Connectivity

In a undirected graph \( G = (V, E) \)

- A **path** is a sequence \( P \) of vertices \( v_1, v_2, \ldots, v_k \in V \) such that \( \{v_i, v_{i+1}\} \in E \) for any \( 1 \leq i < k \).
- A **cycle** is a path \( v_1, v_2, \ldots, v_k \) with \( k \geq 2 \) and \( v_1 = v_k \).
- Graph \( G \) is **connected** if for every pair of vertices \( u \) and \( v \), there is a path from \( u \) to \( v \).
- A **connected component** containing \( u \) is the set of all vertices connected to \( u \).

All carry over naturally to directed graphs, except connectivity.

- A directed graph is **strongly connected** if, for every two vertices \( u \) and \( v \), there is a path from \( u \) to \( v \) and a path from \( v \) to \( u \).
An undirected graph \( G \) is a **tree** if it is connected and does not contain a cycle.

**Proposition**

Every tree has exactly \( n - 1 \) edges.
Graph Traversal

**s \rightarrow t** Connectivity Problem

Given a graph \( G = (V, E) \) and two particular vertices \( u \) and \( v \). Is there a path from \( u \) to \( v \)?

Traversal Problem

Given a graph \( G = (V, E) \) and a vertex \( v \). Find all vertices that can be reached from \( v \).
Basic Graph Search Algorithm

Algorithm: \texttt{Explore}(u):

- Initialize $R = \{u\}$;
- while there is an edge $(x, y)$ with $x \in R$ and $y \notin R$ do
  - Add $y$ to $R$;
- return $R$.

Proposition

\texttt{Explore}(u) returns the connected component that contains $u$.

- Naive search: $O(m)$ time for each scan, $O(mn)$ time in total.
Algorithm: \textbf{SmartExplore}(u):

- Initialize \( R = \{u\} \);
- Mark all vertices as unvisited;
- Mark \( u \) as visited;
- \textbf{while} \( R \) \textit{is not empty} \textbf{do}
  - Pick one vertex \( x \) in \( R \), remove \( x \) from \( R \);
  - \textbf{foreach} vertex \( y \in \text{Adj}(x) \) \textbf{do}
    - \textbf{if} \( y \) \textit{is not visited} \textbf{then}
      - Mark \( y \) as visited and add \( y \) to \( R \);
  - \textbf{return} the set of all visited vertices.

- Runs in \( O(m + n) \) time!
- How to determine which vertex to pick in \( R \)?
The Data Structure

Alternative 1: **Queue**
- First in first out (FIFO)
- **Breadth First Search (BFS)**
- Exploring distances

Alternative 2: **Stack**
- Last in first out (LIFO)
- **Depth First Search (DFS)**
- Exploring graph structure
A **queue** is a linked list with two operations:

- **Enqueue**\((Q, x)\): insert an element \(x\) at the rear of the queue \(Q\).
- **Dequeue**\((Q)\): remove the front element of the queue.

**Implementation:**

- linked list with two pointers
- array with two pointers
Breadth First Search

Algorithm: \textbf{BFS}(u):
\begin{itemize}
  \item Initialize queue \( Q \) to be empty;
  \item Mark all vertices as unvisited;
  \item Initialize search tree \( T \) to be empty;
  \item Mark \( u \) as visited and enqueue \((Q, u)\);
  \item while \( Q \) is not empty do
    \begin{itemize}
      \item \( x = \text{dequeue}(Q) \);
      \item \textbf{foreach vertex} \( y \in \text{Adj}(x) \) \textbf{do}
      \begin{itemize}
        \item \textbf{if} \( y \) \textit{is not visited} \textbf{then}
          \begin{itemize}
            \item add edge \((x, y)\) to \( T \);
            \item Mark \( y \) as visited and enqueue \((Q, y)\);
          \end{itemize}
      \end{itemize}
    \end{itemize}
\end{itemize}

Proposition \( \text{BFS}(u) \) runs in \( O(m + n) \) time.
BFS: An Example

- **BFS tree** is the set of black edges.
Breadth First Search with Distances

Algorithm: \textbf{BFS}(u):

- Initialize queue \( Q \) to be empty;
- Mark all vertices as unvisited; set \( \text{dist}(v) = \infty \) for each \( v \);
- Initialize search tree \( T \) to be empty;
- Mark \( u \) as visited and \text{enqueue}(Q, u); \text{dist}(u) = 0;

\textbf{while} \( Q \) \text{ is not empty} \textbf{ do}

\( x = \text{dequeue}(Q); \)

\textbf{foreach} vertex \( y \in \text{Adj}(x) \) \textbf{ do}

\textbf{if} \( y \) \text{ is not visited} \textbf{ then}

- add edge \((x, y)\) to \( T \);
- Mark \( y \) as visited and \text{enqueue}(Q, y);
- \( \text{dist}(y) = \text{dist}(x) + 1; \)


## Shortest Distance

### Properties

1. If $\text{dist}(u) < \text{dist}(v)$, then $u$ is visited before $v$.  
2. If $e = (u, v)$ is an edge of $G$, then $|\text{dist}(u) - \text{dist}(v)| \leq 1$.  

The **shortest distance** $\delta(u, v)$ between two vertices $u$ and $v$ in an *unweighted* graph $G$ is the length of a shortest path (in terms of the number of edges) from $u$ to $v$.  

- no path between $u$ and $v$ means $\delta(u, v) = \infty$
Shortest Distance

Proposition

Upon termination of $\text{BFS}(u)$, for every vertex $v$, $\text{dist}(v) = \delta(u, v)$.

Proof.

Induction over the number of steps

- $\text{dist}(v) \geq \delta(u, v)$:
  - true for $v$ when we enqueue($v$) $\implies$ also true for $v$'s neighbors

- $\text{dist}(v) \leq \delta(u, v)$:
  - trivially true in the beginning
  - assume true for all $v$ with $\delta(u, v) \leq k$
  - pick any vertex $v$ with $\delta(u, v) = k + 1$
  - let $v'$ be the predecessor of $v$ on a shortest path $u$ to $v$
    $\implies \delta(u, v) = \delta(u, v') + 1$
  - $\text{dist}(v') \leq \delta(u, v')$, $|\text{dist}(v) - \text{dist}(v')| \leq 1$ $\implies \text{dist}(v) \leq \delta(u, v)$
BFS Intuition

- start with vertex $u$
- list all its neighbors (distance 1)
- list all their neighbors (distance 2)
- etc.
BFS Summary

- Runs in time $O(m + n)$ on adjacency lists.

- Visit every vertex reachable from $u$.

- Can be used to compute shortest paths from $u$ to all other vertices in unweighted graphs.
Depth First Search

- A versatile graph exploration strategy.
- Power of DFS to understand the structure of the graph is demonstrated by Hopcroft and Tarjan.
- Can be used to solve many nontrivial problems in linear time ($O(m + n)$).
  - Finding cut-edges and cut-vertices of undirected graphs.
  - Finding strong connected components of directed graphs.
  - Testing whether a graph is planar.
- Basic Graph Search Algorithm with a stack.
A stack is a linked list with two operations

- **Push**\( (S, x) \): insert an element at the front of the stack.
- **Pop**\( (S) \): remove the front element of the stack.

- Elements are processed in a **last-in first-out (LIFO)** order, different from the **first-in first-out (FIFO)** order for queues.
- Implementation: need to maintain only the pointer of the front of the stack.
- Useful to also have **Peek**\( (S) \): retrieve
**Algorithm:** DFS(u):

1. Initialize stack $S$ to be empty;
2. Mark all vertices as unvisited; set $\text{Time} = 0$;
3. Initialize search tree $T$ to be empty;
4. Mark $u$ as visited and push $(S, u)$;
5. $\text{pre}(u) = ++\text{Time}$;
6. **while** $S$ is not empty **do**
   - $x = \text{peek}(S)$;
   - **foreach** vertex $y \in \text{Adj}(x) **do**$
     - **if** $y$ is not visited **then**
       - Mark $y$ as visited and push $(S, y)$;
       - $\text{pre}(y) = ++\text{Time}$;
     - **if** there is no such $y **then**$
       - $\text{post}(x) = ++\text{Time}$;
       - pop$(S)$;