Representing and Exploring Graphs
(Depth First Search and Breadth First Search)

Benjamin Sach
Graph notation recap

\( G \) is a Graph,

\( V \) is the set of vertices
\[
|V| \text{ is the number of vertices}
\]

\( E \) is the set of edges
\[
|E| \text{ is the number of edges}
\]

\( v \in V \) is a vertex

\( (u, v) \in E \) is an edge from \( u \) to \( v \)

This lecture focuses on exploring **undirected** and **unweighted** graphs

*though everything discussed will work for directed, unweighted graphs too*
Graph notation recap

- An **undirected** and **unweighted** graph
- An **directed** and **unweighted** graph
- An **undirected** and **weighted** graph
- A **directed** and **weighted** graph
\( V \) is the set of vertices

\(|V|\) is the number of vertices

\( E \) is the set of edges

\(|E|\) is the number of edges

We will in general assume that the vertices are numbered \(1, 2, 3 \ldots |V|\)
Graph representations

$V$ is the set of vertices
$|V|$ is the number of vertices

$E$ is the set of edges
$|E|$ is the number of edges

$V$ is the set of vertices
$|V|$ is the number of vertices

We will in general assume that the vertices are numbered $1, 2, 3 \ldots |V|$
Graph representations

$V$ is the set of vertices
$|V|$ is the number of vertices

$E$ is the set of edges
$|E|$ is the number of edges

$V$ is the set of vertices
$|V|$ is the number of vertices

We will in general assume that the vertices are numbered $1, 2, 3 \ldots |V|$
Graph representations

$V$ is the set of vertices
$|V|$ is the number of vertices

$E$ is the set of edges
$|E|$ is the number of edges

Adjacency Matrix

$V$ is the set of vertices
$|V|$ is the number of vertices

TO

```plaintext
FROM
1 2 3 4 5 6 7 8 9
1 0 1 1 0 0 0 0 0 0
2 1 0 1 1 1 0 0 0 0
3 1 1 0 1 1 0 0 0 0
4 0 1 1 0 1 0 0 0 0
5 0 1 1 1 0 1 0 0 0
6 0 0 0 1 1 0 0 0 0
7 0 0 0 0 0 0 0 1 1
8 0 0 0 0 0 0 1 0 1
9 0 0 0 0 0 1 1 0
```

We will in general assume that the vertices are numbered $1, 2, 3 \ldots |V|$
$V$ is the set of vertices
$|V|$ is the number of vertices

$E$ is the set of edges
$|E|$ is the number of edges

Adjacency Matrix

We will in general assume that the vertices are numbered $1, 2, 3 \ldots |V|$
\( V \) is the set of vertices
\(|V|\) is the number of vertices
\( E \) is the set of edges
\(|E|\) is the number of edges

We will in general assume that the vertices are numbered 1, 2, 3 \ldots |V|
\( V \) is the set of vertices
\(|V| \) is the number of vertices

\( E \) is the set of edges
\(|E| \) is the number of edges

Adjacency List
(using linked lists)

We will in general assume that the vertices are numbered \(1, 2, 3 \ldots |V|\)
Graph representations

$V$ is the set of vertices
$|V|$ is the number of vertices

$E$ is the set of edges
$|E|$ is the number of edges

Adjacency List
(using linked lists)

We will in general assume that the vertices are numbered $1, 2, 3 \ldots |V|$
Graph representations

$V$ is the set of vertices
$|V|$ is the number of vertices

$E$ is the set of edges
$|E|$ is the number of edges

Adjacency List
(using linked lists)

We will in general assume that the vertices are numbered $1, 2, 3 \ldots |V|$
Graph representations

\[ V \text{ is the set of vertices} \]
\[ |V| \text{ is the number of vertices} \]

\[ E \text{ is the set of edges} \]
\[ |E| \text{ is the number of edges} \]

We will in general assume that the vertices are numbered \( 1, 2, 3 \ldots |V| \)

Adjacency List
(\textit{using linked lists})
$V$ is the set of vertices
$|V|$ is the number of vertices

$E$ is the set of edges
$|E|$ is the number of edges

We will in general assume that the vertices are numbered $1, 2, 3 \ldots |V|$

Both representations are symmetric because the graphs are undirected
**Basic operations**

- **Adjacency Matrix**
- **Adjacency List (using linked lists)**

<table>
<thead>
<tr>
<th>Space</th>
<th>Adjacency Matrix</th>
<th>Adjacency List (using linked lists)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is there an edge from vertex $u$ to $v$?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>List all the edges leaving $u \in V$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**V** is the set of vertices

$|V|$ is the number of vertices

**E** is the set of edges

$|E|$ is the number of edges

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

**Adjacency Matrix**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

**Adjacency List (using linked lists)**
### Basic operations

<table>
<thead>
<tr>
<th></th>
<th>Adjacency Matrix</th>
<th>Adjacency List (using linked lists)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space</strong></td>
<td>$\Theta(</td>
<td>V</td>
</tr>
<tr>
<td>Is there an edge from vertex $u$ to $v$?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>List all the edges leaving $u \in V$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
- $V$ is the set of vertices
- $|V|$ is the number of vertices
- $E$ is the set of edges
- $|E|$ is the number of edges
- The adjacency matrix is a square matrix $A$ with $A_{ij} = 1$ if there is an edge from vertex $i$ to vertex $j$, and $A_{ij} = 0$ otherwise.
- The adjacency list is a list of linked lists, where each list corresponds to a vertex and contains the adjacent vertices.

**Example:**
- **Adjacency Matrix:**
  
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
</table>
  1 | 1 | 1 | 0 | 1 | 0 |
  2 | 1 | 0 | 1 | 1 | 1 |
  3 | 1 | 1 | 0 | 1 | 1 |
  4 | 0 | 1 | 1 | 0 | 1 |
  5 | 0 | 1 | 1 | 1 | 0 |
- **Adjacency List:**
  
  - 1: 2, 3
  - 2: 1, 3, 4
  - 3: 1
  - 4: 2, 3
  - 5: 2
### Basic operations

<table>
<thead>
<tr>
<th></th>
<th>Adjacency Matrix</th>
<th>Adjacency List (using linked lists)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space</strong></td>
<td>$\Theta(</td>
<td>V</td>
</tr>
<tr>
<td>Is there an edge from vertex $u$ to $v$?</td>
<td>$O(1)$ time</td>
<td></td>
</tr>
<tr>
<td>List all the edges leaving $u \in V$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Adjacency Matrix

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adjacency List (using linked lists)

$V$ is the set of vertices

$|V|$ is the number of vertices

$E$ is the set of edges

$|E|$ is the number of edges
<table>
<thead>
<tr>
<th>Basic operations</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Adjacency Matrix</th>
<th>Adjacency List (using linked lists)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>$\Theta(</td>
<td>V</td>
</tr>
<tr>
<td>Is there an edge from vertex $u$ to $v$?</td>
<td>$O(1)$ time</td>
<td>$O(V)$ time</td>
</tr>
<tr>
<td>List all the edges leaving $u \in V$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$V$ is the set of vertices
$|V|$ is the number of vertices

$E$ is the set of edges
$|E|$ is the number of edges

### Adjacency Matrix

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

### Adjacency List

- 1 -> 2, 3
- 2 -> 1, 3, 4
- 3 -> 1
- 4 -> 2, 3
- 5 -> 2
<table>
<thead>
<tr>
<th>Basic operations</th>
</tr>
</thead>
</table>

### Adjacency Matrix

1. **Space**: \( \Theta(|V|^2) \)
2. **Is there an edge from vertex \( u \) to \( v \)?**: \( O(1) \) time
3. **List all the edges leaving \( u \in V \)**

<table>
<thead>
<tr>
<th>Adjacency List (using linked lists)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space</strong>: ( \Theta(</td>
</tr>
<tr>
<td><strong>Is there an edge from vertex ( u ) to ( v )?</strong>: ( O(\text{deg}(u)) ) time</td>
</tr>
</tbody>
</table>

- \( V \) is the set of vertices
- \( |V| \) is the number of vertices
- \( E \) is the set of edges
- \( |E| \) is the number of edges
**Basic operations**

<table>
<thead>
<tr>
<th></th>
<th>Adjacency Matrix</th>
<th>Adjacency List (using linked lists)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space</strong></td>
<td>$\Theta(</td>
<td>V</td>
</tr>
<tr>
<td>Is there an edge from vertex $u$ to $v$?</td>
<td>$O(1)$ time</td>
<td>$O(deg(u))$ time</td>
</tr>
<tr>
<td>List all the edges leaving $u \in V$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$V$ is the set of vertices
$|V|$ is the number of vertices

$E$ is the set of edges
$|E|$ is the number of edges

$deg(u)$ - *the degree*, is the number of edges leaving vertex $u$
**Basic operations**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Adjacency Matrix</th>
<th>Adjacency List (using linked lists)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>$\Theta(</td>
<td>V</td>
</tr>
<tr>
<td>Is there an edge from vertex $u$ to $v$?</td>
<td>$O(1)$ time</td>
<td>$O(\text{deg}(u))$ time</td>
</tr>
<tr>
<td>List all the edges leaving $u \in V$</td>
<td>$O(V)$ time</td>
<td></td>
</tr>
</tbody>
</table>

$V$ is the set of vertices
| $|V|$ is the number of vertices

$E$ is the set of edges
| $|E|$ is the number of edges

$\text{deg}(u)$ - the degree, is the number of edges leaving vertex $u$
**Basic operations**

- **Adjacency Matrix**
  - Space: $\Theta(|V|^2)$
  - Is there an edge from vertex $u$ to $v$? $O(1)$ time
  - List all the edges leaving $u \in V$ $O(|V|)$ time

- **Adjacency List (using linked lists)**
  - Space: $\Theta(|V| + |E|)$
  - Is there an edge from vertex $u$ to $v$? $O(\deg(u))$ time
  - List all the edges leaving $u \in V$ $O(\deg(u))$ time

---

$V$ is the set of vertices
$|V|$ is the number of vertices

$E$ is the set of edges
$|E|$ is the number of edges

$\deg(u)$ - the degree, is the number of edges leaving vertex $u$
### Basic operations

<table>
<thead>
<tr>
<th></th>
<th>Adjacency Matrix</th>
<th>Adjacency List (using linked lists)</th>
<th>Adjacency List (using hash tables)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space</strong></td>
<td>$\Theta(</td>
<td>V</td>
<td>^2)$</td>
</tr>
<tr>
<td>Is there an edge from vertex $u$ to $v$?</td>
<td>$O(1)$ time</td>
<td>$O(\text{deg}(u))$ time</td>
<td>$O(1)$ time</td>
</tr>
<tr>
<td>List all the edges leaving $u \in V$</td>
<td>$O(V)$ time</td>
<td>$O(\text{deg}(u))$ time</td>
<td>$O(\text{deg}(u))$ time</td>
</tr>
</tbody>
</table>

$V$ is the set of vertices  
$|V|$ is the number of vertices  

$E$ is the set of edges  
$|E|$ is the number of edges  

$\text{deg}(u)$ - the degree, is the number of edges leaving vertex $u$
### Basic operations

<table>
<thead>
<tr>
<th></th>
<th>Adjacency Matrix</th>
<th>Adjacency List (using linked lists)</th>
<th>Adjacency List (using hash tables)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>$\Theta(</td>
<td>V</td>
<td>^2)$</td>
</tr>
<tr>
<td>Is there an edge from vertex $u$ to $v$?</td>
<td>$O(1)$ time</td>
<td>$O(\deg(u))$ time</td>
<td>$O(1)$ time</td>
</tr>
<tr>
<td>List all the edges leaving $u \in V$</td>
<td>$O(V)$ time</td>
<td>$O(\deg(u))$ time</td>
<td>$O(\deg(u))$ time</td>
</tr>
</tbody>
</table>

$\deg(u)$ - the degree, is the number of edges leaving vertex $u$

$V$ is the set of vertices  
$|V|$ is the number of vertices

$E$ is the set of edges  
$|E|$ is the number of edges

All three representations work for directed and/or weighted graphs too  
*(with the same complexities)*
**Basic operations**

<table>
<thead>
<tr>
<th></th>
<th>Adjacency Matrix</th>
<th>Adjacency List (using linked lists)</th>
<th>Adjacency List (using hash tables)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>$\Theta(</td>
<td>V</td>
<td>^2)$</td>
</tr>
<tr>
<td>Is there an edge from vertex $u$ to $v$?</td>
<td>$O(1)$ time</td>
<td>$O(\text{deg}(u))$ time</td>
<td>$O(1)$ time</td>
</tr>
<tr>
<td>List all the edges leaving $u \in V$</td>
<td>$O(V)$ time</td>
<td>$O(\text{deg}(u))$ time</td>
<td>$O(\text{deg}(u))$ time</td>
</tr>
</tbody>
</table>

$\text{deg}(u)$ - the degree, is the number of edges leaving vertex $u$

---

$V$ is the set of vertices

$|V|$ is the number of vertices

$E$ is the set of edges

$|E|$ is the number of edges

why don’t we always use these?

all three representations work for directed and/or weighted graphs too

(with the same complexities)
Basic operations

<table>
<thead>
<tr>
<th></th>
<th>Adjacency Matrix</th>
<th>Adjacency List (using linked lists)</th>
<th>Adjacency List (using hash tables)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>$\Theta(</td>
<td>V</td>
<td>^2)$</td>
</tr>
<tr>
<td>Is there an edge from vertex $u$ to $v$?</td>
<td>$O(1)$ time</td>
<td>$O(\text{deg}(u))$ time</td>
<td>$O(1)$ time</td>
</tr>
<tr>
<td>List all the edges leaving $u \in V$</td>
<td>$O(V)$ time</td>
<td>$O(\text{deg}(u))$ time</td>
<td>$O(\text{deg}(u))$ time</td>
</tr>
</tbody>
</table>

$V$ is the set of vertices  
$|V|$ is the number of vertices  

$E$ is the set of edges  
$|E|$ is the number of edges  

why don’t we always use these?
### Basic Operations

<table>
<thead>
<tr>
<th></th>
<th>Adjacency Matrix</th>
<th>Adjacency List (using linked lists)</th>
<th>Adjacency List (using hash tables)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space</strong></td>
<td>$\Theta(</td>
<td>V</td>
<td>^2)$</td>
</tr>
<tr>
<td><strong>Is there an edge from vertex $u$ to $v$?</strong></td>
<td>$O(1)$ time</td>
<td>$O(\text{deg}(u))$ time</td>
<td>$O(1)$ time</td>
</tr>
<tr>
<td><strong>List all the edges leaving $u \in V$</strong></td>
<td>$O(V)$ time</td>
<td>$O(\text{deg}(u))$ time</td>
<td>$O(\text{deg}(u))$ time</td>
</tr>
</tbody>
</table>

Basic hash tables give *expected* time complexities (constant time ‘on average’)
- no worst case guarantees against collisions

---

$V$ is the set of vertices
$|V|$ is the number of vertices

$E$ is the set of edges
$|E|$ is the number of edges
Basic operations

<table>
<thead>
<tr>
<th></th>
<th>Adjacency Matrix</th>
<th>Adjacency List (using linked lists)</th>
<th>Adjacency List (using hash tables)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space</strong></td>
<td>$\Theta(</td>
<td>V</td>
<td>^2)$</td>
</tr>
<tr>
<td>Is there an edge from vertex $u$ to $v$?</td>
<td>$O(1)$ time</td>
<td>$O(\text{deg}(u))$ time</td>
<td>$O(1)$ time</td>
</tr>
<tr>
<td>List all the edges leaving $u \in V$</td>
<td>$O(V)$ time</td>
<td>$O(\text{deg}(u))$ time</td>
<td>$O(\text{deg}(u))$ time</td>
</tr>
</tbody>
</table>

Basic hash tables give expected time complexities (constant time ‘on average’)
- no worst case guarantees against collisions

If you’re interested in hashing with provable guarantees, come to **COMS31900** (Advanced Algorithms)
### Basic operations

<table>
<thead>
<tr>
<th></th>
<th>Adjacency Matrix</th>
<th>Adjacency List (using linked lists)</th>
<th>Adjacency List (using hash tables)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space</strong></td>
<td>$\Theta(</td>
<td>V</td>
<td>^2)$</td>
</tr>
<tr>
<td><strong>Is there an edge from vertex $u$ to $v$?</strong></td>
<td>$O(1)$ time</td>
<td>$O(\text{deg}(u))$ time</td>
<td>$O(1)$ time</td>
</tr>
<tr>
<td><strong>List all the edges leaving $u \in V$</strong></td>
<td>$O(V)$ time</td>
<td>$O(\text{deg}(u))$ time</td>
<td>$O(\text{deg}(u))$ time</td>
</tr>
</tbody>
</table>

Basic hash tables give *expected* time complexities (constant time ‘on average’)
- no worst case guarantees against collisions

If you’re interested in hashing with provable guarantees, come to **COMS31900** (Advanced Algorithms)

Fortunately, few algorithms need to ask “is there an edge?”

Normally, “tell me all the edges” is fine

$V$ is the set of vertices  
$|V|$ is the number of vertices

$E$ is the set of edges  
$|E|$ is the number of edges
Basic operations

why don’t we always use these?

<table>
<thead>
<tr>
<th></th>
<th>Adjacency Matrix</th>
<th>Adjacency List (using linked lists)</th>
<th>Adjacency List (using hash tables)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>(\Theta(</td>
<td>V</td>
<td>^2))</td>
</tr>
<tr>
<td>Is there an edge from vertex (u) to (v)?</td>
<td>(O(1)) time</td>
<td>(O(\text{deg}(u))) time</td>
<td>(O(1)) time</td>
</tr>
<tr>
<td>List all the edges leaving (u \in V)</td>
<td>(O(V)) time</td>
<td>(O(\text{deg}(u))) time</td>
<td>(O(\text{deg}(u))) time</td>
</tr>
</tbody>
</table>

\(V\) is the set of vertices
\(|V|\) is the number of vertices

\(E\) is the set of edges
\(|E|\) is the number of edges
**Basic operations**

<table>
<thead>
<tr>
<th></th>
<th>Adjacency Matrix</th>
<th>Adjacency List (using linked lists)</th>
<th>Adjacency List (using hash tables)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space</strong></td>
<td>$\Theta(</td>
<td>V</td>
<td>^2)$</td>
</tr>
<tr>
<td>Is there an edge from vertex $u$ to $v$?</td>
<td>$O(1)$ time</td>
<td>$O(\text{deg}(u))$ time</td>
<td>$O(1)$ time</td>
</tr>
<tr>
<td>List all the edges leaving $u \in V$</td>
<td>$O(V)$ time</td>
<td>$O(\text{deg}(u))$ time</td>
<td>$O(\text{deg}(u))$ time</td>
</tr>
</tbody>
</table>

$V$ is the set of vertices  
$|V|$ is the number of vertices  
$E$ is the set of edges  
$|E|$ is the number of edges

**why don’t we always use these?**

For this course we’ll stick to Adjacency Matrices and Lists (using linked lists)  
*(for today just Adjacency Lists)*
Basic operations

why don’t we always use these?

<table>
<thead>
<tr>
<th></th>
<th>Adjacency Matrix</th>
<th>Adjacency List (using linked lists)</th>
<th>Adjacency List (using hash tables)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>$\Theta(</td>
<td>V</td>
<td>^2)$</td>
</tr>
<tr>
<td>Is there an edge from vertex $u$ to $v$?</td>
<td>$O(1)$ time</td>
<td>$O(deg(u))$ time</td>
<td>$O(1)$ time</td>
</tr>
<tr>
<td>List all the edges leaving $u \in V$</td>
<td>$O(V)$ time</td>
<td>$O(deg(u))$ time</td>
<td>$O(deg(u))$ time</td>
</tr>
</tbody>
</table>

For this course we’ll stick to Adjacency Matrices and Lists (using linked lists)

(for today just Adjacency Lists)

Another reason to do so is that you may not have control over how your graph is stored.
Basic operations

<table>
<thead>
<tr>
<th></th>
<th>Adjacency Matrix</th>
<th>Adjacency List (using linked lists)</th>
<th>Adjacency List (using hash tables)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>$\Theta(</td>
<td>V</td>
<td>^2)$</td>
</tr>
<tr>
<td>Is there an edge from vertex $u$ to $v$?</td>
<td>$O(1)$ time</td>
<td>$O(\text{deg}(u))$ time</td>
<td>$O(1)$ time</td>
</tr>
<tr>
<td>List all the edges leaving $u \in V$</td>
<td>$O(V)$ time</td>
<td>$O(\text{deg}(u))$ time</td>
<td>$O(\text{deg}(u))$ time</td>
</tr>
</tbody>
</table>

For this course we’ll stick to Adjacency Matrices and Lists (using linked lists) *(for today just Adjacency Lists)*

Another reason to do so is that you may not have control over how your graph is stored.

*it may not even be stored as a graph...*
### Basic operations

<table>
<thead>
<tr>
<th></th>
<th>Adjacency Matrix</th>
<th>Adjacency List (using linked lists)</th>
<th>Adjacency List (using hash tables)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space</strong></td>
<td>$\Theta(</td>
<td>V</td>
<td>^2)$</td>
</tr>
<tr>
<td>Is there an edge from vertex $u$ to $v$?</td>
<td>$O(1)$ time</td>
<td>$O(\text{deg}(u))$ time</td>
<td>$O(1)$ time</td>
</tr>
<tr>
<td>List all the edges leaving $u \in V$</td>
<td>$O(V)$ time</td>
<td>$O(\text{deg}(u))$ time</td>
<td>$O(\text{deg}(u))$ time</td>
</tr>
</tbody>
</table>

For this course we’ll stick to Adjacency Matrices and Lists (using linked lists)

*(for today just Adjacency Lists)*

Another reason to do so is that you may not have control over how your graph is stored.

*it may not even be stored as a graph...*

Fortunately in many cases, you can *pretend* your graph is stored as one of the above.
Intersection Graphs
Intersection Graphs

The intersections of these circles correspond to the graph below.
Intersection Graphs

each circle is a vertex

the intersections of these circles correspond to the graph below
Intersection Graphs

The intersections of these circles correspond to the graph below.
Intersection Graphs

each intersection is an edge

the intersections of these circles correspond to the graph below
Intersection Graphs

each intersection is an edge

the intersections of these circles correspond to the graph below
Intersection Graphs

Each intersection is an edge.

If we only store the circles, we can pretend we have an Adjacency Matrix without building one.

The intersections of these circles correspond to the graph below.
Other intersection graphs... 

both of these collections also correspond to the example graph

(these are 1D intervals on a line spread out for visual clarity)
Other intersection graphs...

- = an edge
= a vertex

Both of these collections also correspond to the example graph

(These are 1D intervals on a line spread out for visual clarity)
Other intersection graphs...

- • = an edge
- \_ = a vertex

both of these collections also correspond to the example graph

there is an edge if two intervals intersect

(these are 1D intervals on a line spread out for visual clarity)
Other intersection graphs...

● = an edge
\ = a vertex

both of these collections also correspond to the example graph

there is an edge if two intervals intersect

Again,

we can pretend we have an Adjacency Matrix without building one

(these are 1D intervals on a line spread out for visual clarity)
Configuration Graphs

This is the configuration graph for the (4 disk) towers of Hanoi

A node is a state that the game can be in.

There is an edge if you can get from one state to another in a single move.
Configuration Graphs

This is the configuration graph for the (4 disk) towers of Hanoi

A node is a state that the game can be in.

There is an edge if you can get from one state to another in a single move.
Configuration Graphs

This is the configuration graph for the (4 disk) towers of Hanoi.

A node is a state that the game can be in.

There is an edge if you can get from one state to another in a single move.
Configuration Graphs

This is the configuration graph for the (4 disk) towers of Hanoi

A node is a state that the game can be in.

There is an edge if you can get from one state to another in a single move.
Configuration Graphs

This is the configuration graph for the (4 disk) towers of Hanoi

A node is a state that the game can be in.

There is an edge if you can get from one state to another in a single move.
Configuration Graphs

This is the configuration graph for the (4 disk) towers of Hanoi.

- A node is a state that the game can be in.
- There is an edge if you can get from one state to another in a single move.
Configuration Graphs

This is the configuration graph for the (4 disk) towers of Hanoi

A node is a state that the game can be in.

There is an edge if you can get from one state to another in a single move.
Configuration Graphs

This is the configuration graph for the (4 disk) towers of Hanoi.

A node is a state that the game can be in.

There is an edge if you can get from one state to another in a single move.
Configuration Graphs

This is the configuration graph for the (4 disk) towers of Hanoi.

A node is a state that the game can be in.

There is an edge if you can get from one state to another in a single move.
Configuration Graphs

This is the configuration graph for the (4 disk) towers of Hanoi.

A node is a state that the game can be in.

There is an edge if you can get from one state to another in a single move.

Storing this uses much less space than storing this graph.
Configuration Graphs

This is the configuration graph for the (4 disk) towers of Hanoi

A node is a state that the game can be in.

There is an edge if you can get from one state to another in a single move.

Storing this uses much less space than storing this graph.

We can pretend we have an Adjacency List without building one.
Exploring a graph

you are here

BAG
Exploring a graph

**Goal**: visit every vertex in a connected graph efficiently
Exploring a graph

The bag (*secretly a data structure*)

- We can put a vertex in the bag
- We can take a vertex from the bag
  *but we don’t know which one we’ll get*
- We don’t care how it works (*for now*)
  *(the bag doesn’t forget or change things)*

**Goal**: visit every vertex in a connected graph efficiently
Exploring a graph

The bag (secretly a data structure)

- We can put a vertex in the bag
- We can take a vertex from the bag
  - but we don’t know which one we’ll get
- We don’t care how it works (for now)
  - (the bag doesn’t forget or change things)

Goal: visit every vertex in a connected graph efficiently
Exploring a graph

**Goal:** visit every vertex in a connected graph efficiently
Exploring a graph

**Goal:** visit every vertex in a connected graph efficiently

**TRAVERSE**(s)

- put s into the bag
- while the bag is not empty
  - take u from the bag
  - if u unmarked
    - mark u
    - for every edge \((u,v)\)
      - put v into the bag
**TRAVVERSE**($s$)

put $s$ into the **bag**
while the **bag** is not empty
  take $u$ from the **bag**
  if $u$ unmarked
    mark $u$
    for every edge $(u, v)$
      put $v$ into the **bag**

**Goal:** visit every vertex in a connected graph efficiently
Exploring a graph

**Goal:** visit every vertex in a connected graph efficiently

**TRAVVERSE(s)**

- put s into the bag
- while the bag is not empty
  - take u from the bag
  - if u unmarked
    - mark u
  - for every edge (u,v)
    - put v into the bag
**TRVERSE**($s$)

put $s$ into the bag
while the bag is not empty
    take $u$ from the bag
    if $u$ unmarked
        mark $u$
        for every edge $(u,v)$
            put $v$ into the bag

**Goal:** visit every vertex in a connected graph efficiently
Exploring a graph

**Goal:** visit every vertex in a connected graph efficiently

**TRAVERSE**($s$)

1. put $s$ into the bag
2. while the bag is not empty:
   1. take $u$ from the bag
   2. if $u$ unmarked:
      1. mark $u$
      2. for every edge $(u,v)$:
         1. put $v$ into the bag
Exploring a graph

**Goal:** visit every vertex in a connected graph efficiently

**TRVERSE(s)**

put $s$ into the bag
while the bag is not empty
  take $u$ from the bag
  if $u$ unmarked
    mark $u$
    for every edge $(u, v)$
      put $v$ into the bag
Exploring a graph

\textbf{TRAVVERSE}(s)

1. put $s$ into the bag
2. while the bag is not empty
   1. take $u$ from the bag
   2. if $u$ unmarked
      1. mark $u$
      2. for every edge $(u,v)$
         1. put $v$ into the bag

\textbf{Goal}: visit every vertex in a connected graph efficiently
**Exploring a graph**

**TRVERSE(s)**

- put $s$ into the bag
- while the bag is not empty
  - take $u$ from the bag
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u,v)$
      - put $v$ into the bag

**Goal**: visit every vertex in a connected graph efficiently
Exploring a graph

**Goal:** visit every vertex in a connected graph efficiently

---

**TRAVESE**($s$)

put $s$ into the bag
while the bag is not empty
    take $u$ from the bag
    if $u$ unmarked
        mark $u$
        for every edge $\langle u, v \rangle$
            put $v$ into the bag
back in!
in twice!
TRAVERSE\( (s) \)

put \( s \) into the bag
while the bag is not empty
  take \( u \) from the bag
  if \( u \) unmarked
    mark \( u \)
    for every edge \( (u,v) \)
      put \( v \) into the bag

Goal: visit every vertex in a connected graph efficiently
Exploring a graph

**TRVERSE(s)**

put $s$ into the bag
while the bag is not empty
take $u$ from the bag
if $u$ unmarked
    mark $u$
    for every edge $(u,v)$
        put $v$ into the bag

Goal: visit every vertex in a connected graph efficiently
Exploring a graph

**Goal:** visit every vertex in a connected graph efficiently

**TRAVERSE(s)**

put \( s \) into the bag
while the bag is not empty
  take \( u \) from the bag
  if \( u \) unmarked
    mark \( u \)
    for every edge \((u,v)\)
      put \( v \) into the bag
**Traverse**($s$)

- put $s$ into the bag
- while the bag is not empty
  - take $u$ from the bag
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u,v)$
      - put $v$ into the bag

**Goal:** visit every vertex in a connected graph efficiently
Exploring a graph

**TRAVERSER**(s)

- put s into the bag
- while the bag is not empty
  - take u from the bag
  - if u unmarked
    - mark u
    - for every edge (u,v)
      - put v into the bag

**Goal**: visit every vertex in a connected graph efficiently
**TRVERSE**($s$)

put $s$ into the bag
while the bag is not empty
take $u$ from the bag
if $u$ unmarked
   mark $u$
   for every edge $(u, v)$
   put $v$ into the bag

**Goal:** visit every vertex in a connected graph efficiently
Exploring a graph

**TRVERSE(s)**

put $s$ into the bag
while the bag is not empty
  take $u$ from the bag
  if $u$ unmarked
    mark $u$
    for every edge $(u,v)$
      put $v$ into the bag

**Goal:** visit every vertex in a connected graph efficiently
**TRAVVERSE**($s$)

put $s$ into the bag

while the bag is not empty

take $u$ from the bag

if $u$ unmarked

  mark $u$

  for every edge $(u, v)$

    put $v$ into the bag

---

**Goal:** visit every vertex in a connected graph efficiently
**TRAVERSE**($s$)

- put $s$ into the **bag**
- while the **bag** is not empty
  - take $u$ from the **bag**
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u,v)$
      - put $v$ into the **bag**

**Goal:** visit every vertex in a connected graph efficiently
Exploring a graph

**Goal:** visit every vertex in a connected graph efficiently

**Traverse**($s$)

- put $s$ into the **bag**
- while the **bag** is not empty
  - take $u$ from the **bag**
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u,v)$
      - put $v$ into the **bag**
Exploring a graph

**TRVERSE(s)**

put $s$ into the bag
while the bag is not empty
  take $u$ from the bag
  if $u$ unmarked
    mark $u$
    for every edge $(u,v)$
      put $v$ into the bag

Goal: visit every vertex in a connected graph efficiently
Exploring a graph

**TRVERSE(s)**

put $s$ into the bag
while the bag is not empty
  take $u$ from the bag
  if $u$ unmarked
    mark $u$
    for every edge $(u, v)$
      put $v$ into the bag

**Goal**: visit every vertex in a connected graph efficiently

Some time later...
**Exploring a graph**

**TRAVVERSE**($s$)

- put $s$ into the bag
- while the bag is not empty
  - take $u$ from the bag
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u,v)$
      - put $v$ into the bag

**Goal:** visit every vertex in a connected graph efficiently

Some time later...?
Exploring a graph

**TRAVVERSE(s)**

- put \( s \) into the bag
- while the bag is not empty
  - take \( u \) from the bag
  - if \( u \) unmarked
    - mark \( u \)
    - for every edge \((u,v)\)
      - put \( v \) into the bag

**Goal:** visit every vertex in a connected graph efficiently

Questions:

1. *Does TRAVVERSE always stop?*
2. *Does it always visit every vertex?*
3. *How fast is it? (in the worst case)*

Some time later...?
Exploring a graph

**TRAVERSE(s)**

- put $s$ into the bag
- while the bag is not empty
- take $u$ from the bag
- if $u$ unmarked
  - mark $u$
  - for every edge $(u,v)$
  - put $v$ into the bag

**Questions:**

1. *Does TRAVERSE always stop?*
2. *Does it always visit every vertex?*
3. *How fast is it? (in the worst case)*
`TRAVERSE(s)`

put $s$ into the bag
while the bag is not empty
  take $u$ from the bag
  if $u$ unmarked
    mark $u$
    for every edge $(u,v)$
    put $v$ into the bag

1. Does `TRAVERSE` always stop?
Exploring a graph

**TRAVERESE(s)**

- put \( s \) into the **bag**
- while the **bag** is not empty
  - take \( u \) from the **bag**
  - if \( u \) unmarked
    - mark \( u \)
    - for every edge \((u, v)\)
      - put \( v \) into the **bag**

1. Does TRAVERESE *always stop*?
Exploring a graph

**TRAVERSE**($s$)

- put $s$ into the bag
- while the bag is not empty
  - take $u$ from the bag
  - if $u$ unmarked
    - mark $u$
  - for every edge $(u,v)$
    - put $v$ into the bag

1. Does **TRAVERSE** always stop?
**Exploring a graph**

**TRAVVERSE**(s)
put s into the bag
while the bag is not empty
take u from the bag
if u unmarked
    mark u
    for every edge (u, v)
        put v into the bag

1. *Does TRAVVERSE always stop?*

we may put a *vertex* in the *bag* many times but each *edge*
  is only processed twice (once in each direction)
Exploring a graph

TRAVVERSE(s)

put $s$ into the bag
while the bag is not empty
  take $u$ from the bag
  if $u$ unmarked
    mark $u$
    for every edge $(u,v)$
      put $v$ into the bag

1. *Does TRAVVERSE always stop?*

we may put a vertex in the bag many times but each edge is only processed twice (once in each direction)
Exploring a graph

**TRAVVERSE**(s)

- put $s$ into the bag
- while the bag is not empty
  - take $u$ from the bag
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u, v)$
      - put $v$ into the bag

1. **Does TRAVVERSE always stop?**

   - back in!
   - in twice!

   - we may put a vertex in the bag many times but each edge is only processed twice (once in each direction)

   - number of ‘puts’ $\leq 2|E|$
Exploring a graph

\textbf{TRAVVERSE}(s)

put $s$ into the bag
while the bag is not empty
  take $u$ from the bag
  if $u$ unmarked
    mark $u$
    for every edge $(u, v)$
      put $v$ into the bag

1. \textit{Does TRAVVERSE always stop?} 
   \textbf{Yes}

we may put a \textit{vertex} in the bag many times but each \textit{edge} is only processed twice (once in each direction)

\text{number of ‘puts’} \leq 2|E|$
Exploring a graph

**TRAVVERSE**(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u,v)
      put v into the bag

Questions:

1. *Does TRAVVERSE always stop?*
2. *Does it always visit every vertex?*
3. *How fast is it? (in the worst case)*

Yes
**Exploring a graph**

```
TRAVVERSE(s)

put s into the bag
while the bag is not empty
    take u from the bag
    if u unmarked
        mark u
        for every edge (u, v)
            put v into the bag
```

2. Does it always visit every vertex?
**TRVERSE(s)**

put $s$ into the bag
while the bag is not empty
take $u$ from the bag
if $u$ unmarked
    mark $u$
for every edge $(u,v)$
    put $v$ into the bag

2. Does it always visit every vertex?

**Proof (by induction)**

Consider any path from $s$ to some $v$:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

(unless it was already marked)
Exploring a graph

**TRAVVERSE(s)**

- put $s$ into the bag
- while the bag is not empty
- take $u$ from the bag
- if $u$ unmarked
  - mark $u$
  - for every edge $(u, v)$
  - put $v$ into the bag

---

2. *Does it always visit every vertex?*

---

Proof (by induction)

Consider any path from $s$ to some $v$:

When a vertex on this path is marked, the next vertex is put in the bag.

Eventually, that vertex is removed from the bag and is marked.

(Unless it was already marked)
Exploring a graph

**TRAVERSE(s)**

- put $s$ into the **bag**
- while the **bag** is not empty
  - take $u$ from the **bag**
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u, v)$
      - put $v$ into the **bag**

---

2. **Does it always visit every vertex?**

---

**Proof (by induction)**

Consider any path from $s$ to some $v$:

- When a **vertex** on this path is **marked**, the next **vertex** is put in the **bag**
- Eventually, that **vertex** is removed from the **bag** and is **marked**.
  - (unless it was already **marked**).
Exploring a graph

**TRAVERSE**\( (s) \)

- put \( s \) into the bag
- while the bag is not empty
  - take \( u \) from the bag
  - if \( u \) unmarked
    - mark \( u \)
    - for every edge \((u, v)\)
      - put \( v \) into the bag

2. Does it always visit every vertex?

**Proof (by induction)**

Consider any path from \( s \) to some \( v \):
- When a vertex on this path is marked, the next vertex is put in the bag
- Eventually, that vertex is removed from the bag and is marked.
  (unless it was already marked)
Exploring a graph

**TRAVVERSE**($s$)

- put $s$ into the bag
- while the bag is not empty
  - take $u$ from the bag
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u, v)$
      - put $v$ into the bag

2. *Does it always visit every vertex?*

**Proof** (by induction)

Consider any path from $s$ to some $v$:

When a vertex on this path is marked, the next vertex is put in the bag.

Eventually, that vertex is removed from the bag and is marked.

*(unless it was already marked)*
Exploring a graph

**TRAVERSE**(*s*)

- Put *s* into the bag
- While the bag is not empty:
  - Take *u* from the bag
  - If *u* unmarked:
    - Mark *u*
    - For every edge (*u*, *v*):
      - Put *v* into the bag

2. Does it always visit every vertex?

**Proof (by induction)**

Consider any path from *s* to some *v*:

When a vertex on this path is marked, the next vertex is put in the bag.

Eventually, that vertex is removed from the bag and is marked. (unless it was already marked)
Exploring a graph

**Proof (by induction)**

Consider any path from \( s \) to some \( v \):

When a vertex on this path is marked, the next vertex is put in the bag.

Eventually, that vertex is removed from the bag and is marked.

*(unless it was already marked)*
Exploring a graph

**TRAVERTSE(s)**

- put $s$ into the bag
- while the bag is not empty
  - take $u$ from the bag
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u,v)$
      - put $v$ into the bag

2. **Does it always visit every vertex?**

Proof (by induction)

Consider any path from $s$ to some $v$:

- When a vertex on this path is **marked**, the next vertex is put in the **bag**
- Eventually, that vertex is removed from the **bag** and is **marked**.
  - (unless it was already **marked**)
Exploring a graph

**TRAVERSE**(*s*)

- put *s* into the **bag**
- while the **bag** is not empty
  - take *u* from the **bag**
  - if *u* unmarked
    - mark *u*
    - for every edge (*u*, *v*)
    - put *v* into the **bag**

2. Does it always visit every vertex?

**Proof** (by induction)

Consider any path from *s* to some *v*:

- When a vertex on this path is marked, the next vertex is put in the **bag**
- Eventually, that vertex is removed from the **bag** and is marked.
  - *(unless it was already marked)*
Exploring a graph

**Traverse**($s$)

- put $s$ into the bag
- while the bag is not empty
  - take $u$ from the bag
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u,v)$
      - put $v$ into the bag

2. *Does it always visit every vertex?*

---

**Proof** (by induction)

Consider any path from $s$ to some $v$:

- When a vertex on this path is marked, the next vertex is put in the bag
- Eventually, that vertex is removed from the bag and is marked.
  (unless it was already marked)
Exploring a graph

**TRAVERSE(s)**

put $s$ into the bag
while the bag is not empty
  take $u$ from the bag
  if $u$ unmarked
    mark $u$
  for every edge $(u,v)$
    put $v$ into the bag

2. Does it always visit every vertex?

Consider any path from $s$ to some $v$:

When a vertex on this path is marked, the next vertex is put in the bag
Eventually, that vertex is removed from the bag and is marked.
(unless it was already marked)
Exploring a graph

**TRVERSE(s)**

- put $s$ into the **bag**
- while the **bag** is not empty
  - take $u$ from the **bag**
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u, v)$
      - put $v$ into the **bag**

2. **Does it always visit every vertex?**

**Proof (by induction)**

Consider any path from $s$ to some $v$:
- When a **vertex** on this path is **marked**, the next **vertex** is put in the **bag**
- Eventually, that **vertex** is removed from the **bag** and is **marked**.
  - (unless it was already **marked**)

**BAG**
Exploring a graph

TRAVVERSE(s)

put $s$ into the bag
while the bag is not empty
  take $u$ from the bag
  if $u$ unmarked
    mark $u$
    for every edge $(u, v)$
      put $v$ into the bag

2. Does it always visit every vertex?

Proof (by induction)

Consider any path from $s$ to some $v$:

When a vertex on this path is marked, the next vertex is put in the bag
Eventually, that vertex is removed from the bag and is marked.
(unless it was already marked)
Exploring a graph

**TRAVERSE(\(s\))**

- put \(s\) into the bag
- while the bag is not empty
  - take \(u\) from the bag
  - if \(u\) unmarked
    - mark \(u\)
    - for every edge \((u, v)\)
      - put \(v\) into the bag

2. **Does it always visit every vertex?**

---

**Proof (by induction)**

Consider any path from \(s\) to some \(v\):

- When a vertex on this path is marked, the next vertex is put in the bag
- Eventually, that vertex is removed from the bag and is marked.
  (unless it was already marked)
Exploring a graph

**TRaverse**(*s*)

- put *s* into the bag
- while the bag is not empty
  - take *u* from the bag
  - if *u* unmarked
    - mark *u*
    - for every edge *(u, v)*
      - put *v* into the bag

2. *Does it always visit every vertex?*

Proof (by induction)

Consider any path from *s* to some *v*:

- When a vertex on this path is marked, the next vertex is put in the bag.
- Eventually, that vertex is removed from the bag and is marked.
  (unless it was already marked)
**Exploring a graph**

**TRVERSE(s)**

- put $s$ into the **bag**
- while the **bag** is not empty
  - take $u$ from the **bag**
  - if $u$ unmarked
    - **mark** $u$
    - for every edge $(u,v)$
      - put $v$ into the **bag**

2. *Does it always visit every vertex?*

---

**Proof (by induction)**

Consider **any path from** $s$ **to some** $v$:

- When a **vertex** on this path is **marked**, the next **vertex** is put in the **bag**
- Eventually, that **vertex** is removed from the **bag** and is **marked**.
  (unless it was already **marked**)
**Exploring a graph**

**TRAVERSE(s)**

- put $s$ into the bag
- while the bag is not empty
  - take $u$ from the bag
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u,v)$
      - put $v$ into the bag

2. Does it always visit every vertex?

**Proof (by induction)**

Consider any path from $s$ to some $v$:

- When a vertex on this path is marked, the next vertex is put in the bag
- Eventually, that vertex is removed from the bag and is marked.
  (unless it was already marked)
Exploring a graph

**TRVERSE(s)**

put *s* into the bag
while the bag is not empty
  take *u* from the bag
  if *u* unmarked
    mark *u*
    for every edge \((u, v)\)
      put *v* into the bag

2. Does it always visit every vertex?

**Proof (by induction)**

Consider any path from *s* to some *v*:

- When a vertex on this path is marked, the next vertex is put in the bag
- Eventually, that vertex is removed from the bag and is marked.
  (unless it was already marked)
Exploring a graph

**TRAVERSER**(s)

- put *s* into the **bag**
- while the **bag** is not empty
  - take *u* from the **bag**
  - if *u* unmarked
    - mark *u*
    - for every edge *(u,v)*
      - put *v* into the **bag**

2. Does it always visit every vertex?

**Proof** (by induction)

Consider any path from *s* to some *v*:

- When a **vertex** on this path is **marked**, the next **vertex** is put in the **bag**
- Eventually, that **vertex** is removed from the **bag** and is **marked**.
  (unless it was already **marked**)
Exploring a graph

**TRaverse(s)**

- put $s$ into the bag
- while the bag is not empty
  - take $u$ from the bag
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u,v)$
      - put $v$ into the bag

2. **Does it always visit every vertex?**

**Proof (by induction)**

Consider any path from $s$ to some $v$:

- When a vertex on this path is marked, the next vertex is put in the bag
- Eventually, that vertex is removed from the bag and is marked.
  (unless it was already marked)
Exploring a graph

**TRAVVERSE**(s)

put \( s \) into the bag
define the bag is not empty

take \( u \) from the bag
if \( u \) unmarked
    mark \( u \)
    for every edge \((u,v)\)
        put \( v \) into the bag

2. Does it always visit every vertex?

**Proof (by induction)**

Consider any path from \( s \) to some \( v \):

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

(\textit{unless it was already marked})
Exploring a graph

**TRAVERSE**(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u,v)
      put v into the bag

---

2. Does it always visit every vertex?

---

**Proof** (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

(unless it was already marked)
Exploring a graph

TRAVERTSE(s)

put $s$ into the bag
while the bag is not empty
  take $u$ from the bag
if $u$ unmarked
  mark $u$
  for every edge $(u,v)$
    put $v$ into the bag

2. Does it always visit every vertex?

---

Proof (by induction)

Consider any path from $s$ to some $v$:

When a vertex on this path is marked, the next vertex is put in the bag
Eventually, that vertex is removed from the bag and is marked.
(unless it was already marked)
Exploring a graph

**TRAVERESE**($s$)

- put $s$ into the bag
- while the bag is not empty
  - take $u$ from the bag
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u,v)$
      - put $v$ into the bag

2. *Does it always visit every vertex?* Yes

---

**Proof** (by induction)

Consider any path from $s$ to some $v$:

- When a vertex on this path is marked, the next vertex is put in the bag
- Eventually, that vertex is removed from the bag and is marked.
  
  *(unless it was already marked)*
Exploring a graph

**TRAVERESE(s)**

put $s$ into the bag
while the bag is not empty
    take $u$ from the bag
    if $u$ unmarked
        mark $u$
        for every edge $(u,v)$
            put $v$ into the bag

2. **Does it always visit every vertex?**

Yes

the correctness doesn’t depend on how the bag works!

**Proof (by induction)**

Consider any path from $s$ to some $v$:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

(Unless it was already marked)
Exploring a graph

**TRAVERSE\(s\)**

put \(s\) into the bag
while the bag is not empty
  take \(u\) from the bag
  if \(u\) unmarked
    mark \(u\)
    for every edge \((u,v)\)
      put \(v\) into the bag

2. Does it always visit every vertex? Yes

the correctness doesn’t depend on how the bag works!

(but this doesn’t tell you anything about which order vertices are marked in)

**Proof** (by induction)

Consider any path from \(s\) to some \(v\):

When a vertex on this path is marked, the next vertex is put in the bag
Eventually, that vertex is removed from the bag and is marked.
(Unless it was already marked)
Exploring a graph

Assumption

bag operations take $O(1)$ time

(we’ll come back to this)

Questions:

1. *Does TRAVERSE always stop?* Yes
2. *Does it always visit every vertex?* Yes
3. *How fast is it? (in the worst case)*
Exploring a graph

Assumption

- bag operations take $O(1)$ time
  (we’ll come back to this)

Questions:

1. Does TRAVERSE always stop?  
   Yes
2. Does it always visit every vertex?  
   Yes
3. How fast is it? (in the worst case)

\textbf{TRAVERSE}(s)

- put $s$ into the bag
- while the bag is not empty
  - take $u$ from the bag
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u,v)$
      - put $v$ into the bag
Exploring a graph

Questions:

1. Does TRAVERSE always stop?  
   Yes

2. Does it always visit every vertex? 
   Yes

3. How fast is it? (in the worst case)

Time complexity:

**TRAVERSE**(s)

put *s* into the **bag**
while the **bag** is not empty
   take *u* from the **bag**
   if *u* unmarked
      mark *u*
      for every edge (*u*, *v*)
         put *v* into the **bag**
Exploring a graph

Assumption

- Bag operations take $O(1)$ time

(we’ll come back to this)

Questions:

1. Does TRAVERSE always stop? Yes
2. Does it always visit every vertex? Yes
3. How fast is it? (in the worst case)

Time complexity:

$O(1)$ time every time we take from the bag

TRAVERSE(s)

put $s$ into the bag
while the bag is not empty
  take $u$ from the bag
  if $u$ unmarked
    mark $u$
    for every edge $(u,v)$
      put $v$ into the bag
Exploring a graph

Questions:

1. Does Traverse always stop?  [Yes]
2. Does it always visit every vertex?  [Yes]
3. How fast is it? (in the worst case)

Assumption

- bag operations take $O(1)$ time
- (we’ll come back to this)

**TRAVERSE**(s)

- put $s$ into the bag
- while the bag is not empty
  - take $u$ from the bag
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u,v)$
      - put $v$ into the bag

Time complexity:

$O(1)$ time every time we take from the bag

(store an array where $mark[u] = 1$ iff $u$ is marked)
Exploring a graph

Assumption

- bag operations take $O(1)$ time
- (we'll come back to this)

Questions:

1. Does `TRAVVERSE` always stop? **Yes**
2. Does it always visit every vertex? **Yes**
3. How fast is it? (in the worst case)

Time complexity:

- $O(1)$ time every time we take from the bag
- (store an array where $\text{mark}[u] = 1$ iff $u$ is marked)
- $O(1)$ time every time we put into the bag

```
TRAVVERSE(s)
```

- put $s$ into the bag
- while the bag is not empty
  - take $u$ from the bag
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u, v)$
      - put $v$ into the bag
Exploring a graph

Questions:

1. Does Traverse always stop?  Yes
2. Does it always visit every vertex?  Yes
3. How fast is it? (in the worst case)

Assumption

- bag operations take $O(1)$ time
  (we’ll come back to this)

Time complexity:

- $O(1)$ time every time we take from the bag
  (store an array where $mark[u] = 1$ iff $u$ is marked)
- $O(1)$ time every time we put into the bag

What is the total number of bag operations?

**Traverse($s$)**

- put $s$ into the bag
- while the bag is not empty
  - take $u$ from the bag
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u,v)$
      - put $v$ into the bag
Exploring a graph

Questions:
1. Does TRAVERSE always stop?
   Yes
2. Does it always visit every vertex?
   Yes
3. How fast is it? (in the worst case)

Time complexity:
- $O(1)$ time every time we take from the bag
  (store an array where $\text{mark}[u] = 1$ iff $u$ is marked)
- $O(1)$ time every time we put into the bag

What is the total number of bag operations? We do at most $2|E|$ put operations
Exploring a graph

Questions:
1. Does TRAVERSE always stop? Yes
2. Does it always visit every vertex? Yes
3. How fast is it? (in the worst case)

Assumption

bag operations take $O(1)$ time

(we’ll come back to this)

Time complexity:

$O(1)$ time every time we take from the bag

(store an array where $mark[u] = 1$ iff $u$ is marked)

$O(1)$ time every time we put into the bag

What is the total number of bag operations?

we do at most $2|E|$ put operations

so we do at most $2|E|$ take operations

TRAVERSE(s)

put $s$ into the bag
while the bag is not empty
take $u$ from the bag
if $u$ unmarked
mark $u$
for every edge $(u, v)$
put $v$ into the bag
Exploring a graph

Assumption

- bag operations take \( O(1) \) time

(we’ll come back to this)

Questions:

1. Does \textsc{Traverse} always stop? \( \text{Yes} \)
2. Does it always visit every vertex? \( \text{Yes} \)
3. How fast is it? (in the worst case)

Time complexity:

\textsc{Traverse}(s)

\begin{itemize}
  \item put \( s \) into the bag
  \item while the bag is not empty
    \begin{itemize}
      \item take \( u \) from the bag
      \item if \( u \) unmarked
        \begin{itemize}
          \item mark \( u \)
          \item for every edge \((u, v)\)
            \begin{itemize}
              \item put \( v \) into the bag
            \end{itemize}
        \end{itemize}
    \end{itemize}
\end{itemize}

\( O(1) \) time every time we take from the bag

\( O(1) \) time every time we put into the bag

\( O(1) \) time every time we take from the bag

(\textit{store an array where mark}[u] = 1 \textit{iff } u \textit{ is marked})

What is the total number of bag operations?

we do at most \( 2|E| \) put operations

so we do at most \( 2|E| \) take operations

The overall time complexity is \( O(|E|) \)
Exploring a graph

Questions:

1. Does Traverse always stop?  
   Yes

2. Does it always visit every vertex?  
   Yes

3. How fast is it? (in the worst case)  
   \( O(|E|) \)

Assumption

- bag operations take \( O(1) \) time
- (we’ll come back to this)

\( \text{Traverse}(s) \)

- put \( s \) into the bag
- while the bag is not empty
  - take \( u \) from the bag
  - if \( u \) unmarked
    - mark \( u \)
    - for every edge \( (u, v) \)
      - put \( v \) into the bag

Time complexity:

- \( O(1) \) time every time we take from the bag
  - (store an array where \( mark[u] = 1 \) iff \( u \) is marked)
- \( O(1) \) time every time we put into the bag
- What is the total number of bag operations?
  - we do at most \( 2|E| \) put operations
  - so we do at most \( 2|E| \) take operations

The overall time complexity is \( O(|E|) \)
Exploring a graph

Questions:

1. Does \textsc{Traverse} always stop? \hspace{2cm} Yes
2. Does it always visit every vertex? \hspace{2cm} Yes
3. How fast is it? (in the worst case) \hspace{2cm} \( O(|E|) \)

Assumption

- bag operations take \( O(1) \) time

(we’ll come back to this)

\textsc{Traverse}(s)

- put \( s \) into the bag
- while the bag is not empty
  - take \( u \) from the bag
  - if \( u \) unmarked
    - mark \( u \)
    - for every edge \((u,v)\)
      - put \( v \) into the bag

Assumption (we’ll come back to this)

- bag operations take \( O(1) \) time

(we’ll come back to this)
Exploring a graph

Questions:

1. **Does TRAVERSE always stop?**
   - Yes

2. **Does it always visit every vertex?**
   - Yes

3. **How fast is it? (in the worst case)**
   - $O(|E|)$

**Assumption**
- bag operations take $O(1)$ time
- (we’ll come back to this)

**TRAVERSE($s$)**

- put $s$ into the bag
- while the bag is not empty
  - take $u$ from the bag
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u, v)$
      - put $v$ into the bag

What if we had used an adjacency matrix?

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</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>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Exploring a graph

Questions:

1. Does \textsc{Traverse} always stop? \hspace{1cm} \textbf{Yes}
2. Does it always visit every vertex? \hspace{1cm} \textbf{Yes}
3. How fast is it? (in the worst case) \hspace{1cm} O(|E|)

\textbf{Assumption}

- bag operations take $O(1)$ time
- \textit{(we'll come back to this)}

\textbf{\textsc{Traverse}(s)}

- put $s$ into the bag
- while the bag is not empty
  - take $u$ from the bag
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u, v)$
      - put $v$ into the bag

What if we had used an adjacency matrix?

Finding all edges leaving $u$ would have taken $O(|V|)$ time.
Exploring a graph

Questions:

1. Does TRAVERSE always stop?  
   Yes

2. Does it always visit every vertex?  
   Yes

3. How fast is it? (in the worst case)  
   $O(|E|)$

Assumption

- bag operations take $O(1)$ time
- (we’ll come back to this)

TRAVERSE($s$)

- put $s$ into the bag
- while the bag is not empty
  - take $u$ from the bag
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u, v)$
      - put $v$ into the bag

What if we had used an adjacency matrix?

- finding all edges leaving $u$ would have taken $O(|V|)$ time

overall, the time complexity would have been $O(|V|^2)$ instead of $O(|E|)$. 
How should we implement the bag?

Queue

Stack

Operations take $O(1)$ time

**TRAVERSE** visits every vertex in a connected graph in $O(|E|)$ time

*but in different orders with different types of bag*

with a Queue the algorithm is called

**Breadth First Search**

Applications

**Shortest paths in unweighted graphs**

Max-Flow

Testing whether a graph is bipartite

with a Stack the algorithm is called

**Depth First Search**

Applications

Finding (strongly) connected components

Topicologically sorting a Directed Acyclic Graph

Testing for planarity
How should we implement the bag?

Queue

Stack

Operations take $O(1)$ time

TRAVIS-visits every vertex in a connected graph in $O(|E|)$ time

but in different orders with different types of bag

(and it works for directed graphs too)

with a Queue the algorithm is called Breadth First Search

with a Stack the algorithm is called Depth First Search

Applications

- Shortest paths in unweighted graphs
- Max-Flow
- Testing whether a graph is bipartite

Applications

- Finding (strongly) connected components
- Topologically sorting a Directed Acyclic Graph
- Testing for planarity
Shortest paths in unweighted graphs

**Goal**: find the distance from the source $s$ to every other vertex.
Shortest paths in unweighted graphs

**Goal**: find the distance from the source \( s \) to every other vertex

\[ \text{distance} = 1 \]
Goal: find the distance from the source $s$ to every other vertex
Shortest paths in unweighted graphs

Goal: find the distance from the source s to every other vertex

distance = 6
Goal: find the distance from the source $s$ to every other vertex
Shortest paths in unweighted graphs

Goal: **find the distance from the source** *s* **to every other vertex**

**Breadth-First Search (BFS)** is **TRAVVERSE** using a *queue* as the bag

**TRAVVERSE**(*s*) - with a *queue*

- put *s* into the *queue*
- while the *queue* is not empty
  - take *u* from the *queue*
  - if *u* unmarked
    - mark *u*
    - for every edge (*u*, *v*)
      - put *v* into the *queue*
**Goal**: find the distance from the source $s$ to every other vertex

**Lemma**: In BFS, the vertices are marked in distance order (smallest first)

**Breadth-First Search (BFS)** is TRAVERSE using a queue as the bag

**TRAVERSE($s$) - with a queue**

- put $s$ into the queue
- while the queue is not empty
  - take $u$ from the queue
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u,v)$
      - put $v$ into the queue
Goal: find the distance from the source $s$ to every other vertex

Breadth-First Search (BFS) is $\text{TRAVERSE}$ using a queue as the bag

Lemma In BFS, the vertices are marked in distance order (smallest first)

$\text{TRAVERSE}(s)$ - with a queue

put $s$ into the queue
while the queue is not empty
  take $u$ from the queue
  if $u$ unmarked
    mark $u$
    for every edge $(u,v)$
      put $v$ into the queue

Shortest paths in unweighted graphs
Shortest paths in unweighted graphs

**Goal:** find the distance from the source \( s \) to every other vertex

**Breadth-First Search (BFS)** is \( \text{TRAVERSE} \) using a queue as the bag

**Lemma** In BFS, the vertices are marked in distance order (smallest first)

\[
\text{TRAVERSE}(s) - \text{with a queue}
\]

- put \( s \) into the queue
- while the queue is not empty
  - take \( u \) from the queue
  - if \( u \) unmarked
    - mark \( u \)
    - for every edge \((u,v)\)
      - put \( v \) into the queue
Goal: find the distance from the source \( s \) to every other vertex

**Breadth-First Search (BFS)**

is Traverse using a queue as the bag

**Lemma** In BFS, the vertices are marked in distance order (smallest first)
Shortest paths in unweighted graphs

**Goal:** find the distance from the source \( s \) to every other vertex

**Lemma** In BFS, the vertices are marked in distance order (smallest first)

**Breadth-First Search (BFS)** is \( \text{TRAVVERSE} \) using a queue as the bag

\[
\begin{align*}
\text{TRAVVERSE}(s) & \text{- with a queue} \\
\text{put } s \text{ into the queue} \\
\text{while the queue is not empty} \\
\text{take } u \text{ from the queue} \\
\text{if } u \text{ unmarked} \\
\text{mark } u \\
\text{for every edge } (u,v) \\
\text{put } v \text{ into the queue}
\end{align*}
\]
Shortest paths in unweighted graphs

**Goal:** find the distance from the source $s$ to every other vertex

**Breadth-First Search (BFS)** is TRAVERSE using a queue as the bag.

**Lemma** In BFS, the vertices are marked in distance order (smallest first).

**TRAVERSE($s$) - with a queue**

- put $s$ into the queue
- while the queue is not empty
  - take $u$ from the queue
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u,v)$
      - put $v$ into the queue

**Diagram**

- Node $s$ is the source.
- Nodes are connected by edges.
- The diagram illustrates the breadth-first traversal starting from $s$.
Shortest paths in unweighted graphs

**Goal:** find the distance from the source $s$ to every other vertex

**Breadth-First Search (BFS)** is $\text{TRAVERSE}$ using a queue as the bag

**Lemma** In BFS, the vertices are marked in distance order (smallest first)

$\text{TRAVERSE}(s)$ - with a queue

- put $s$ into the queue
- while the queue is not empty
  - take $u$ from the queue
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u, v)$
      - put $v$ into the queue
Shortest paths in unweighted graphs

**Goal:** find the distance from the source \( s \) to every other vertex

**Breadth-First Search (BFS)**

is TRAVERSE using a queue as the bag

**TRAVERSE\((s)\) - with a queue**

- put \( s \) into the queue
- while the queue is not empty
- take \( u \) from the queue
- if \( u \) unmarked
  - mark \( u \)
  - for every edge \((u,v)\)
    - put \( v \) into the queue

**Lemma** In BFS, the vertices are marked in distance order (smallest first)
Shortest paths in unweighted graphs

**Goal**: find the distance from the source $s$ to every other vertex

**Breadth-First Search (BFS)**

is **TRAVERSE** using a *queue* as the bag

**Lemma** In BFS, the vertices are marked in distance order (smallest first)

**TRAVERSE**($s$) - with a *queue*

- put $s$ into the *queue*
- while the *queue* is not empty
  - take $u$ from the *queue*
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u, v)$
      - put $v$ into the *queue*
Shortest paths in unweighted graphs

**Goal:** find the distance from the source $s$ to every other vertex

**Breadth-First Search (BFS)** is Traverse using a queue as the bag

**Lemma** In BFS, the vertices are marked in distance order (smallest first)

**TRAVERSE($s$) - with a queue**

- put $s$ into the queue
- while the queue is not empty
  - take $u$ from the queue
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u, v)$
      - put $v$ into the queue
Shortest paths in unweighted graphs

**Goal**: find the distance from the source \( s \) to every other vertex

**Breadth-First Search (BFS)** is \( \text{TRAVVERSE} \) using a queue as the bag

**Lemma** In BFS, the vertices are marked in distance order (smallest first)

**\( \text{TRAVVERSE}(s) \) - with a queue**

- put \( s \) into the queue
- while the queue is not empty
  - take \( u \) from the queue
  - if \( u \) unmarked
    - mark \( u \)
    - for every edge \( (u, v) \)
      - put \( v \) into the queue
Shortest paths in unweighted graphs

**Goal**: find the distance from the source $s$ to every other vertex

**Breadth-First Search (BFS)** is `TRAVERSE` using a queue as the bag.

**Lemma** In BFS, the vertices are marked in distance order (smallest first).

`TRAVERSE(s) - with a queue`

- put $s$ into the queue
- while the queue is not empty
  - take $u$ from the queue
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u, v)$
      - put $v$ into the queue
Shortest paths in unweighted graphs

**Goal**: find the distance from the source \( s \) to every other vertex

**Breadth-First Search (BFS)** is \( \text{TRAVERSE} \) using a queue as the bag

**Lemma** In BFS, the vertices are marked in distance order (smallest first)

\[
\text{TRAVERSE}(s) - \text{with a queue}
\]

put \( s \) into the queue
while the queue is not empty
  take \( u \) from the queue
  if \( u \) unmarked
    mark \( u \)
    for every edge \((u, v)\)
      put \( v \) into the queue
Shortest paths in unweighted graphs

**Goal:** find the distance from the source $s$ to every other vertex

**Breadth-First Search (BFS)**

is $\text{TRAVERSE}$ using a queue as the bag

**Lemma** In BFS, the vertices are marked in distance order (smallest first)

**$\text{TRAVERSE}(s)$ - with a queue**

put $s$ into the queue
while the queue is not empty
take $u$ from the queue
if $u$ unmarked
mark $u$
for every edge $(u, v)$
put $v$ into the queue
Shortest paths in unweighted graphs

**Goal**: find the distance from the source \( s \) to every other vertex

**Breadth-First Search (BFS)** is TRAVERSE using a queue as the bag

**Lemma** In BFS, the vertices are marked in distance order (smallest first)

**TRAVERSE** \((s)\) - with a queue

- put \( s \) into the queue
- while the queue is not empty
- take \( u \) from the queue
- if \( u \) unmarked
  - mark \( u \)
  - for every edge \((u, v)\)
    - put \( v \) into the queue
Goal: find the distance from the source $s$ to every other vertex.

Breadth-First Search (BFS) is $\text{TRAVERSE}$ using a queue as the bag.

Lemma: In BFS, the vertices are marked in distance order (smallest first).

$\text{TRAVERSE}(s)$ - with a queue

put $s$ into the queue
while the queue is not empty
  take $u$ from the queue
  if $u$ unmarked
    mark $u$
    for every edge $(u,v)$
      put $v$ into the queue
**Goal:** find the distance from the source $s$ to every other vertex

**Breadth-First Search (BFS) is TRAVERSE using a queue as the bag**

**Lemma** In BFS, the vertices are marked in distance order (smallest first)

**TRAVERSE($s$) - with a queue**

- put $s$ into the queue
- while the queue is not empty
  - take $u$ from the queue
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u,v)$
      - put $v$ into the queue
Goal: find the distance from the source $s$ to every other vertex

Breadth-First Search (BFS) is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices are marked in distance order (smallest first)

TRAVERSE($s$) - with a queue

1. put $s$ into the queue
2. while the queue is not empty
   - take $u$ from the queue
   - if $u$ unmarked
     - mark $u$
     - for every edge $(u,v)$
       - put $v$ into the queue
Shortest paths in unweighted graphs

**Goal**: find the distance from the source $s$ to every other vertex

**Breadth-First Search (BFS)**

is TRAVERSE using a queue as the bag

**Lemma** In BFS, the vertices are marked in distance order (smallest first)

This is why it’s called *Breadth-First Search*

**TRAVERSE($s$) - with a queue**

put $s$ into the queue

while the queue is not empty

take $u$ from the queue

if $u$ unmarked

mark $u$

for every edge $(u, v)$

put $v$ into the queue
Shortest paths in unweighted graphs

Lemma In BFS, the vertices are marked in distance order (smallest first)

Proof by induction

**Base Case**: BFS trivially marks all vertices at distance 0 before marking any vertex at distance > 0.

**Inductive Hypthesis**: Assume that BFS marks all vertices at distance \((i - 1)\) before marking any vertex at distance > \((i - 1)\).

When BFS marks a vertex at distance \((i - 1)\), it puts all its neighbours in the queue.

Immediately after marking all vertices at distance \((i - 1)\):
- every vertex at distance \(i\) is in the queue.
- no vertex at distance > \(i\) has been put in the queue. 

\(v\) is \(u\)'s neighbour iff \((u, v) \in E\)

Because we haven’t marked any of its neighbours (they are all at distance > \((i - 1)\))

Therefore, all vertices at distance \(i\) will be marked before any at distance > \(i\)
Shortest paths in unweighted graphs

**Goal:** find the distance from the source \( s \) to every other vertex

**Breadth-First Search (BFS)** is \textsc{Traverse} using a queue as the bag.

**Lemma** In BFS, the vertices are marked in distance order (smallest first).

With a little bit of *book-keeping*, we can use this to find shortest paths in unweighted graphs in \( O(|V| + |E|) \) time.

\[
\text{Traverse}(s) - \text{with a queue}
\]

- put \( s \) into the queue
- while the queue is not empty
  - take \( u \) from the queue
  - if \( u \) unmarked
    - mark \( u \)
    - for every edge \((u, v)\)
      - put \( v \) into the queue
Shortest paths in unweighted graphs

**Goal:** find the distance from the source $s$ to every other vertex

Breadth-First Search (BFS) is $\text{TRAVERSE}$ using a queue as the bag.

**Lemma** In BFS, the vertices are marked in distance order (smallest first).

With a little bit of *book-keeping*, we can use this to find shortest paths in unweighted graphs in $O(|V| + |E|)$ time.

This also works for directed, unweighted graphs.
Shortest Paths using BFS

We’ve added four (NEW!) lines to TRAVERSE to track the distances from \( s \)

\[
\text{BFS}(s)
\]

for all \( v \), set \( \text{dist}(v) = \infty \) (NEW!)
set \( \text{dist}(s) = 0 \) (NEW!)
put \( s \) into the queue
while the queue is not empty
  take \( u \) from the queue
  if \( u \) unmarked
    mark \( u \)
    for every edge \( (u,v) \)
      put \( v \) into the queue
      if \( \text{dist}(v) = \infty \) (NEW!)
        \( \text{dist}(v) = \text{dist}(u) + 1 \) (NEW!)

\( \text{dist}(v) \) gives the distance between \( s \) and \( v \)
Shortest Paths using BFS

We’ve added four (NEW!) lines to TRAVERSE to track the distances from \( s \)

\[
\text{BFS}(s)
\]

for all \( v \), set \( \text{dist}(v) = \infty \) (NEW!)
set \( \text{dist}(s) = 0 \) (NEW!)
put \( s \) into the queue
while the queue is not empty
    take \( u \) from the queue
    if \( u \) unmarked
        mark \( u \)
        for every edge \( (u, v) \)
            put \( v \) into the queue
            if \( \text{dist}(v) = \infty \) (NEW!)
                \( \text{dist}(v) = \text{dist}(u) + 1 \) (NEW!)

\( \text{dist}(v) \) gives the distance between \( s \) and \( v \)

How does this affect the time complexity?
Shortest Paths using BFS

We’ve added four (NEW!) lines to TRAVERSE to track the distances from $s$

```plaintext
BFS($s$)
for all $v$, set $\text{dist}(v) = \infty$ (NEW!)
set $\text{dist}(s) = 0$ (NEW!)
put $s$ into the queue
while the queue is not empty
  take $u$ from the queue
  if $u$ unmarked
    mark $u$
    for every edge $(u,v)$
      put $v$ into the queue
      if $\text{dist}(v) = \infty$ (NEW!)
        $\text{dist}(v) = \text{dist}(u) + 1$ (NEW!)
```

$\text{dist}(v)$ gives the distance between $s$ and $v$

How does this affect the time complexity?

Time complexity:
Shortest Paths using BFS

We’ve added four (NEW!) lines to TRAVERSE to track the distances from $s$.

\[
\text{BFS}(s)
\]

for all $v$, set $\text{dist}(v) = \infty$ (NEW!)
set $\text{dist}(s) = 0$ (NEW!)
put $s$ into the queue
while the queue is not empty
  take $u$ from the queue
  if $u$ unmarked
    mark $u$
    for every edge $(u,v)$
      put $v$ into the queue
      if $\text{dist}(v) = \infty$ (NEW!)
        $\text{dist}(v) = \text{dist}(u) + 1$ (NEW!)

$\text{dist}(v)$ gives the distance between $s$ and $v$

How does this affect the time complexity?

**Time complexity:**

$O(|V|)$ time to initialise an array $\text{dist}$
Shortest Paths using BFS

We’ve added four \textit{(NEW!)} lines to \textsc{TRaverse} to track the distances from $s$.

\begin{tcolorbox}
\textbf{BFS}(s)

for all $v$, set $\text{dist}(v) = \infty$ (NEW!)
set $\text{dist}(s) = 0$ (NEW!)
put $s$ into the queue
while the queue is not empty
  take $u$ from the queue
  if $u$ unmarked
    mark $u$
    for every edge $(u,v)$
      put $v$ into the queue
      if $\text{dist}(v) = \infty$ (NEW!)
        $\text{dist}(v) = \text{dist}(u) + 1$ (NEW!)
\end{tcolorbox}

\textbf{Time complexity:}

- $O(|V|)$ time to initialise an array $\text{dist}$
- $O(1)$ time to set the distance to $s$

How does this affect the time complexity?

$\text{dist}(v)$ gives the distance between $s$ and $v$
Shortest Paths using BFS

We’ve added four (NEW!) lines to TRAVERSE to track the distances from \( s \)

\[
\text{BFS}(s)
\]

for all \( v \), set \( \text{dist}(v) = \infty \) (NEW!)
set \( \text{dist}(s) = 0 \) (NEW!)
put \( s \) into the queue
while the queue is not empty
    take \( u \) from the queue
    if \( u \) unmarked
        mark \( u \)
        for every edge \((u,v)\)
            put \( v \) into the queue
            if \( \text{dist}(v) = \infty \) (NEW!)
                \( \text{dist}(v) = \text{dist}(u) + 1 \) (NEW!)

\( \text{dist}(v) \) gives the distance between \( s \) and \( v \)

How does this affect the time complexity?

Time complexity:

- \( O(|V|) \) time to initialise an array \( \text{dist} \)
- \( O(1) \) time to set the distance to \( s \)
- \( O(1) \) time whenever we put into the queue (so this doesn't change the complexity)
Shortest Paths using BFS

We’ve added four (NEW!) lines to TRAVERSE to track the distances from $s$

BFS($s$)

for all $v$, set $\text{dist}(v) = \infty$ (NEW!)
set $\text{dist}(s) = 0$ (NEW!)
put $s$ into the queue
while the queue is not empty
    take $u$ from the queue
    if $u$ unmarked
        mark $u$
        for every edge $(u,v)$
            put $v$ into the queue
            if $\text{dist}(v) = \infty$ (NEW!)
                $\text{dist}(v) = \text{dist}(u) + 1$ (NEW!)

$\text{dist}(v)$ gives the distance between $s$ and $v$

How does this affect the time complexity?

**Time complexity:**

$O(|V|)$ time to initialise an array $\text{dist}$

$O(1)$ time to set the distance to $s$

$O(1)$ time whenever we put into the queue (so this doesn’t change the complexity)

so the overall complexity is $O(|V| + |E|)$
Shortest Paths using BFS

We’ve added four (NEW!) lines to TRAVERSE to track the distances from \( s \)

\[
\text{BFS}(s)
\]

for all \( v \), set \( \text{dist}(v) = \infty \) (NEW!)  
set \( \text{dist}(s) = 0 \) (NEW!)  
put \( s \) into the queue  
while the queue is not empty  
\hspace{1em} \text{take } u \text{ from the queue}  
\hspace{1em} \text{if } u \text{ unmarked}  
\hspace{2em} \text{mark } u  
\hspace{1em} \text{for every edge } (u,v)  
\hspace{2em} \text{put } v \text{ into the queue}  
\hspace{2em} \text{if } \text{dist}(v) = \infty \) (NEW!)  
\hspace{3em} \text{dist}(v) = \text{dist}(u) + 1 \) (NEW!)

\( \text{dist}(v) \) gives the distance  
between \( s \) and \( v \)
Shortest Paths using BFS

We’ve added four (NEW!) lines to TRAVERSE to track the distances from $s$

$$\text{BFS}(s)$$

for all $v$, set $\text{dist}(v) = \infty$ (NEW!)
set $\text{dist}(s) = 0$ (NEW!)
put $s$ into the queue
while the queue is not empty
  take $u$ from the queue
  if $u$ unmarked
    mark $u$
    for every edge $(u, v)$
      put $v$ into the queue
      if $\text{dist}(v) = \infty$ (NEW!)
        $\text{dist}(v) = \text{dist}(u) + 1$ (NEW!)

$\text{dist}(v)$ gives the distance between $s$ and $v$
Shortest Paths using BFS

We’ve added four (NEW!) lines to TRAVERSE to track the distances from $s$

BFS($s$)

for all $v$, set $\text{dist}(v) = \infty$ (NEW!)
set $\text{dist}(s) = 0$ (NEW!)
push $s$ into the queue
while the queue is not empty
  take $u$ from the queue
  if $u$ unmarked
    mark $u$
    for every edge $(u,v)$
      put $v$ into the queue
      if $\text{dist}(v) = \infty$ (NEW!)
        $\text{dist}(v) = \text{dist}(u) + 1$ (NEW!)

$\text{dist}(v)$ gives the distance between $s$ and $v$

Why does this work?

Lemma In BFS, the vertices are marked in distance order (smallest first)
Shortest Paths using BFS

We’ve added four (NEW!) lines to TRAVERSE to track the distances from $s$

```
BFS(s)
```

for all $v$, set $\text{dist}(v) = \infty$ (NEW!)
set $\text{dist}(s) = 0$ (NEW!)
put $s$ into the queue
while the queue is not empty
    take $u$ from the queue
    if $u$ unmarked
        mark $u$
        for every edge $(u,v)$
            put $v$ into the queue
            if $\text{dist}(v) = \infty$ (NEW!)
                $\text{dist}(v) = \text{dist}(u) + 1$ (NEW!)

$\text{dist}(v)$ gives the distance between $s$ and $v$

Why does this work?

**Lemma** In BFS, the vertices are marked in distance order (smallest first)

**Correctness sketch:** (using the Lemma)
Shortest Paths using BFS

We’ve added four (NEW!) lines to TRAVERSE to track the distances from $s$

**BFS($s$)**

for all $v$, set $\text{dist}(v) = \infty$ (NEW!)  
set $\text{dist}(s) = 0$ (NEW!)  
put $s$ into the queue  
while the queue is not empty  
  take $u$ from the queue  
  if $u$ unmarked  
    mark $u$  
    for every edge $(u,v)$  
      put $v$ into the queue  
      if $\text{dist}(v) = \infty$ (NEW!)  
        $\text{dist}(v) = \text{dist}(u) + 1$ (NEW!)

$\text{dist}(v)$ gives the distance between $s$ and $v$

Why does this work?

**Lemma** In BFS, the vertices are marked in distance order (smallest first)

**Correctness sketch:** (using the Lemma)

For each $v$, $\text{dist}(v)$ only changes once, when it is first discovered and inserted into the queue.
Shortest Paths using BFS

We’ve added four (NEW!) lines to TRAVERSE to track the distances from $s$

\[
\text{BFS}(s)
\]

for all $v$, set $\text{dist}(v) = \infty$ (NEW!)
set $\text{dist}(s) = 0$ (NEW!)
put $s$ into the queue
while the queue is not empty
    take $u$ from the queue
    if $u$ unmarked
        mark $u$
        for every edge $(u,v)$
            put $v$ into the queue
            if $\text{dist}(v) = \infty$ (NEW!)
                $\text{dist}(v) = \text{dist}(u) + 1$ (NEW!)

$\text{dist}(v)$ gives the distance between $s$ and $v$

Why does this work?

**Lemma** In BFS, the vertices are marked in distance order (smallest first)

**Correctness sketch:** (using the Lemma)

For each $v$, $\text{dist}(v)$ only changes once, when it is first discovered and inserted into the queue.

Let $u$ be the vertex which first discovered $v$
Shortest Paths using BFS

We’ve added four (NEW!) lines to TRAVERSE to track the distances from $s$

### BFS($s$)

- for all $v$, set $\text{dist}(v) = \infty$ (NEW!)
- set $\text{dist}(s) = 0$ (NEW!)
- put $s$ into the queue
- while the queue is not empty
  - take $u$ from the queue
  - if $u$ unmarked
    - mark $u$
    - for every edge $(u, v)$
      - put $v$ into the queue
    - if $\text{dist}(v) = \infty$ (NEW!)
      - $\text{dist}(v) = \text{dist}(u) + 1$ (NEW!)

$\text{dist}(v)$ gives the distance between $s$ and $v$

Why does this work?

**Lemma** In BFS, the vertices are marked in distance order (smallest first)

**Correctness sketch:** (using the Lemma)

For each $v$, $\text{dist}(v)$ only changes once, when it is first discovered and inserted into the queue.

Let $u$ be the vertex which first discovered $v$

Assume that there is a path of length less than $\text{dist}(u) + 1$
Shortest Paths using BFS

We’ve added four (NEW!) lines to TRAVERSE to track the distances from \( s \)

\[\text{BFS}(s)\]

for all \( v \), set \( \text{dist}(v) = \infty \) (NEW!)
set \( \text{dist}(s) = 0 \) (NEW!)
put \( s \) into the queue
while the queue is not empty
    take \( u \) from the queue
    if \( u \) unmarked
        mark \( u \)
        for every edge \((u,v)\)
            put \( v \) into the queue
        if \( \text{dist}(v) = \infty \) (NEW!)
            \( \text{dist}(v) = \text{dist}(u) + 1 \) (NEW!)

\( \text{dist}(v) \) gives the distance between \( s \) and \( v \)

Why does this work?

**Lemma** In BFS, the vertices are marked in distance order (smallest first)

**Correctness sketch:** (using the Lemma)

For each \( v \), \( \text{dist}(v) \) only changes once, when it is first discovered and inserted into the queue.

Let \( u \) be the vertex which first discovered \( v \)

Assume that there is a path of length less than \( \text{dist}(u) + 1 \)

Let \( x \) be the previous vertex on this path
Shortest Paths using BFS

We’ve added four (NEW!) lines to TRAVERSE to track the distances from $s$.

**BFS($s$)**

for all $v$, set $\text{dist}(v) = \infty$ (NEW!)
set $\text{dist}(s) = 0$ (NEW!)
put $s$ into the queue
while the queue is not empty
    take $u$ from the queue
    if $u$ unmarked
        mark $u$
        for every edge $(u,v)$
            put $v$ into the queue
        if $\text{dist}(v) = \infty$ (NEW!)
            $\text{dist}(v) = \text{dist}(u) + 1$ (NEW!)

$\text{dist}(v)$ gives the distance between $s$ and $v$

Why does this work?

**Lemma** In BFS, the vertices are marked in distance order (smallest first)

**Correctness sketch:** (using the Lemma)

For each $v$, $\text{dist}(v)$ only changes once, when it is first discovered and inserted into the queue.

Let $u$ be the vertex which first discovered $v$

Assume that there is a path of length less than $\text{dist}(u) + 1$

Let $x$ be the previous vertex on this path

$x$ has distance $< \text{dist}(u)$ so should have been marked before $u$ and discovered $v$ first.
Shortest Paths using BFS

We’ve added four (NEW!) lines to TRAVERSE to track the distances from \( s \)

\[
\text{BFS}(s)
\]

for all \( v \), set \( \text{dist}(v) = \infty \) (NEW!)
set \( \text{dist}(s) = 0 \) (NEW!)
put \( s \) into the queue
while the queue is not empty
  take \( u \) from the queue
  if \( u \) unmarked
    mark \( u \)
    for every edge \( (u,v) \)
      put \( v \) into the queue
      if \( \text{dist}(v) = \infty \) (NEW!)
        \( \text{dist}(v) = \text{dist}(u) + 1 \) (NEW!)

\( \text{dist}(v) \) gives the distance between \( s \) and \( v \)

Why does this work?

\[
\text{Lemma} \text{ In BFS, the vertices are marked in distance order (smallest first)}
\]

Correctness sketch: (using the Lemma)

For each \( v \), \( \text{dist}(v) \) only changes once, when it is first discovered and inserted into the queue.

Let \( u \) be the vertex which first discovered \( v \)

Assume that there is a path of length less than \( \text{dist}(u) + 1 \)

Let \( x \) be the previous vertex on this path

\( x \) has distance \( < \text{dist}(u) \) so should have been marked before \( u \) and discovered \( v \) first

\text{contradiction!}
Conclusion

**TRAVERSE** visits every vertex in a connected graph in $O(|E|)$ time

(with an $O(1)$ time bag)

- the traversal order depends on the type of bag

(And it works for directed graphs too)

with a Queue the algorithm is called

**Breadth First Search** (BFS)

with a Stack the algorithm is called

**Depth First Search** (DFS)

**Applications**

Max-Flow

Testing whether a graph is bipartite

Shortest paths in unweighted graphs take $O(|V| + |E|)$ time using BFS

(Works for directed graphs too)

**Applications**

Finding (strongly) connected components

Topicologically sorting a Directed Acyclic Graph

Testing for planarity

**Question**

What does **TRAVERSE** do on an unconnected graph?