Summary of the course so far:

Algorithmic Paradigms
- reductions
- divide and conquer
- greedy algorithms
- dynamic programming

Next:  Graph Algorithms
Graph $G = (V, E)$
- $V$ = vertices (or nodes) = \{1, 2, \ldots, n\}
- $E$ = edges $\subseteq V \times V$ $|E| = m$

We assume:
- no edge $(v,v)$ (no loops)
- $E$ is a set (no multiple edges)

Edges may be undirected (unordered pairs) or directed (ordered pairs)

<table>
<thead>
<tr>
<th>Undirected</th>
<th>Directed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V = {1, 2, 3, 4, 5}$</td>
<td>$V = {1, 2, 3}$</td>
</tr>
<tr>
<td>$E = {(1,2), (1,3), (1,5), \ldots}$</td>
<td>$E = {(1,2), (2,3), (1,3), (3,1)}$</td>
</tr>
</tbody>
</table>

How many edges can a graph have?

- Undirected: $0 \leq m \leq \binom{n}{2} = \frac{n(n-1)}{2}$
- Directed: $0 \leq m \leq n(n-1)$

So, $m \in O(n^2)$ and can be $\Theta(n^2)$

A graph with $O(n)$ edges is called **sparse**.
Basic notions

- \( u \) and \( v \) are adjacent (or “neighbours”) if \( (u,v) \in E \)
- \( u \in V \) is incident to \( e \in E \) if \( e = (u,v) \) for some \( v \in V \)
- the degree of \( v \), \( \deg(v) = \) number of edges incident to \( v \)
- for a directed graph we have indegree and outdegree

- a path is a sequence of vertices \( v_1, v_2, \ldots, v_k \) such that \( (v_i,v_{i+1}) \in E, i = 1 \ldots k-1 \).
  The length of a path is the number of edges.
- a simple path does not repeat vertices
- a cycle is a path that starts and ends at the same vertex, and a simple cycle does not repeat vertices. The length is the number of edges.
CAUTION: some sources use “path” to mean simple path. When in doubt, ask.

- a tree is a connected undirected graph without cycles
- an undirected graph is connected if every \( u,v \in V \) are joined by a path
- a connected component of a graph is a maximal connected subgraph

indegree = 2  outdegree = 3
First, Euler pointed out that the choice of route inside each land mass is irrelevant. The only important feature of a route is the sequence of bridges crossed. This allowed him to reformulate the problem in abstract terms (laying the foundations of graph theory), eliminating all features except the list of land masses and the bridges connecting them. In modern terms, one replaces each land mass with an abstract "vertex" or node, and each bridge with an abstract connection, an "edge", which only serves to record which pair of vertices (land masses) is connected by that bridge. The resulting mathematical structure is a graph.

Since only the connection information is relevant, the shape of pictorial representations of a graph may be distorted in any way, without changing the graph itself. Only the existence (or absence) of an edge between each pair of nodes is significant. For example, it does not matter whether the edges drawn are straight or curved, or whether one node is to the left or right of another.

Next, Euler observed that (except at the endpoints of the walk), whenever one enters a vertex by a bridge, one leaves the vertex by a bridge. In other words, during any walk in the graph, the number of times one enters a non-terminal vertex equals the number of times one leaves it. Now, if every bridge has been traversed exactly once, it follows that, for each land mass (except for the ones chosen for the start and finish), the number of bridges touching that land mass must be even (half of them, in the particular traversal, will be traversed "toward" the landmass; the other half, "away" from it). However, all four of the land masses in the original problem are touched by an odd number of bridges (one is touched by 5 bridges, and each of the other three is touched by 3). Since, at most, two land masses can serve as the endpoints of a walk, the proposition of a walk traversing each bridge once leads to a contradiction.

In modern language, Euler shows that the possibility of a walk through a graph, traversing each edge exactly once, depends on the degrees of the nodes. The degree of a node is the number of edges touching it. Euler's argument shows that a necessary condition for the walk of the desired form is that the graph be connected and have exactly zero or two nodes of odd degree. This condition turns out also to be sufficient—a result stated by Euler and later proved by Carl Hierholzer. Such a walk is now called an Eulerian path or Euler walk in his honor. Further, if there are nodes of odd degree, then any Eulerian path will start at one of them and end at the other. Since the graph corresponding to historical Königsburg has four nodes of odd degree, it cannot have an Eulerian path.

An alternative form of the problem asks for a path that traverses all bridges and also has the same starting and ending point. Such a walk is called an Eulerian circuit or Euler tour. Such a circuit exists if, and only if, the graph is connected, and there are no nodes of odd degree at all. All Eulerian circuits are also Eulerian paths, but not all Eulerian paths are Eulerian circuits.

Graphs first used by Euler in 1735 for the Königsburg bridge problem

many, many applications

https://en.wikipedia.org/wiki/Graph_theory
Sierpinski triangle from Towers of Hanoi

Apache dependency graph

Stanford bunny

Darwin, evolutionary tree

caffeine
Data Structure for Graphs

We focus on core graph problems. Assume the vertices are 1, 2, . . . , n. We may call these $v_1, v_2, \ldots, v_n$, or use letters (e.g. “for each vertex $v$ . . . “)

In practical applications, graphs have labels for vertices and for edges. So we need a mapping from the labels (names) to the vertex numbers. Also, we may want to add/delete vertices and edges (a “dynamic” graph).
Reference: Robert Sedgewick, Algorithms (CS 240 text)
Data Structure for Graphs

Two basic ways to store a graph with vertices 1, 2, \ldots, n:

**Adjacency matrix:**

\[
\begin{array}{cccc}
  & 1 & 2 & \ldots & n \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
\end{array}
\]

- \(n \times n\) matrix
- space \(\Theta(n^2)\)
- \(A[i,j] = 1\) if \((i,j) \in E\)
- \(0\) otherwise

**Adjacency lists:** for each vertex \(u\) store a linked list of its [forward] neighbours, i.e., vertices \(v\) such that \((u,v) \in E\).

- space \(\Theta(n + m)\)
- recall \(m = |E|\)
- We often measure graph algorithms in terms of \(n\) and \(m\)

**Example**

Undirected graph with adjacency matrix:

\[
\begin{array}{cccc}
  & 1 & 2 & 3 & 4 \\
1 & 0 & 1 & 1 & 0 \\
2 & 1 & 0 & 1 & 0 \\
3 & 1 & 1 & 0 & 1 \\
4 & 0 & 0 & 1 & 0
\end{array}
\]

Adjacency list:

- 1: 2 → 3
- 2: 3 → 1
- 3: 1 → 4 → 2
- 4: 3

Diagram of the graph with vertices labeled and edges connecting them.
Data Structure for Graphs

Two basic ways to store a graph with vertices 1, 2, . . . , n:

**Adjacency matrix:**

- A[n x n matrix with space Θ(n^2)]
- \[ A_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases} \]

**Adjacency lists:** for each vertex u store a linked list of its [forward] neighbours, i.e., vertices v such that (u,v) ∈ E.

- space Θ(n + m)
- recall m = |E|

We often measure graph algorithms in terms of n and m.

**Example**

Directed

Exercise

more examples in CLRS or see Erickson
Which model of computing for graph algorithms? a word-RAM

Recall:
Bit model: count bits for space and size of input; count bit operations for time. Appropriate for problems with one or two numbers as input.
Examples: - multiply 2 numbers
           - test if a number is prime.

Word RAM model: count number of words for space and size of input; count word operations for time. To keep this realistic, we limit $w$, the number of bits per word. If the input has $N$ words, we use $w = O(\log N)$.
Examples: sorting, input is $N$ numbers. $O(N \log N)$ if numbers are $O(\log N)$ bits.

A graph has $n$ vertices, each needing $O(\log n)$ bits, and $m = O(n^2)$ edges. Use the word RAM. Input size is $O(n + m)$.

Later we will consider weights on the edges (e.g. road network). Use word RAM. i.e. assume arithmetic on weights takes $O(1)$.
## Basic operations for graphs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Adjacency Matrix</th>
<th>Adjacency Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>list all of v’s neighbours</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(1 + \deg(v))$</td>
</tr>
<tr>
<td>list all edges</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n + m)$</td>
</tr>
<tr>
<td>is (u,v) an edge?</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1 + \deg(v))$</td>
</tr>
<tr>
<td>space</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n + m)$</td>
</tr>
</tbody>
</table>

For algorithms in this course, we will use adjacency lists (unless explicitly stated otherwise)
### Basic operations for graphs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Adjacency matrix</th>
<th>Adjacency lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>List all of v’s neighbours</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1 + \text{deg}(v))$</td>
</tr>
<tr>
<td>List all edges</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n + m)$</td>
</tr>
<tr>
<td>Is (u,v) an edge?</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1 + \text{deg}(v))$</td>
</tr>
<tr>
<td>Space</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n + m)$</td>
</tr>
</tbody>
</table>

For algorithms in this course, we will use adjacency lists (unless explicitly stated otherwise).

$O(n + m)$ is called linear time for a graph algorithm.
Summary of Lecture 12, part 1

- graphs, definitions, data structures

What you should know from Lecture 12, part 1:

- how to store graphs and work with adjacency lists
- what $O(n+m)$ means, and how big $m$ can be

Next:

- exploring graphs, basic graph algorithms
Exploring a graph

Visit all nodes or all nodes reachable from some “source”.
Further: find shortest paths, connected components, . . .

Breadth-First/Depth-First Search

Breadth First Search — a cautious search where we check all vertices one edge away from the start node, then two edges away, etc.
Use a Queue to store vertices that have been *discovered* but not yet *explored*. Vertices are first marked *undiscovered* and change to *discovered*.

### Explore(v)
- for each neighbour u of v
- if mark(u) = undiscovered
  - mark(u) := discovered
  - add u to Queue
  - \(\text{parent}(u) \leftarrow v; \text{level}(u) \leftarrow \text{level}(v) + 1\)

### BFS\((v_0)\) — for initial vertex \(v_0\)
- initialize: mark every vertex undiscovered
- add \(v_0\) to Queue; mark\((v_0) := \text{discovered}\)
- while Queue not empty
  - pop \(v\) from Queue
  - Explore\((v)\)
  - \(\text{parent}(v_0) \leftarrow \emptyset; \text{level}(v_0) \leftarrow 0\)

It is also useful to store parent\((v)\) and level\((v)\)

![Graph](image.png)

- Level 1:
  - Vertices 1, 3, 6, 8
- Level 2:
  - Vertices 2, 4, 5, 7
Use a Queue to store vertices that have been *discovered* but not yet *explored*. Vertices are first marked *undiscovered* and change to *discovered*.

### **Explore(v)**
- for each neighbour u of v
- if mark(u) = undiscovered
  - mark(u) := discovered; parent(u) := v; level(u) := level(v) + 1;
  - add u to Queue

### **BFS(v₀)** — for initial vertex v₀
- initialize: mark every vertex undiscovered
- add v₀ to Queue; mark(v₀) := discovered; parent(v₀) := ∅; level(v₀) := 0;
- while Queue not empty
  - pop v from Queue
  - Explore(v)

**Analysis**: BFS takes $O(n+m)$ time.
We explore each vertex at most once and check all its neighbours

\[ \text{Time} = O(n + \sum_v \deg(v)) = O(n + m) \]

**Note**: $\sum_v \deg(v) = 2m$ because each edge is counted twice.
Properties of BFS

1. the parent pointers create a directed tree

2. if (u, v) is an edge then level(u) and level(v) differ by 0 or 1.

**Note:** we explore all vertices on level i before starting level i + 1.

Suppose v explored before u.
We check all edges (v, w).
If u is undiscovered then
level(w) ≤ level(v) + 1

Else u is discovered but not explored - thus
in level(v) or level(v) + 1
Properties of BFS

3. vertex \( v \) is connected to \( v_0 \) iff BFS from \( v_0 \) reaches \( v \)

Stronger: \( \text{level}(v) = \) the length of a shortest path from \( v_0 \) to \( v \)

**Lemma.** vertex \( v \) is on level \( i \) iff the shortest path from \( v_0 \) to \( v \) has length \( i \)

**Proof.** By induction on \( i \).

\( i = 0 \). shortest path \( v_0 \to v_0 \) has length 0.

\( i > 0 \). Assume true for \( j < i \).

\( \Rightarrow \) vertex \( v \) on level \( i \) \( \Rightarrow \) parent \( (v) \) on level \( i-1 \)

\( \Rightarrow \) (induction) \exists \) path \( v_0 \) to parent \( (v) \) of length \( i-1 \)

\( \Rightarrow \) \( \exists \) path \( v_0 \) to \( v \) of length \( i \). Is it shortest?

If there is a shorter path then (by ind.) \( v \) on level \( < i \).

\( \Leftarrow \) shortest path \( v_0 \) to \( v \) has length \( i \). Let \( u \) be 2nd last vertex of path. Then shortest path \( v_0 \) to \( u \) has length \( i-1 \).

\( \Rightarrow \) (ind.) \( u \) on level \( i-1 \) \( \Rightarrow \) by property 2

\( v \) on level \( \leq i \). If \( v \) on level \( < i \) then by ind. \( \exists \) shorter path \( v_0 \to v \).
Applications of BFS (for undirected graphs)

1. find all connected components of a graph
   Exercise. Enhance the pseudocode to do this.

2. Find the shortest path from a start vertex \( v_0 \) to any other vertex \( v \)
   — just return \( \text{level}(v) \)

3. Test if a graph has a cycle.
   Exercise.

4. Test if a graph is bipartite.
Testing if a graph is bipartite

A graph $G = (V,E)$ is **bipartite** if we can partition $V$ into $V_1$, $V_2$ such that every edge has one end in $V_1$ and the other end in $V_2$.

Note: If $G$ is bipartite then it has no odd length cycle.

### Test bipartite

Run BFS

$V_1 :=$ vertices in odd numbered levels  
$V_2 :=$ vertices in even numbered levels

for each edge $(u,v)$

if $u,v$ in $V_1$ OR $u,v$ in $V_2$ return NO  
return YES

Runtime: $O(n+m)$

Correctness

if the algorithm returns YES then $G$ is bipartite

must prove that if the algorithm returns NO then $G$ is not bipartite
Testing if a graph is bipartite

\( V_1 := \) odd levels \hspace{0.5cm} \( V_2 := \) even levels

**Claim.** If there is an edge in \( V_1 \) or in \( V_2 \) then \( G \) is not bipartite

Suppose edge \((u, v)\) \( u, v \in V_i \) \( i = 1 \) or \( i = 2 \)
Then \( \text{Property 2} \) \( \text{level}(u) \) and \( \text{level}(v) \) differ by \( 0 \) or \( 1 \)
But can't differ by \( 1 \) --- one in \( V_1 \) and one in \( V_2 \).
Thus \( u, v \) are in same level, say level \( k \).

\[
\begin{aligned}
\text{Let } z &= \text{least common ancestor of } u \text{ and } v \\
&\text{in parent tree. } z \text{ in level } l \\
&\text{Get cycle } C \\
&\text{length of } C = 2t + 1 \\
&t = k - l
\end{aligned}
\]

This is an odd length cycle.

\( \therefore G \) is not bipartite.
Testing if a graph is bipartite

This proves:

**Lemma.** G is bipartite iff it has no odd cycle.

The proof was via an algorithm that finds a bipartition OR an odd cycle.
Summary of Lecture 12

- graphs, definitions, data structures

- Breadth First Search

What you should know from Lecture 12:

- how to store graphs and do basic graph algorithms

- BFS and its uses

Next:

- Depth First Search for undirected and directed graphs