3.1 Basic Definitions and Applications

Graphs

Graph: $G = (V, E)$
- $V = \text{nodes}$
- $E = \text{edges between pairs of nodes}$

- Undirected graph represents symmetric relation
- Directed graph represents general binary relation

Graph size parameters: $n = |V|$, $m = |E|$

Simple: no loops and no multiple edges

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>circuit</td>
<td>gates</td>
<td>wires</td>
</tr>
</tbody>
</table>

World Wide Web
- Node: web page
- Edge: hyperlink from one page to another

Ecological Food Web
- Node: species
- Edge: from prey to predator

Graph Representation: Adjacency Matrix

Adjacency matrix. n-by-n matrix with \( A_{uv} = 1 \) if (u, v) is an edge.
- Two 1s of each edge for undirected graph.
- Space proportional to \( n^2 \).
- Identifying all edges takes \( O(n^2) \) time.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Graph Representation: Adjacency List

Adjacency list. Node indexed array of lists.
- Two representations of each edge for undirected graphs.
- Space proportional to \( m + n \).
- Checking if (u, v) is an edge takes \( O(\text{deg}(u)) \) time.
- Identifying all edges takes \( O(m + n) \) time.

Paths and Connectivity

Def. A path in a 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 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 from \( u \) to \( v \).

Def. A directed graph is strongly connected if for every pair of nodes \( u \) and \( v \), there is a path from \( u \) to \( v \).

Cycles

Def. A cycle is a path \( v_1, v_2, \ldots, v_k-1, v_k \) in which \( v_1 = v_k \), \( k \geq 2 \), and the first \( k-1 \) nodes are all distinct.

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.
- \( G \) has \( n-1 \) edges.
- \( G \) contains a cycle.

Rooted Trees

Rooted tree. A directed graph where the underlying undirected graph is a tree and there is a node \( r \) called root, and each edge points away from \( r \).

Importance. Models hierarchical structure.
3.2 Graph Traversal

**Connectivity**
- **s-t connectivity problem.** Given two nodes s and t, is there a path between s and t?
- **s-t shortest path problem.** Given two nodes s and t, what is the length of the shortest path between s and t?

**Applications:**
- Maze traversal.
- 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 neighbors 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.

**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 representation.

**Pf.**
- Runs in \( O(m + n) \) time:
  - when we consider node \( u \), there are \( \deg(u) \) incident edges \( (u, v) \)
  - total time processing edges is \( \sum_{v \in V} \deg(u) = 2m \) — each edge \((u, v)\) is counted exactly twice in sum: once in \( \deg(u) \) and once in \( \deg(v) \)

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

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

Find all nodes reachable from s.

Theorem. Upon termination, R is the connected component containing s. BFS = explore in order of distance from s.

- R will consist of nodes to which s has a path.
- Initially R = {s}.
- While there is an edge (u, v) where u ∈ R and v ∉ R, add v to R.
- Endwhile.

Breadth-First Search Example

- Start search at vertex 1.
- Visit/mark/label start vertex and put in a FIFO queue.
- Remove 1 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 2 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 3 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 4 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 5 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 6 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 7 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 8 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 9 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 10 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 11 from Q; visit adjacent unvisited vertices; put in Q.
- FIFO Queue: 1

**FIFO Queue**

- Start search at vertex 1.
- Visit/mark/label start vertex and put in a FIFO queue.
- Remove 1 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 2 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 3 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 4 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 5 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 6 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 7 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 8 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 9 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 10 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 11 from Q; visit adjacent unvisited vertices; put in Q.
- FIFO Queue: 1

**FIFO Queue**

- Start search at vertex 1.
- Visit/mark/label start vertex and put in a FIFO queue.
- Remove 1 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 2 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 3 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 4 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 5 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 6 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 7 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 8 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 9 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 10 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 11 from Q; visit adjacent unvisited vertices; put in Q.
- FIFO Queue: 1

**FIFO Queue**

- Start search at vertex 1.
- Visit/mark/label start vertex and put in a FIFO queue.
- Remove 1 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 2 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 3 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 4 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 5 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 6 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 7 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 8 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 9 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 10 from Q; visit adjacent unvisited vertices; put in Q.
- Remove 11 from Q; visit adjacent unvisited vertices; put in Q.
- FIFO Queue: 1
Breadth-First Search Example

Remove 2 from Q; visit adjacent unvisited vertices; put in Q.

FIFO Queue
4 5 3 6

Breadth-First Search Example

Remove 4 from Q; visit adjacent unvisited vertices; put in Q.

FIFO Queue
4 5 3 6

Breadth-First Search Example

Remove 4 from Q; visit adjacent unvisited vertices; put in Q.

FIFO Queue
5 3 6

Breadth-First Search Example

Remove 5 from Q; visit adjacent unvisited vertices; put in Q.

FIFO Queue
5 3 6

Breadth-First Search Example

Remove 5 from Q; visit adjacent unvisited vertices; put in Q.

FIFO Queue
3 6 9 7

Breadth-First Search Example

Remove 3 from Q; visit adjacent unvisited vertices; put in Q.
Breadth-First Search Example

Remove 3 from Q; visit adjacent unvisited vertices; put in Q.

Remove 6 from Q; visit adjacent unvisited vertices; put in Q.

Remove 6 from Q; visit adjacent unvisited vertices; put in Q.

Remove 9 from Q; visit adjacent unvisited vertices; put in Q.

Remove 9 from Q; visit adjacent unvisited vertices; put in Q.

Remove 7 from Q; visit adjacent unvisited vertices; put in Q.
Breadth-First Search Example

Remove 7 from Q; visit adjacent unvisited vertices; put in Q.

FIFO Queue

Remove 8 from Q; visit adjacent unvisited vertices; put in Q.

FIFO Queue

Queue is empty. Search terminates.

Time Complexity

Each visited vertex is put on (and as removed from) the queue exactly once. When a vertex is removed from the queue, we examine its adjacent vertices.

- O(n) if adjacency matrix used
- O(deg) if adjacency lists used
- O(k), where k is number of vertices in the component that is searched (adjacency matrix)
- O(deg), where m is the number of edges in the component (adjacency list)

Connected Components

Start a breadth-first search at any as yet unvisited vertex of the graph. Newly visited vertices (plus edges between them) define a component. Repeat until all vertices are visited.

- O(n^2) when adjacency matrix used
- O(n+m) when adjacency lists used (m is number of edges)

Spanning Tree

A subgraph of a connected graph that contains every vertex and is a tree.

Breadth-first search from vertex 1.
Breadth-first spanning tree.
Spanning Tree

- Start a breadth-first search at any vertex of the graph.
- If the graph is connected, the n-1 edges used to get to unvisited vertices define a spanning tree (breadth-first spanning tree).
- All visited vertices have the shortest steps to the start vertex in this spanning tree.
- Time:
  - $O(n^2)$ when adjacency matrix used
  - $O(n+m)$ when adjacency lists used ($m$ is number of edges)

Depth-First Search for Undirected Graph

```java
depthFirstSearch(v) {
    Mark vertex v as reached.
    for (each vertex u adjacent from v)
        if (unreached(u))
            depthFirstSearch(u);  // (v, u) is a tree edge.
            // else either (u, v) is a true edge or (v, u) is a back edge.
    }
```

Depth-First Search Example

1. Start search at vertex 1.
2. Label vertex 1 and do a depth first search from either 2 or 4.
   Suppose that vertex 2 is selected.
3. Label vertex 2 and do a depth first search from either 3, 5, or 6.
   Suppose that vertex 5 is selected.
4. Label vertex 5 and do a depth first search from either 3, 7, or 9.
   Suppose that vertex 9 is selected.
5. Label vertex 9 and do a depth first search from either 6 or 8.
   Suppose that vertex 8 is selected.
Label vertex 8 and return to vertex 9. From vertex 9 do dfs(6).

Label vertex 6 and do a depth first search from either 4 or 7. Suppose that vertex 4 is selected.

Label vertex 4 and return to 6. From vertex 6 do dfs(7).

Label vertex 7 and return to 6. Return to 9.

Return to 5.

Do a dfs(3).
Depth-First Search Example

Label 3 and return to 5.
Return to 2.

Return to 1.

Return to the invoking method.

Depth-First Search Properties

Same complexity as BFS.
Some properties with respect to path finding, connected components, and spanning trees.
Edges used to reach unlabeled vertices define a depth-first spanning tree when the graph is connected.
There are problems for which BFS is better than DFS and vice versa.

All edges are classified as either "tree edges" or "back edges".
Theorem: For any back edge \((v, u)\), \((u, v)\) lies in a cycle.

3.4 Testing Bipartiteness

Bipartite Graphs

Def. An undirected graph \(G = (V, E)\) is bipartite if the nodes can be colored 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 bipartite graph
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 structure of bipartite graphs.

A bipartite graph G

Another drawing of G

An Obstruction to Bipartiteness

Lemma. If a graph G is bipartite, it cannot contain an odd length cycle.

Proof. Not possible to 2-color the odd cycle, let alone G.

Obstruction to Bipartiteness

Corollary. A graph G is bipartite iff it contain no odd length cycle.

Testing Bipartiteness using Depth-First Search

```
depthFirstSearch(v) {
  Mark vertex v as reached.
  for (each vertex u adjacent from v)
    if (unreachable(u)) depthFirstSearch(u); // (v, u) is a tree edge.
    // else either (u, v) is a tree edge or (v, u) is a back edge.
}
```

```
boolean bipartiteDFS(v, color) {
  Mark vertex v as color
  color = oppositeColor(color)       // white = oppositeColor(black), …
  for (each vertex u adjacent from v)
    if (unmarked(u))                     // (v, u) is a tree edge.
      if ( ! bipartiteDFS(u, color)) return false;
    else if (color(v) == color(u))    // (v, u) is a back edge.
      return false
  return true
}
```

3.5 Connectivity in Directed Graphs

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.

Graph Search
Strong Connectivity

Def. Nodes 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. 

Strong Connectivity: Algorithm

Theorem. Can determine if G is strongly connected in O(m + n) time.

Pf. Pick any node s. 
- Run DFS from s in G. reverse orientation of every edge in G
- Run DFS from s in G'.
- Return true iff all nodes reached in both DFS executions.
- Correctness follows immediately from previous lemma.

Strongly Connected Components

Any directed graph can be partitioned into a unique set of strong components.

The algorithm for finding the strong components of a directed graph G uses the transpose of the graph.

1. Execute the depth-first search dfs() for the graph G which creates the list dfsList consisting of the vertices in G in the reverse order of their finishing times.
2. Generate the transpose graph GT.
3. Using the order of vertices in dfsList, make repeated calls to dfs() for vertices in GT. The list returned by each call is a strongly connected component of G.

Strongly Connected Components (example)

dfsList: [A, B, C, E, D, G, F]

Using the order of vertices in dfsList, make successive calls to dfs() for graph GT

Vertex A: dfs(A) returns the list [A, C, B] of vertices reachable from A in GT.
Vertex E: The next unvisited vertex in dfsList is E. Calling dfs(E) returns the list [E].
Vertex D: The next unvisited vertex in dfsList is D. dfs(D) returns the list [D, F, G] whose elements form the last strongly connected component.
3.6 DAGs and Topological Ordering

Directed Acyclic Graphs

Def. An 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 \(v_1, v_2, \ldots, v_n\) so that for every edge \((v_i, v_j)\) we have \(i < j\).

Applications:
- A study plan with course prerequisite graph.
- A schedule of modules for compilation.

Topological Order: A listing of all nodes \(v_1, v_2, \ldots, v_n\), such that all edge \((v_i, v_j)\) has the property that \(i < j\).

Applications:
- A study plan with course prerequisite graph.
- A schedule of modules for compilation.

Precedence Constraints

Precedence constraints: Edge \((v_i, v_j)\) means \(v_i\) 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 \(v_j\).

Applications:
- A study plan with course prerequisite graph.
- A schedule of modules for compilation.

Directed Acyclic Graphs

Lemma. If \(G\) has a topological order, then \(G\) is a DAG.

Proof. (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 \(v_i\); 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.

Directed Acyclic Graphs

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?
Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a node with no incoming edges.

Pf. (by contradiction)
1. Suppose that G is a DAG and every node has at least one incoming edge. Let’s see what happens.
2. 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.
3. Then, since u has at least one incoming edge (x, u), we can walk backward to x.
4. Repeat until we visit a node, say w, twice.
5. Let C denote the sequence of nodes encountered between successive visits to w. C is a cycle. □

Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a topological ordering.

Pf. (by induction on n)
1. Base case: true if n = 1.
2. Given DAG on n > 1 nodes, find a node v with no incoming edges.
3. G - {v} is a DAG, since deleting v cannot create cycles.
4. By inductive hypothesis, G - {v} has a topological ordering.
5. Place v first in topological ordering, then append nodes of G - {v} in topological order. This is valid since v has no incoming edges. □
Graph Algorithms: Topological Sort

The topological sorting algorithm:
As each vertex is removed, update the predecessor counts, and for any vertex whose count has become zero, put it in the queue.

Graph Algorithms: Topological Sort

The topological sorting algorithm:
As each vertex is removed, update the predecessor counts, and for any vertex whose count has become zero, put it in the queue.

Graph Algorithms: Topological Sort

The topological sorting algorithm:
As each vertex is removed, update the predecessor counts, and for any vertex whose count has become zero, put it in the queue.

Topological Sorting Algorithm: Running Time

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

Proof. Maintain the following information:
- count[w] = remaining number of incoming edges
- S = 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