Graph Traversals

- **Input:** A simple, undirected, and connected graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges.

- **Objective:** Find a traversal path that
  - visits all the vertices of the graph and
  - traverses all the edges of a graph.

- **Remark:** Vertices can be visited more than once and edges can be traversed more than once.
Graph Traversals – General Scheme

- **Start** with one of the vertices and **traverse** one of its incident edges to reach another vertex.
  - Connectivity implies that this is always possible.

- Using traversed edges, **go** to one of the visited vertices that has an incident untraversed edge.
  - Connectivity implies that this is always possible.

- **Traverse** this untraversed edge.

- **Continue** until all the edges are traversed.
  - Connectivity implies that all vertices are visited.
Efficiency objectives:
- Apply simple rules to find an untraversed edge.
- Implement with an efficient data structure.
- Finish fast in $O(m)$ running time.

Example: An Euler path is the shortest possible traversal path if exists – no edge is traversed more than once.
Variants

- **Directed graphs:** An edge is traversed only from its origin to its destination.

- **Disconnected graphs:**
  - When stuck, **jump** to an unvisited vertex.
  - **Continue** until all vertices are visited and all edges are traversed.
  - Finish **fast** in $O(n + m)$ running time.
Example

Traversal path: A
Example

Traversal path: $AB$
Traversal path: \textit{ABE}
Traversal path: \textit{ABED}
Traversal path: \textit{ABEDF}
Traversal path: **ABEDFE**
Traversal path: \textit{ABEDFEB}
Example

Traversal path: \textit{ABEDFEBCE}
Example

Traversal path: \textit{ABEDFEBCA}
Traversal path: **ABEDFEBCAD**
Example

Traversal path: $ABEDFEBCADF$
Example

Traversal path: \textit{ABEDFEBCADFC}
Traversals Trees

- **The tree structure:** a *rooted*, *ordered*, and *directed* tree.
  - The first visited vertex is the *root* of the tree.
  - Vertex $u$ is the *parent* of $v$ if the first visit to $v$ happened after traversing the edge $(u, v)$.
  - The children of a vertex $u$ are ordered according to the time they were first visited from $u$. 

---

**Edge classification:**
- **Tree edge:** an edge from a vertex to one of its children.
- **Back edge:** an edge from a vertex to one of its ancestors.
- **Forward edge:** an edge from a vertex to one of its descendants which is not its child.
- **Cross edge:** an edge from a vertex to another vertex that is neither one of its ancestors nor one of its descendants in the traversal tree.
Traversal Trees

- **The tree structure:** a rooted, ordered, and directed tree.
  - The first visited vertex is the *root* of the tree.
  - Vertex $u$ is the *parent* of $v$ if the first visit to $v$ happened after traversing the edge $(u, v)$.
  - The children of a vertex $u$ are ordered according to the time they were first visited from $u$.

- **Edge classification:**
  - **Tree edge:** an edge from a vertex to one of its children.
  - **Back edge:** an edge from a vertex to one of its ancestors.
  - **Forward edge:** an edge from a vertex to one of its descendants which is not its child.
  - **Cross edge:** an edge from a vertex to another vertex that is neither one of its ancestors nor one of its descendants in the traversal tree.
Example

Traversal path: \( A \)
Example

Traversal path: $AB$
Traversal path: $ABE$
Example

Traversal path: \(ABED\)
Traversal path: \textit{ABEDF}
Example

Traversal path: $ABEDFE$
Traversal path: \textit{ABEDFEB}
Example

Traversal path: \textit{ABEDFEBBC}
Traversal path: $ABEDFEBCA$
Example

Traversal path: \( ABEDFEBCADE \)
Traversal path: $ABEDFEBCADF$
Traversal path: $ABEDFEBCADFC$
DFS – Depth First Search

- **Visiting order:** Visit a vertex, then recursively visit all of its neighbors in order.

- **Input:** An undirected graph $G = (V, E)$ and a global order on the $n$ vertices.

- **Output:** A traversal forest that contains a traversal tree for each connected component of the graph.

- **Directed graphs:**
  - In the traversal forest, each tree is a directed tree.
  - In each tree, there is a directed path from the root to any other vertex.
Example – a DFS traversal Path

Traversal path: A
Example – a DFS traversal Path

Traversal path: $AB$
Example – a DFS traversal Path

Traversal path: $ABC$
Example – a DFS traversal Path

Traversal path: \textit{ABCA}
Example – a DFS traversal Path

Traversal path: ABCAC
Example – a DFS traversal Path

Traversal path: ABCACF
Example – a DFS traversal Path

Traversal path: $ABCACFD$
Example – a DFS traversal Path

Traversal path: **ABCACFDA**
Example – a DFS traversal Path

Traversal path: $\textit{ABCACFDAD}$
Example – a DFS traversal Path

Traversal path: \textit{ABCACFDADE}
Example – a DFS traversal Path

Traversal path: $ABCACFDADEB$
Example – a DFS traversal Path

Traversal path: $ABCACFDADAEBC$
Example – a DFS traversal Path

Traversal path: ABCACFDADEBEF
Example – a DFS traversal Path

Traversal path: \textit{ABCACFDADEBEF – EDFCBA}
Example – the DFS traversal Tree

Traversal path: A
Example – the DFS traversal Tree

Traversal path: $AB$
Example – the DFS traversal Tree

Traversal path: \( ABC \)
Example – the DFS traversal Tree

Traversal path: \textit{ABCA}
Example – the DFS traversal Tree

Traversal path: $ABCAC$
Example – the DFS traversal Tree

Traversal path: $ABCACF$
Example – the DFS traversal Tree

Traversal path: \textit{ABCACFD}
Example – the DFS traversal Tree

Traversal path: \textit{ABCACFDA}
Example – the DFS traversal Tree

Traversal path: $ABCACFDAD$
Example – the DFS traversal Tree

Traversal path: \textit{ABCACFDADE}
Example – the DFS traversal Tree

Traversal path: **ABCACFDADEB**
Example – the DFS traversal Tree

Traversal path: \(ABCACFDADEBE\)
Example – the DFS traversal Tree

Traversal path: \textit{ABCACFDADDEBEF}
Variables for the DFS Procedure

- Each vertex is colored by one of the following colors:
  - **White**: The recursive visit has not started.
  - **Gray**: The recursive visit started but not finished.
  - **Black**: The recursive visit finished.

A global discrete time variable $time$ that is updated when a vertex becomes **Gray** and when a vertex becomes **Black**.

- $d(v)$: The time vertex $v$ becomes **Gray**.
- $f(v)$: The time vertex $v$ becomes **Black**.

A parenthood function $\Pi(v)$ in the traversal forest:
- $\Pi(v) = \text{nil}$ if $v$ is a root of a tree in the forest.
- $\Pi(v)$ is $v$'s parent in the traversal tree that contains $v$. 
Variables for the DFS Procedure

Each vertex is colored by one of the following colors:

- **White**: The recursive visit has not started.
- **Gray**: The recursive visit started but not finished.
- **Black**: The recursive visit finished.

A global discrete time variable \( \text{time} \) that is updated when a vertex becomes **Gray** and when a vertex becomes **Black**.

- \( d(v) \): The time vertex \( v \) becomes **Gray**.
- \( f(v) \): The time vertex \( v \) becomes **Black**.
Variables for the DFS Procedure

Each vertex is colored by one of the following colors:
- **White**: The recursive visit has not started.
- **Gray**: The recursive visit started but not finished.
- **Black**: The recursive visit finished.

A global discrete time variable **time** that is updated when a vertex becomes **Gray** and when a vertex becomes **Black**.
- **d(v)**: The time vertex v becomes **Gray**.
- **f(v)**: The time vertex v becomes **Black**.

A parenthood function **Π(v)** in the traversal forest:
- **Π(v) = nil** if v is a root of a tree in the forest.
- **Π(v)** is v’s parent in the traversal tree that contains v.
The DFS Procedure – Initial Call

\[
\text{DFS}(G) \\
\text{for each vertex } v \in V \text{ do} \\
\quad \text{Color}(v) = \text{White} \\
\quad \Pi(v) = \text{nil} \\
\quad \text{time} = 0 \\
\text{for each vertex } v \in V \text{ do} \\
\quad \text{if Color}(v) = \text{White then} \\
\quad \text{DFS-Visit}(v)
\]
The DFS Procedure – Recursive Call

**DFS-Visit**($v$)

- $Color(v) = Gray$
- $time++$
- $d(v) = time$
- **for each** neighbor $u$ of $v$ **do**
  - **if** $Color(u) = white$ **then**
    - $\Pi(u) = v$
    - **DFS-Visit**($u$)
  - $Color(v) = Black$
- $time++$
- $f(v) = time$
Example – Directed DFS

time = 0
Example – Directed DFS

\[ \text{time} = 1 \]
Example – Directed DFS

time = 2
Example – Directed DFS

\[ time = 3 \]
Example – Directed DFS

time = 4
Example – Directed DFS

time = 5
Example – Directed DFS

votes = 6
Example – Directed DFS

time = 7
Example – Directed DFS

time = 8
Example – Directed DFS

\[ \text{time} = 9 \]
Example – Directed DFS

![Directed DFS Diagram]

\[ \text{time} = 10 \]
Example – Directed DFS

\[ \text{time} = 11 \]
Example – Directed DFS

time = 12
DFS – Correctness

**Lemma:** DFS visits all the vertices.

**Proof:** Each vertex changes colors as follows: \( \text{White} \rightarrow \text{Gray} \rightarrow \text{Black} \)

**Lemma:** DFS traverses all the edges.

**Proof:** For each vertex, DFS examines all of its incident edges in undirected graphs and all of its outgoing edges in directed graphs.
Lemma: DFS visits all the vertices.

Proof: Each vertex changes colors as follows:

\[ \text{White} \rightarrow \text{Gray} \rightarrow \text{Black} \]

Lemma: DFS traverses all the edges.

Proof: For each vertex, DFS examines all of its incident edges in undirected graphs and all of its outgoing edges in directed graphs.

Traversal Path:

- In undirected graphs the traversal path is not explicit. Traversing back the edges is done implicitly due to the recursive calls.
- In directed graphs the traversal path sometimes uses the wrong direction due to the recursion.
DFS – Time Complexity

**Adjacency lists:** $\Theta(n + m)$.

- $\Theta(m)$: Each edge is examined once in directed graphs and twice in undirected graphs.

- $\Theta(n)$: Each vertex is colored three times.
DFS – Time Complexity

- **Adjacency lists:** $\Theta(n + m)$.
  - $\Theta(m)$: Each edge is examined once in directed graphs and twice in undirected graphs.
  - $\Theta(n)$: Each vertex is colored three times.

- **Adjacency matrix:** $\Theta(n^2)$.
  - $\Theta(n)$: For each vertex, for examining all of its incident edges in undirected graphs and all of its outgoing edges in directed graphs.
**Definition:** The DFS interval of vertex $v$ is $[d(v), f(v)]$.

**Lemma:** Assume $d(u) < d(v)$ for an edge $(u, v)$:
- Either the intervals of $u$ and $v$ are disjoint
  $$d(u) < f(u) < d(v) < f(v).$$
- Or $u$’s interval contains $v$’s interval
  $$d(u) < d(v) < f(v) < f(u).$$
Example – Time Intervals

\begin{align*}
time = 1 & \quad A = \text{Gray} \quad d(A) = 1 \\
time = 2 & \quad B = \text{Gray} \quad d(B) = 2 \\
time = 3 & \quad C = \text{Gray} \quad d(C) = 3 \\
time = 4 & \quad C = \text{Black} \quad f(C) = 4 \\
time = 5 & \quad E = \text{Gray} \quad d(E) = 5 \\
time = 6 & \quad D = \text{Gray} \quad d(D) = 6 \\
time = 7 & \quad D = \text{Black} \quad f(D) = 7 \\
time = 8 & \quad E = \text{Black} \quad f(E) = 8 \\
time = 9 & \quad B = \text{Black} \quad f(B) = 9 \\
time = 10 & \quad A = \text{Black} \quad f(A) = 10 \\
time = 11 & \quad F = \text{Gray} \quad d(F) = 11 \\
time = 12 & \quad F = \text{Black} \quad f(F) = 12
\end{align*}
Example – Time Intervals

A

B

C

D

E

F

1 2 3 4 5 6 7 8 9 10 11 12
DFS – Traversal Tree Edge Classification

- **A tree edge (T):** An edge from a *Gray* vertex to a *White* vertex.

- **A back edge (B):** An edge from a *Gray* vertex to a *Gray* vertex.

- **A forward edge (F):** An edge from a *Gray* vertex to a *Black* vertex whose interval is nested.

- **A cross edge (C):** An edge from a *Gray* vertex to a *Black* vertex whose interval finished.
Example – Edge Classification

(A,B) \(\text{Gray} \rightarrow \text{White}\)
(B,C) \(\text{Gray} \rightarrow \text{White}\)
(B,E) \(\text{Gray} \rightarrow \text{White}\)
(E,D) \(\text{Gray} \rightarrow \text{White}\)
(D,A) \(\text{Gray} \rightarrow \text{Gray}\)
(A,C) \(\text{Gray} \rightarrow \text{Black}\) (nested)
(F,C) \(\text{Gray} \rightarrow \text{Black}\) (finished)
(F,D) \(\text{Gray} \rightarrow \text{Black}\) (finished)
(F,E) \(\text{Gray} \rightarrow \text{Black}\) (finished)
Lemma: DFS has no forward and no cross edges.

Proof:

1. There is no edge from a *Gray* vertex to a *Black* vertex.
2. Otherwise, the *Black* vertex, before becoming black, would examine its incident edge to the *Gray* vertex in the other direction.
Cycles in Undirected Graphs

**Theorem:** An undirected graph has a cycle if and only if the DFS creates at least one back edge.
Cycles in Undirected Graphs

- **Theorem**: An undirected graph has a cycle iff the DFS creates at least one **back** edge.

- **Proof** \(\iff\)
  - Suppose that \((u, v)\) is a **back** edge examined at \(u\).
  - Then \(v\) is an ancestor of \(u\).
  - Therefore, there exists a path from \(v\) to \(u\).
  - Adding \((u, v)\) to the path creates a cycle.
Cycles in Undirected Graphs

**Theorem:** An undirected graph has a cycle iff the DFS creates at least one **back** edge.

**Proof ⇒**

- Suppose that \( C \) is a cycle in \( G \).
- Let \( v \) be the first *Gray* vertex in \( C \).
- Let \((u, v)\) be the preceding edge in \( C \).
- After time \( d(v) \), DFS explores the \( v \) to \( u \) path.
- When \((u, v)\) is traversed, both \( v \) and \( u \) are *Gray*.
- Hence, \((u, v)\) is a **back** edge.
DFS – Application

Problem:

- Is an undirected graph a forest or a tree?
- Does an undirected graph have a cycle?

Algorithm:

Run the DFS algorithm.

Terminate if a back edge is found.

Time complexity: \(O(n)\).

If there are no cycles there are \(O(n)\) edges. Therefore, the DFS time complexity \(O(m + n)\) becomes \(O(n)\).

If there exists a cycle, the algorithm terminates the first time a Gray vertex is visited for a second time.
**Problem:**

- Is an undirected graph a forest or a tree?
- Does an undirected graph have a cycle?

**Algorithm:**

- Run the DFS algorithm.
- Terminate if a **back** edge is found.
Problem:
- Is an undirected graph a forest or a tree?
- Does an undirected graph have a cycle?

Algorithm:
- Run the DFS algorithm.
- Terminate if a back edge is found.

Time complexity: $O(n)$.
- If there are no cycles there are $O(n)$ edges. Therefore, the DFS time complexity $O(m + n)$ becomes $O(n)$.
- If there exists a cycle, the algorithm terminates the first time a Gray vertex is visited for a second time.
Theorem: A directed graph $G$ is DAG iff running DFS on $G$ does not produce a back edge.
**Theorem:** A directed graph $G$ is DAG iff running DFS on $G$ does not produce a *back* edge.

**Proof $\Rightarrow$**

- Suppose that $(u \rightarrow v)$ is a *back* edge.
- Then $v$ is an ancestor of $u$.
- Therefore, there exists a directed path from $v$ to $u$.
- Adding $(u \rightarrow v)$ to the path creates a directed *cycle*. 
Theorem: A directed graph $G$ is DAG iff running DFS on $G$ does not produce a back edge.

Proof $\Leftarrow$

- Suppose that $C$ is a directed cycle in $G$.
- Let $v$ be the first Gray vertex in $C$.
- Let $(u \rightarrow v)$ be the preceding edge in $C$.
- After time $d(v)$, DFS explores the $v$ to $u$ directed path.
- When $(u \rightarrow v)$ is traversed, both $v$ and $u$ are Gray.
- Hence, $(u \rightarrow v)$ is a back edge.
Definition: A topological sort of a DAG is a linear ordering of its vertices such that vertex $u$ appears before vertex $v$ for any directed edge $(u \rightarrow v)$.

Problem: Find one of the topological sorts of a DAG.
**Topological Sort of a DAG**

**Definition:** A **topological sort** of a DAG is a linear ordering of its vertices such that vertex \( u \) appears before vertex \( v \) for any directed edge \( (u \to v) \).

**Problem:** Find one of the topological sorts of a DAG.

**Algorithm:**

- **Create** an empty linked list.
- **Run** DFS on the DAG.
- A vertex becomes **Black**: add it to the front of the list.
- **Return** the linked list as a topological sort.
Example
Example
The Algorithm Finds a Topological Sort for a DAG

- **Correctness:**
  - Consider examining the edge \((u \rightarrow v)\).
  - At this time \(u\) is *Gray*.
  - \(v\) is not *Gray* since then \((u \rightarrow v)\) is a *back* edge.
  - If \(v\) is *White* then \(f(v) < f(u)\) since \(v\) becomes *Black* before \(u\).
  - If \(v\) is *Black* then \(f(v) < f(u)\) since \(u\) is still *Gray*.
  - \(f(v) < f(u)\) implies that \(u\) appears before \(v\) in the list.
The Algorithm Finds a Topological Sort for a DAG

**Correctness:**

- Consider examining the edge \((u \rightarrow v)\).
- At this time \(u\) is *Gray*.
- \(v\) is not *Gray* since then \((u \rightarrow v)\) is a **back** edge.
- If \(v\) is *White* then \(f(v) < f(u)\) since \(v\) becomes *Black* before \(u\).
- If \(v\) is *Black* then \(f(v) < f(u)\) since \(u\) is still *Gray*.
- \(f(v) < f(u)\) implies that \(u\) appears before \(v\) in the list.

**Complexity:**

- \(\Theta(n + m)\) running time.
- The same complexity as DFS.
**Visiting order:** Visit a vertex, then visit all of its neighbors, then visit all of the neighbors of its neighbors, ...  

**Input:** An undirected graph $G = (V, E)$ and a global order on the $n$ vertices.  

**Output:** A traversal forest that contains a traversal tree for each connected component of the graph.  

**Directed graphs:**  
- In the traversal forest, each tree is a directed tree.  
- In each tree, there is a directed path from the root to any other vertex.
Example – a BFS traversal Path

Traversal path: A
Example – a BFS traversal Path

Traversal path: \( AB \)
Example – a BFS traversal Path

Traversal path: ABA
Example – a BFS traversal Path

Traversal path: \textit{ABAC}
Example – a BFS traversal Path

Traversal path: **ABACA**
Example – a BFS traversal Path

Traversal path: \textit{ABACD}
Example – a BFS traversal Path

Traversal path: \textit{ABACDA}
Example – a BFS traversal Path

Traversal path: \( ABACDAB \)
Example – a BFS traversal Path

Traversal path: \textit{ABACDABC}
Example – a BFS traversal Path

Traversal path: \textit{ABACDABCB}
Example – a BFS traversal Path

Traversal path: $ABACDABCBE$
Example – a BFS traversal Path

Traversal path: \textit{ABACDABBCBEB}
Example – a BFS traversal Path

Traversal path: \textit{ABACDABCBEBA}
Example – a BFS traversal Path

Traversal path: \textit{ABACDABCBEBAC}
Example – a BFS traversal Path

Traversal path: $ABACDABCBEBACF$
Example – a BFS traversal Path

Traversal path: \( ABACDABCBEBACFC \)
Example – a BFS traversal Path

Traversal path: $ABACDABCBEBACFCA$
Example – a BFS traversal Path

Traversal path:  \textit{ABACDABCBEBACFCAD}
Example – a BFS traversal Path

Traversal path : \textit{ABACDABCBEBACFCADE}
Example – a BFS traversal Path

Traversal path: \textit{ABACDABCBEBACFCADED}
Example – a BFS traversal Path

Traversal path: \textit{ABACDABCBEBACFCADEDF}
Example – a BFS traversal Path

Traversal path: \(ABACDABCBEBACFCADEDFD\)
Example – a BFS traversal Path

Traversal path: \(ABACDABCBEBACFCADEDFDA\)
Traversals path: **ABACDABCBEBACFCADEDFDAB**
Example – a BFS traversal Path

Traversal path: \textit{ABACDABCBEBACFCADEDFDABE}
Example – a BFS traversal Path

Traversal path: \textit{ABACDABCBEBAFCADEDFDABEF}
Example – the BFS Traversal Tree

ABACADABCBEBACFCADEDFDABEF
Variables for the BFS Procedure

- A level function $Level(v)$:
  - $Level(v) = 0$ if $v$ is a root of a tree in the forest.
  - $Level(v) = \ell$ if $v$ is at distance $\ell$ from the root.
Variables for the BFS Procedure

- A level function $Level(v)$:
  - $Level(v) = 0$ if $v$ is a root of a tree in the forest.
  - $Level(v) = \ell$ if $v$ is at distance $\ell$ from the root.

- A FIFO queue $Q$:
  - $x = First(Q)$: Delete $x$ the first element in $Q$.
  - $Last(Q) = x$: Add $x$ as the last element in $Q$.
  - $Create(Q)$: Create an empty queue $Q$.
  - $Empty(Q)$: Check if the queue $Q$ is empty.
Variables for the BFS Procedure

- **A level function** $Level(v)$:
  - $Level(v) = 0$ if $v$ is a root of a tree in the forest.
  - $Level(v) = \ell$ if $v$ is at distance $\ell$ from the root.

- **A FIFO queue** $Q$:
  - $x = First(Q)$: Delete $x$ the first element in $Q$.
  - $Last(Q) = x$: Add $x$ as the last element in $Q$.
  - $Create(Q)$: Create an empty queue $Q$.
  - $Empty(Q)$: Check if the queue $Q$ is empty.

- **A parenthood function** $\Pi(v)$ in the traversal forest:
  - $\Pi(v) = nil$ if $v$ is a root of a tree in the forest.
  - $\Pi(v)$ is $v$’s parent in the traversal tree that contains $v$. 

Amotz Bar-Noy (CUNY)
The BFS Procedure – Initial Call

\textbf{BFS}(G)

\begin{itemize}
\item \textbf{for each} vertex \( v \in V \) \textbf{do}
  \begin{itemize}
  \item \texttt{Level}(v) = \infty
  \item \texttt{Π}(v) = \textit{Nil}
  \end{itemize}
\end{itemize}

\textbf{for each} vertex \( r \in V \) \textbf{do}

\begin{itemize}
\item \textbf{if} \texttt{Level}(r) = \infty \textbf{then}
  \begin{itemize}
  \item \texttt{BFS-Visit}(r)
  \end{itemize}
\end{itemize}
BFS-Visit($r$)

$Level(r) = 0$

Create($Q$)

$Last(Q) = r$

while (not Empty($Q$)) do

$v = First(Q)$

for each neighbor $u$ of $v$ do

if $Level(u) = \infty$ then

$\Pi(u) = v$

$Level(u) = Level(v) + 1$

$Last(Q) = u$
BFS – Example

Amotz Bar-Noy (CUNY)  Graph Traversals  Spring 2012  122 / 140
BFS – Correctness

- **Lemma:** BFS *visits* all the vertices.
  
  **Proof:** Each vertex changes its level $\ell$ as follows
  $$\ell = \infty \rightarrow 0 \leq \ell < \infty$$

- **Lemma:** BFS *traverses* all the edges.
  
  **Proof:** For each vertex, BFS examines all of its incident edges in undirected graphs and all of its outgoing edges in directed graphs.

**Traversal Path:**
- **Undirected graphs:** The traversal path is not explicit. Traversing back the edges is done implicitly due to the queue handling.
- **Directed graphs:** The traversal path sometimes uses the wrong direction due to the queue handling.
BFS – Correctness

**Lemma:** BFS visits all the vertices.

**Proof:** Each vertex changes its level $l$ as follows

$$l = \infty \rightarrow 0 \leq l < \infty$$

**Lemma:** BFS traverses all the edges.

**Proof:** For each vertex, BFS examines all of its incident edges in undirected graphs and all of its outgoing edges in directed graphs.

**Traversal Path:**

- **Undirected graphs:** The traversal path is not explicit. Traversing back the edges is done implicitly due to the queue handling.
- **Directed graphs:** The traversal path sometimes uses the wrong direction due to the queue handling.
BFS – Time Complexity

- **Adjacency lists:** $\Theta(n + m)$.
  - $\Theta(m)$: Each edge is examined once in directed graphs and twice in undirected graphs.
  - $\Theta(n)$: Each vertex gets a level $\ell < \infty$ exactly once.
Adjacency lists: $\Theta(n + m)$.
- $\Theta(m)$: Each edge is examined once in directed graphs and twice in undirected graphs.
- $\Theta(n)$: Each vertex gets a level $\ell < \infty$ exactly once.

Adjacency matrix: $\Theta(n^2)$.
- $\Theta(n)$: For each vertex, for examining all of its incident edges in undirected graphs and all of its outgoing edges in directed graphs.
The BFS Traversal Tree

**Lemma:** There are no forward edges neither in directed graphs nor in undirected graphs.

**Proof:** A forward edge would be a tree edge.

**Lemma:** There are no back edges in undirected graphs.

**Proof:** A back edge would be a tree edge while examined in the other direction.
The BFS Traversal Tree

**Lemma:** In undirected graphs, \( \text{Level}(u) = \text{Level}(v) \) or \( \text{Level}(u) = \text{Level}(v) - 1 \) for a **cross** edge \((u, v)\).

**Proof:**
- \( u \) is added to the queue before \( v \).
- Otherwise BFS would examine \((v, u)\) before \((u, v)\).
- Therefore, \( \text{Level}(u) \leq \text{Level}(v) \).
- \( \text{Level}(u) < \text{Level}(v) - 1 \Rightarrow (u, v) \) would be a **tree** edge.
Problem: Let $G$ be a connected undirected graph and let $u$ and $v$ be 2 vertices. Find one of the shortest paths from $u$ to $v$ and its length.

Algorithm:
Run the BFS algorithm starting with the vertex $v$.
Terminate when the vertex $u$ is visited the first time.
The length of the shortest path from $u$ to $v$ is the level of vertex $u$ in the BFS traversal tree.
The path $(u, \pi(u), \pi(\pi(u)), \ldots, v)$ is one of the shortest paths from $u$ to $v$.

Time complexity $O(n+m)$: the same as BFS.
**Problem:** Let $G$ be a connected undirected graph and let $u$ and $v$ be 2 vertices. Find one of the shortest paths from $u$ to $v$ and its length.

**Algorithm:**
- **Run** the BFS algorithm starting with the vertex $v$.
- **Terminate** when the vertex $u$ is visited the first time.
- The length of the shortest path from $u$ to $v$ is the level of vertex $u$ in the BFS traversal tree.
- The path $(u, \Pi(u), \Pi(\Pi(u)), \ldots, v)$ is one of the shortest paths from $u$ to $v$. 

**Time complexity** $O(n + m)$: the same as BFS.
**Problem:** Let $G$ be a connected undirected graph and let $u$ and $v$ be 2 vertices. Find one of the shortest paths from $u$ to $v$ and its length.

**Algorithm:**
- **Run** the BFS algorithm starting with the vertex $v$.
- **Terminate** when the vertex $u$ is visited the first time.
- The length of the shortest path from $u$ to $v$ is the level of vertex $u$ in the BFS traversal tree.
- The path $(u, \Pi(u), \Pi(\Pi(u)), \ldots, v)$ is one of the shortest paths from $u$ to $v$.

**Time complexity** $O(n + m)$: the same as BFS.
Diameter of a Graph

- **Assumption:** Let $G$ be a connected undirected graph.

- **Notation:** For two vertices $u$ and $v$, denote by $\text{dist}(u, v)$ the length of the shortest path from $u$ to $v$ in $G$.

- **Definition:** The diameter of $G$, denoted by $D(G)$, is the longest shortest path in $G$:

  $$D(G) = \max_{u, v \in G} \{ \text{dist}(u, v) \}.$$  

- **Problem:** Find the diameter of $G$. 
Algorithm for any Graph $G$

**Algorithm:**

- For all vertices $v$, run BFS starting with the vertex $v$.
- Let $u$ be one of the **farthest** vertices from $v$ and let $r(v) = \text{dist}(v, u)$.
- $D(G) = \max_v \{r(v)\}$.

**Correctness:** By definition of BFS.

**Complexity:** $O(nm)$ for $n$ times BFS.
Algorithm for any Graph $G$

**Algorithm:**

- For all vertices $v$, run BFS starting with the vertex $v$.
- Let $u$ be one of the farthest vertices from $v$ and let $r(v) = \text{dist}(v, u)$.
- $D(G) = \max_v \{ r(v) \}$.

**Correctness:** By definition of BFS.

**Complexity:** $O(nm)$ for $n$ times BFS.
Problem: Find the diameter of a tree $T$. 

Algorithm:
- Run BFS starting with an arbitrary vertex $v$.
- Let $u$ be one of the farthest vertices from $v$.
- Run BFS starting with the vertex $u$.
- Let $w$ be one of the farthest vertices from $u$.
- $D(G) = \text{dist}(u, w)$.

Complexity: $O(m)$ for two times BFS.
**Problem:** Find the diameter of a tree $T$.

**Algorithm:**
- Run BFS starting with an arbitrary vertex $v$.
- Let $u$ be one of the **farthest** vertices from $v$.
- Run BFS starting with the vertex $u$.
- Let $w$ be one of the **farthest** vertices from $u$.
- $D(G) = \text{dist}(u, w)$.
**Problem:** Find the diameter of a tree $T$.

**Algorithm:**
- Run BFS starting with an arbitrary vertex $v$.
- Let $u$ be one of the **farthest** vertices from $v$.
- Run BFS starting with the vertex $u$.
- Let $w$ be one of the **farthest** vertices from $u$.
- $D(G) = \text{dist}(u, w)$.

**Complexity:** $O(m)$ for two times BFS.
Correctness

Let $x$ and $y$ be two vertices such that $D(G) = \text{dist}(x, y)$.

If $v$ is either $x$ or $y$, then $\text{dist}(v, w) = D(G)$ and therefore $\text{dist}(u, w) = D(G)$.

Assume that $\text{dist}(v, y) \geq \text{dist}(v, x)$.

Let $z$ be the vertex that connects $v$ to the path $x$ to $y$.

$z$ can be $v$ or $y$. 
Correctness

Let $x$ and $y$ be two vertices such that $D(G) = \text{dist}(x, y)$.

If $v$ is either $x$ or $y$, then $\text{dist}(v, w) = D(G)$ and therefore $\text{dist}(u, w) = D(G)$.

Assume that $\text{dist}(v, y) \geq \text{dist}(v, x)$.

Let $z$ be the vertex that connects $v$ to the path $x$ to $y$.

$z$ can be $v$ or $y$. 

![Graph Diagram]

(From the given content, it can be inferred that the diagram shows a sequence of vertices connected by edges, with $x$, $z$, $y$, and $v$ labeled.)
Assume that \( z \) is on the path from \( v \) to \( u \).

By definition, \( \text{dist}(v, u) \geq \text{dist}(v, y) \).

Therefore, \( \text{dist}(z, u) \geq \text{dist}(z, y) \).

Therefore, \( D(G) = \text{dist}(x, u) \).

By definition, \( \text{dist}(u, w) \geq \text{dist}(u, x) \).

Therefore, \( D(G) = \text{dist}(u, w) \).
Assume that \( z \) is not on the path from \( v \) to \( u \).

Let \( p \) be the vertex on the path from \( v \) to \( u \) and the path from \( v \) to \( z \) (\( p \) can be \( v \)).

By definition, \( \text{dist}(v, u) \geq \text{dist}(v, y) \).

Therefore, \( \text{dist}(p, u) \geq \text{dist}(p, y) \).
Correctness: Case II

- \( \text{dist}(z, u) > \text{dist}(p, u) \) and \( \text{dist}(z, y) < \text{dist}(p, y) \) by definition.
- Therefore, \( \text{dist}(z, u) > \text{dist}(z, y) \).
- Therefore, \( \text{dist}(x, u) > \text{dist}(x, y) \).
- A contradiction since \( D(G) = \text{dist}(x, y) \).
The Algorithm Fails for Non-Trees

- $u$ is the only farthest vertex from $v$.
- $v = w$ is the only farthest vertex from $u$.
- $\text{dist}(x, y) > \text{dist}(u, w)$.
**Definition:** A directed graph $G = (V, E)$ is strongly connected if for any pair of vertices $u, v \in V$ there exists a directed path from $u$ to $v$.

**Problem:** Given a directed graph $G = (V, E)$, determine if $G$ is strongly connected.
Algorithm I

**Algorithm:**

- For all vertices \( v \in V \), run DFS or BFS starting with the vertex \( v \).
- If there exists a vertex \( u \) that is not reachable from \( v \), then return **NO** else return **YES**.
Algorithm I

- **Algorithm:**
  - For all vertices \( v \in V \), run DFS or BFS starting with the vertex \( v \).
  - If there exists a vertex \( u \) that is not reachable from \( v \), then return **NO** else return **YES**.

- **Correctness:** By definition of DFS and BFS.
Algorithm I

**Algorithm:**
- For all vertices \( v \in V \), run DFS or BFS starting with the vertex \( v \).
- If there exists a vertex \( u \) that is not reachable from \( v \), then return \( \text{NO} \) else return \( \text{YES} \).

**Correctness:** By definition of DFS and BFS.

**Complexity:** \( O(nm) \) for \( n \) times DFS or BFS.
- If \( m < n \), then the underlying undirected graph is not connected, and DFS or BFS would return \( \text{NO} \) after the first run.
- The **worst case** of \( n \) runs of DFS or BFS happens only when \( m > n \). Therefore, the complexity is \( O(nm) \) and not \( O(n(n + m)) \).
Algorithm II

1. For some vertex \( w \), run DFS or BFS on \( G \) starting with \( w \).
2. If there exists \( u \) that is not reachable from \( w \), then return \textbf{NO}.
3. Reverse the direction of all the edges in \( G \) to get a new directed graph \( G' \).
4. Run DFS or BFS on \( G' \) starting with \( w \).
5. If there exists \( v \) that is not reachable from \( w \), then return \textbf{NO}.
6. Else, return \textbf{YES}.
Algorithm II – Correctness

The algorithm returns **NO**: 
- The algorithm produces an evidence of two vertices with no directed path between them.
- Either there exists \( u \) s.t. there is no directed path from \( w \) to \( u \) in \( G \).
- Or there exists \( v \) s.t there is no directed path from \( w \) to \( v \) in \( G' \) implying having no directed path from \( v \) to \( w \) in \( G \).

The algorithm returns **YES**: 
- Let \( v, u \in V \) be any pair of vertices in \( G \).
- \( YES \) in \( G \) \( \Rightarrow \) there is a directed path from \( w \) to \( u \) in \( G \).
- \( YES \) in \( G' \) \( \Rightarrow \) there is a directed path from \( w \) to \( v \) in \( G' \) implying a directed path from \( v \) to \( w \) in \( G \).

Combining the path from \( v \) to \( w \) with the path from \( w \) to \( u \) generates the path from \( v \) to \( u \).
Algorithm II – Correctness

The algorithm returns **NO:**
- The algorithm produces an evidence of two vertices with no directed path between them.
- Either there exists $u$ s.t. there is no directed path from $w$ to $u$ in $G$.
- Or there exists $v$ s.t there is no directed path from $w$ to $v$ in $G'$ implying having no directed path from $v$ to $w$ in $G$.

The algorithm returns **YES:**
- Let $v, u \in V$ be any pair of vertices in $G$.
- **YES** in $G$ $\Rightarrow$ there is a directed path from $w$ to $u$ in $G$.
- **YES** in $G'$ $\Rightarrow$ there is a directed path from $w$ to $v$ in $G'$ implying a directed path from $v$ to $w$ in $G$.
- Combining the path from $v$ to $w$ with the path from $w$ to $u$ generates the path from $v$ to $u$. 
Algorithm II – Complexity

- $O(n + m)$ for running twice DFS or BFS.
- $O(n + m)$ for reversing the direction of all edges to produce the graph $G'$ by scanning all the outgoing list to generate new outgoing lists.
- $O(n + m)$ overall complexity.
Algorithm II – Complexity

- $O(n + m)$ for running twice DFS or BFS.
- $O(n + m)$ for reversing the direction of all edges to produce the graph $G'$ by scanning all the outgoing list to generate new outgoing lists.
- $O(n + m)$ overall complexity.

Better than Algorithm I!