Introduction

What is common among:

- cities and highways between them
- computers and network cables between them
- people and relationships
- in a board game: a state and legal moves to other states

Mathematicians call it a “graph”. You probably think “network”.

Graph (Undirected): Definition

An undirected graph has:

- a set of vertices
- a set of edges: an edge is a set of two vertices (usually, we disallow edges from a vertex to itself)

Each edge does not specify a direction, you may go both ways.

Mathematicians write “an undirected graph is a pair \((V, E)\) where…”

Quiz: What is the maximum possible number of edges?

\[|V| \cdot (|V| - 1)/2\]

\[\Theta(|V|^2)\]

Terminology: Degree

The degree of a vertex is how many edges “touch” it.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>degree</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Terminology: Adjacent

Two vertices are adjacent iff there is an edge between them.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>√</td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>C</td>
<td>√</td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This brings us to two nice ways to store a graph…
**Storing A Graph: Adjacency Matrix**

![Graph Diagram](image)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adjacency matrix = store this in a 2D array.

- space: $\Theta(|V|^2)$
- “who are adjacent to $v$?”: $\Theta(|V|)$ time
- “are $v$ and $w$ adjacent?”: $\Theta(1)$ time
- convenient for some other operations/queries

---

**Storing A Graph: Adjacency List**

![Graph Diagram](image)

<table>
<thead>
<tr>
<th>is adjacent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>E</td>
</tr>
</tbody>
</table>

Adjacency list = store this in a 1D array or dictionary.

Use a list or a set for each entry on the right.

- space: $\Theta(|V| + |E|)$
- “who are adjacent to $v$?”: $\Theta(deg(v))$ time
- “are $v$ and $w$ adjacent?”: $\Theta(deg(v))$ time if list
- optimal for graph searches

---

**Terminology: Path**

A simple path is a sequence of vertices in which

- consecutive vertices are adjacent
- vertices are distinct
  (if not, you can say “path”, but not “simple path”)

Some versions specify a $\langle$vertex, edge, vertex$\ldots$\rangle sequence.

$\langle D, B, C \rangle$ is a simple path. $\langle D, B, C, B \rangle$ is a path, but not simple. $\langle D, A, B \rangle$ is not a path.

---

**Terminology: Cycle**

A simple cycle is a sequence of vertices in which

- consecutive vertices are adjacent
- edges used are distinct
- first vertex = last vertex
- vertices are distinct except for first/last
  (if not, you can say “cycle”, but not “simple cycle”)

Some versions specify a $\langle$vertex, edge, vertex$\ldots$\rangle sequence.

$\langle B, C, A, B \rangle$ is a simple cycle. $\langle B, C, A, B, D, B \rangle$ is a cycle, but not simple.
**Terminology: (Dis)Connected**

A graph is connected iff between every two different vertices there is a path.

A graph is disconnected iff not connected.

Disconnected:

```
A -- B -- D
  |   |
  C   E
```

Connected:

```
A -- B -- D
  |   |
  C   E
```

**Terminology: Tree**

A tree is a special case of graphs. A tree is connected and has no cycles.

A forest has no cycles (may be disconnected); a collection of trees.

**Breadth-First Search**

Specify or arbitrarily pick a start vertex.

0. visit the start vertex
1. visit vertices 1 edge away from the start vertex
2. visit vertices 2 edges away from the start vertex
3. 

```
0  1  2  3  4
  |   |   |
  1   2   3
```

Breadth-First Search

```
start := pick a vertex
q := new Queue()
q.enqueue(start)
mark: start is seen
// start's distance is 0
while not q.empty():
  u := q.dequeue()
  for each v adjacent to u:
    if v not yet seen:
      q.enqueue(v)
      mark: v is seen
      // edge \{u,v\} is a discovery edge
      // v's predecessor is u
      // v's distance is u's distance + 1
```
Findings of Breadth-First Search

- discovery edges and predecessors
- a spanning tree/forest from them (spanning tree = a tree reaching all reachable vertices)
- for each vertex: a shortest path and distance from start

Unvisited Vertices

What if some vertex is not visited?

- Depends on your purpose.
  - To determine which vertices are reachable from the start: now you have the answer.
  - To visit all vertices: another round of BFS from an unvisited vertex.

Breadth-First Search Time

BFS does:

- enqueue and dequeue each vertex once
- consider each edge twice
- find each vertex's adjacency list once
- check each vertex's "seen" status \(\text{deg}(v)\) times

Assume \(\Theta(1)\) time for

- marking/checking a vertex's "seen" status
- finding a vertex's adjacency list

Then BFS total time: \(\Theta(|V| + |E|)\).
Depth-First Search

Specify or arbitrarily pick a start vertex.

0. visit the start vertex
1. choose one adjacent, unvisited vertex of the previous; visit it
2. choose one adjacent, unvisited vertex of the previous; visit it
3. ... 
4. whenever you have no choice, backtrack to the last time you had a choice, choose another one

It's hard to say it right. The next 15 slides show an example in action.
Depth-First Search Progression

1. Choose an adjacent, unvisited vertex to visit.

2. No adjacent, unvisited vertex; backtrack.

3. Choose an adjacent, unvisited vertex to visit.
no adjacent, unvisited vertex; backtrack

choose an adjacent, unvisited vertex to visit
Depth-First Search Progression

no adjacent, unvisited vertex; backtrack

no adjacent, unvisited vertex; backtrack

no adjacent, unvisited vertex; backtrack

no adjacent, unvisited vertex; nothing to backtrack, the end.
Depth-First Search

Most straightforward to code up recursively.

DFS-Visit(u):
mark: u is visited
for each v adjacent to u:
   if v not visited yet:
      // edge \{u,v\} is a discovery edge
      // v’s predecessor is u
      DFS-Visit(v)

start := pick a vertex
DFS-Visit(start)

Findings of Depth-First Search

- discovery edges and predecessors
- a spanning tree/forest from them

Unvisited Vertices

What if some vertex is not visited?

Depends on your purpose.

- To determine which vertices are reachable from the start: now you have the answer.
- To visit all vertices: another round of DFS from an unvisited vertex.

Depth-First Search Time

DFS does:

- visit each vertex once
- consider each edge twice
- find each vertex's adjacency list once
- check each vertex’s “visited” status \(\text{deg}(v)\) times

Similar to BFS, total time is \(\Theta(|V| + |E|)\) or \(\Theta((|V| + |E|) \cdot \log(|V|))\) depending on:

- time to mark/check a vertex’s “visited” status
- time to find a vertex’s adjacency list
Directed Graph

A directed graph has:

- a set of vertices
- a set of edges: an edge is a tuple of two vertices
  (usually, we disallow edges from a vertex to itself)

Each edge specifies one direction.
(A, B) lets you go from A to B, if present.
(B, A) lets you go from B to A, if present.

Many definitions need slight modifications.

Modifications for Directed Graphs

- out-degree: how many edges go out of a vertex
- in-degree: how many edges go into a vertex
- path, cycle:
  must comply with edge directions
- BFS, DFS:
  "for each v adjacent to u": requires edge direction (u,v)
- DFS classifies edges into:
  - discovery
  - back: from a vertex to an ancestor
    important for cycle detection
  - forward: from a vertex to a descendent, non-discovery
  - cross: others