Chapter 3
Graphs
3.1 Basic Definitions and Applications
Undirected Graphs

Undirected graph. \( G = (V, E) \)
- \( V = \) nodes.
- \( E = \) edges between pairs of nodes.
- Captures pairwise relationship between objects.
- Graph size parameters: \( n = |V|, m = |E| \).

\[ V = \{ 1, 2, 3, 4, 5, 6, 7, 8 \} \]
\[ E = \{ (1,2), (1,3), (2,3), (2,4), (2,5), \]
\[ (3,5), (3,7), (3,8), (4,5), (5,6), (7,8) \} \]
\[ n = 8 \]
\[ m = 11 \]
Some Graph Applications

<table>
<thead>
<tr>
<th>Graph</th>
<th>Nodes</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>transportation</td>
<td>street intersections</td>
<td>highways</td>
</tr>
<tr>
<td>communication</td>
<td>computers</td>
<td>fiber optic cables</td>
</tr>
<tr>
<td>World Wide Web</td>
<td>web pages</td>
<td>hyperlinks</td>
</tr>
<tr>
<td>social</td>
<td>people</td>
<td>relationships</td>
</tr>
<tr>
<td>food web</td>
<td>species</td>
<td>predator-prey</td>
</tr>
<tr>
<td>software systems</td>
<td>functions</td>
<td>function calls</td>
</tr>
<tr>
<td>scheduling</td>
<td>tasks</td>
<td>precedence constraints</td>
</tr>
<tr>
<td>circuits</td>
<td>gates</td>
<td>wires</td>
</tr>
</tbody>
</table>
World Wide Web

Web graph.
- Node: web page.
- Edge: hyperlink from one page to another.
Ecological Food Web

Food web graph.

- Node = species.
- Edge = from prey to predator.

Graph Representation: Adjacency Matrix

Adjacency matrix. n-by-n matrix with $A_{uv} = 1$ iff (u, v) is an edge.
- Two representations of each edge.
- Space proportional to $n^2$.
- Checking if (u, v) is an edge takes $\Theta(1)$ time. (Checking k pairs (u,v) will cost $\Theta(k)$ time.)
- Identifying all edges takes $\Theta(n^2)$ time.

\[ n = \text{number of vertices} \]
Graph Representation: Adjacency List

**Adjacency list.** Node indexed array of lists.

- Two representations of each edge.
- Space proportional to $m + n$.
- Checking if $(u, v)$ is an edge takes $O(\text{deg}(u))$ time.
  (Checking $k$ pairs $(u,v)$ may cost up to $\Theta(kn)$ time.)
- Identifying all edges takes $\Theta(m + n)$ time.

$n = \text{number of vertices}$
Paths and Connectivity

Def. A path in an undirected graph $G = (V, E)$ is a sequence $P$ of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$ with the property that each consecutive pair $v_i, v_{i+1}$ is joined by an edge in $E$.

Def. A path is simple if all nodes are distinct.

Def. An undirected graph is connected if for every pair of nodes $u$ and $v$, there is a path between $u$ and $v$. 

![Graph Diagram]
**Cycles**

Def. A *cycle* is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first $k-1$ nodes are all distinct.

-cycle $C = 1-2-4-5-3-1$
Trees

Def. An undirected graph is a **tree** if it is connected and does not contain a cycle.

Theorem. Let $G$ be an undirected graph on $n$ nodes. Any two of the following statements imply the third.

- $G$ is connected.
- $G$ does not contain a cycle.
- $G$ has $n-1$ edges.
Rooted Trees

**Rooted tree.** Given a tree $T$, choose a root node $r$ and orient each edge away from $r$.

**Importance.** Models hierarchical structure.
3.2 Graph Traversal
Connectivity

\textbf{s-t connectivity problem.} Given two nodes \(s\) and \(t\), is there a path between \(s\) and \(t\)?

\textbf{s-t shortest path problem.} Given two node \(s\) and \(t\), what is the length (number of edges) of the shortest path between \(s\) and \(t\)?

\textbf{Applications.}
- Facebook.
- Maze traversal.
- Erdos number.
- Fewest number of hops in a communication network.
Breadth First Search

**BFS intuition.** Explore outward from \( s \) in all possible directions, adding nodes one "layer" at a time.

**BFS algorithm.**
- \( L_0 = \{ s \} \).
- \( L_1 \) = all neighbours of \( L_0 \).
- \( L_2 \) = all nodes that do not belong to \( L_0 \) or \( L_1 \), and that have an edge to a node in \( L_1 \).
- \( L_{i+1} \) = all nodes that do not belong to an earlier layer, and that have an edge to a node in \( L_i \).

**Theorem.** For each \( i \), \( L_i \) consists of all nodes at distance exactly \( i \) from \( s \). There is a path from \( s \) to \( t \) iff \( t \) appears in some layer.
Breadth First Search

BFS(s):

Set Discovered[s] = true and Discovered[v] = false for all other v
Initialize L[0] to consist of the single element s
Set the layer counter i = 0
Set the current BFS tree T = ∅

While L[i] is not empty
    Initialize an empty list L[i+1]
    For each node u ∈ L[i]
        Consider each edge (u, v) incident to u
        If Discovered[v] = false then
            Set Discovered[v] = true
            Add edge (u, v) to the tree T
            Add v to the list L[i+1]
        Endif
    Endfor
    Increment the layer counter i by one
Endwhile
Property. Let $T$ be a BFS tree of $G = (V, E)$, and let $(x, y)$ be an edge of $G$. Then the level of $x$ and $y$ differ by at most 1.
Breadth First Search: Analysis

**Theorem.** The above implementation of BFS runs in $O(m + n)$ time if the graph is given by its adjacency list representation.

**Pf.**
- Easy to prove $O(n^2)$ running time:
  - at most $n$ lists $L[i]$
  - each node occurs on at most one list
  - when we consider node $u$, there are $\leq n$ incident edges $(u, v)$, and we spend $O(1)$ processing each edge

- Actually runs in $O(m + n)$ time:
  - when we consider node $u$, there are $\text{deg}(u)$ incident edges $(u, v)$
  - total time processing edges is $\sum_{u \in V} \text{deg}(u) = 2m$

  Each edge $(u, v)$ is counted exactly twice in sum: once in $\text{deg}(u)$ and once in $\text{deg}(v)$
**Connected Component**

*Connected component.* Find all nodes reachable from $s$.

![Graph Image]

*Connected component containing node 1* = \{1, 2, 3, 4, 5, 6, 7, 8\}. 
**Connected Component**

**Connected component.** Find all nodes reachable from \( s \).

---

\[ R \text{ will consist of nodes to which } s \text{ has a path} \]

Initially \( R = \{s\} \)

While there is an edge \((u, v)\) where \( u \in R \) and \( v \notin R \)

Add \( v \) to \( R \)

Endwhile

---

**Theorem.** Upon termination, \( R \) is the connected component containing \( s \).

- BFS = explore in order of distance from \( s \).
- DFS = explore in a different way.
Chapter 3
Graphs

CLRS 12-13

Slides by Kevin Wayne.
Copyright © 2005 Pearson-Addison Wesley.
All rights reserved.
3.4 Testing Bipartiteness
Bipartite Graphs

**Def.** An undirected graph $G = (V, E)$ is bipartite if the nodes can be coloured red or blue such that every edge has one red and one blue end.

**Applications.**
- Stable marriage: men = red, women = blue.
- Scheduling: machines = red, jobs = blue.

![A diagram of a bipartite graph](attachment:image.png)
Testing bipartiteness. Given a graph $G$, is it bipartite?

- Many graph problems become:
  - easier if the underlying graph is bipartite (matching)
  - tractable if the underlying graph is bipartite (independent set)

- Before attempting to design an algorithm, we need to understand the structure of bipartite graphs.

**Diagram:**

- A bipartite graph $G$
- Another drawing of $G$
**Lemma.** If a graph $G$ is bipartite, it cannot contain an odd length cycle.

**Pf.** Not possible to 2-colour the odd cycle, let alone $G$. 

![Bipartite Graph](image1.png)  
**bipartite**  
(2-colorable)

![Not Bipartite Graph](image2.png)  
**not bipartite**  
(not 2-colorable)
Lemma. Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.

(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

Case (i)

Case (ii)
Bipartite Graphs

Lemma. Let $G$ be a connected graph, and let $L_0, ..., L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.
(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

Pf. (i)

- Suppose no edge joins two nodes in the same layer.
- By above property, this implies all edges join nodes on adjacent layers.
- Bipartition: red = nodes on odd levels, blue = nodes on even levels.
Lemma. Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.

(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

Pf. (ii)

- Suppose $(x, y)$ is an edge with $x, y$ in same level $L_j$.
- Let $z = \text{lca}(x, y) = \text{lowest common ancestor}^*.$
- Let $L_i$ be level containing $z$.
- Consider cycle that takes edge from $x$ to $y$, then path* from $y$ to $z$, then path* from $z$ to $x$.
- Its length is $1 + (j-i) + (j-i)$, which is odd.

*Consider only edges of the BFS tree.*
Corollary. A graph $G$ is bipartite iff it contains no odd length cycle.
3.5 Connectivity in Directed Graphs
Directed Graphs

Directed graph. $G = (V, E)$
- Edge $(u, v)$ goes from node $u$ to node $v$.

**Ex.** Web graph - hyperlink points from one web page to another.
- Directedness of graph is crucial.
- Modern web search engines exploit hyperlink structure to rank web pages by importance.
Graph Search

Directed reachability. Given a node \( s \), find all nodes reachable from \( s \).

Directed \( s \)-\( t \) shortest path problem. Given two node \( s \) and \( t \), what is the length of the shortest path between \( s \) and \( t \)?

Graph search. BFS extends naturally to directed graphs.

Web crawler. Start from web page \( s \). Find all web pages linked from \( s \), either directly or indirectly.
Strong Connectivity

Def. Node u and v are mutually reachable if there is a path from u to v and also a path from v to u.

Def. A graph is strongly connected if every pair of nodes is mutually reachable.

Lemma. Let s be any node. G is strongly connected iff every node is reachable from s, and s is reachable from every node.

Pf. ⇒ Follows from definition.
Pf. ⇐ Path from u to v: concatenate u-s path with s-v path.
    Path from v to u: concatenate v-s path with s-u path. □
    ok if paths overlap
Theorem. Can determine if $G$ is strongly connected in $O(m + n)$ time.

Pf.

- Pick any node $s$.
- Run BFS from $s$ in $G$.
- Run BFS from $s$ in $G^{\text{rev}}$.
- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma.

![Strong Connectivity Diagram](image-url)
3.6 DAGs and Topological Ordering
3.6 DAGs and Topological Ordering

What is the connection between computer science and algorithms?

I study CS and I hear a lot if you want to be a good programmer you must be good at algorithm, why? and if it’s true what algorithm should I read or study?

Thomas Cormen, The C in CLRS.

Written Sep 12 · Upvoted by Siddarth Sampangi, UCSD B.S. in CS ’14; UMass Amherst M.S. in CS ’16, Bill Poucher, Baylor CS prof, ICPC Exec Director, Software: energy, synthetic genetics, etc., and Rohit RK

I’ll tell you a little story. A true story.
In the late 1970s and early 1980s, I worked at a startup that made systems for computer-aided design. Users could define parts and store them in a library of parts. Each part could include another part by reference, so that if you changed the definition of a part, then all of its uses would update automatically. Part A could include a reference to part B, which could include a reference to part C, and so on. Circular references were not allowed, as a part could not include itself.

We had a customer that wanted the library of parts written out to tape so that each part appeared on the tape before any other part that used it. I was the only person at the company who knew that what this customer wanted was a topological sort of a directed acyclic graph. I knew that there was an efficient algorithm for this problem, and I knew where I’d seen it (in Knuth). I didn’t remember the details of the algorithm, and so I went to the library, got a copy of Knuth, and implemented the algorithm.

People at the company thought I was a god for knowing how to solve the problem, and how to solve it efficiently.

That’s why you want to know about algorithms.
Directed Acyclic Graphs

**Def.** A DAG is a directed graph that contains no directed cycles.

**Ex.** Precedence constraints: edge \((v_i, v_j)\) means \(v_i\) must precede \(v_j\).

**Def.** A topological order of a directed graph \(G = (V, E)\) is an ordering of its nodes as \(v_1, v_2, ..., v_n\) so that for every edge \((v_i, v_j)\) we have \(i < j\).
Precedence Constraints

Precedence constraints. Edge \((v_i, v_j)\) means task \(v_i\) must occur before \(v_j\).

Applications.
- Course prerequisite graph: course \(v_i\) must be taken before \(v_j\).
- Compilation: module \(v_i\) must be compiled before \(v_j\).
- Pipeline of computing jobs: output of job \(v_i\) needed to determine input of job \(v_j\).
Lemma. If $G$ has a topological order, then $G$ is a DAG.

Pf. (by contradiction)

- Suppose that $G$ has a topological order $v_1, \ldots, v_n$ and that $G$ also has a directed cycle $C$. Let's see what happens.
- Let $v_i$ be the lowest-indexed node in $C$, and let $v_j$ be the node just before $v_i$ in $C$; thus $(v_j, v_i)$ is an edge.
- By our choice of $i$, we have $i < j$.
- On the other hand, since $(v_j, v_i)$ is an edge and $v_1, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction. ▪

![Diagram of directed acyclic graph with nodes $v_1, v_i, \ldots, v_j, v_n$ and edges indicating the directed cycle $C$. The supposed topological order is $v_1, \ldots, v_n$.](image-url)
Lemma. If $G$ has a topological order, then $G$ is a DAG.

Q. Does every DAG have a topological ordering?

Q. If so, how do we compute one?
Lemma. If $G$ is a DAG, then $G$ has a node with no incoming edges.

Pf. (by contradiction)

- Suppose that $G$ is a DAG and every node has at least one incoming edge. Let's see what happens.
- Pick any node $v$, and begin following edges backward from $v$. Since $v$ has at least one incoming edge $(u, v)$ we can walk backward to $u$.
- Then, since $u$ has at least one incoming edge $(x, u)$, we can walk backward to $x$.
- Repeat until we visit a node, say $w$, twice.
- Let $C$ denote the sequence of nodes encountered between successive visits to $w$. $C$ is a cycle.
Lemma. If $G$ is a DAG, then $G$ has a topological ordering.

Pf. (by induction on $n$)

- Base case: true if $n = 1$.
- Given DAG on $n > 1$ nodes, find a node $v$ with no incoming edges.
- $G - \{ v \}$ is a DAG, since deleting $v$ cannot create cycles.
- By inductive hypothesis, $G - \{ v \}$ has a topological ordering.
- Place $v$ first in topological ordering; then append nodes of $G - \{ v \}$ in topological order. This is valid since $v$ has no incoming edges.

To compute a topological ordering of $G$:
Find a node $v$ with no incoming edges and order it first
Delete $v$ from $G$
Recursively compute a topological ordering of $G - \{v\}$ and append this order after $v$
Topological Ordering Algorithm: Example

Topological order:
Topological Ordering Algorithm: Example

Topological order: \( v_1 \)
Topological Ordering Algorithm: Example

Topological order: $v_1, v_2$
Topological Ordering Algorithm: Example

Topological order: $v_1, v_2, v_3$
Topological Ordering Algorithm: Example

Topological order: \( v_1, v_2, v_3, v_4 \)
Topological Ordering Algorithm: Example

Topological order: $v_1, v_2, v_3, v_4, v_5$
Topological Ordering Algorithm: Example

Topological order: \( v_1, v_2, v_3, v_4, v_5, v_6 \)
Topological Ordering Algorithm: Example

Topological order: $v_1, v_2, v_3, v_4, v_5, v_6, v_7$. 

Diagram representation with nodes $v_1, v_2, v_3, v_4, v_5, v_6, v_7$ and directed edges showing the topological order.
Topological Sorting Algorithm: Running Time

To compute a topological ordering of $G$:
Find a node $v$ with no incoming edges and order it first
Delete $v$ from $G$
Recursively compute a topological ordering of $G - \{v\}$
and append this order after $v$

Theorem. Algorithm finds a topological order in $O(m + n)$ time.

Pf.
- Maintain the following information:
  - for each node $w$, $count[w] = \text{number of remaining incoming edges}$
  - $S = \text{set of remaining nodes with no incoming edges}$
- Initialization: $O(m + n)$ via single scan through graph.
- Update: to delete $v$
  - remove $v$ from $S$
  - decrement $count[w]$ for all edges from $v$ to $w$, and
    add $w$ to $S$ if $count[w]$ hits 0
  - this is $O(1)$ per edge  ■
Chapter 3
Graphs