on \( G \), and once on \( G^R \), which is \( G \) with all edges reversed.

- Intuition. Perform DFS on \( G \), and list the vertices in the **post-order**.

- Figure 1 shows a directed graph, and its DFS.

![Figure 1: A directed graph and its DFS.](image)

- The post-order numbering of nodes is: \( G, H, J, I, B, F, C, A, D, E \).

- We now perform a DFS on \( G^R \), always starting new DFS at the highest numbered vertex. So, in the example, first DFS starts at node \( G \), numbered 10. This leads nowhere, so \( G \) is a singleton node component.

- See Figure 2.

- Next DFS starts at \( H \), and this call adds \( I \) and \( J \) to the component of \( H \).
• Next starts at $B$, and adds $\{A, C, F\}$ before finishing.

• DFS at $D$ ends with singleton, as does for $E$.

![Graph](image)

Figure 2: $G^R$, with post-order numbering from the first DFS.

• Proof of Correctness. Key idea is that if $u, v$ are in the same SCC, then there are paths from $u$ to $v$, and from $v$ to $u$, in both $G$ and $G^R$.

• Thus, if two nodes are not in the same DFS tree, then they cannot be in one SCC.

• We show that if $x$ is the root of the DFS tree in $G^R$ containing $v$, then there is a path from $x$ to $v$, and from $v$ to $x$. Applying the same logic to $w$ gives a pair of paths between $x$ and $w$, and thus shows that $x, v, w$ are in the same SCC.

• Since $v$ is a descendant of $x$ in $G^R$ DFS, there is path from $x$ to $v$ in $G^R$, and thus a path from $v$ to $x$ in $G$.

• Since $x$ is the root, it has the higher post-order than $v$. Therefore, during the DFS in $G$, the recursive call at $v$ finished before the recursive call at $x$ finished. Since a path from $v$ to $x$ exists, it must be that $v$ is a descendant of $x$ in the DFS of $G$—otherwise, $v$ would finish after $x$. Therefore, there is a path from $x$ to $v$, and the proof is complete.