Graph Algorithms
BFS properties

- **Property.** If \((u, v)\) is an edge in \(E\) then \(\text{dist}[u]\) and \(\text{dist}[v]\) differ by at most 1

\[ |\text{dist}[u] - \text{dist}[v]| \leq 1 \]

- **Proof.** WLOG, suppose \(u\) is dequeued first
  - \(\text{dist}[u] \leq \text{dist}[v]\)  
    - Recall: if \(u\) is dequeued before \(v\) then \(\text{dist}[u] \leq \text{dist}[v]\)
  - We need to show \(\text{dist}[v] - \text{dist}[u] \leq 1\)
  - Proof by contradiction. Suppose that is not true. Therefore, \(\text{dist}[v] > \text{dist}[u] + 1\)
    - When BFS chooses to dequeue \(u\) from \(Q\), vertex \(v\) is in one of the following states:
      - **WHITE**
        - \(\text{dist}[v] = \text{dist}[u] + 1\)  
          → contradiction with the assumption that \(\text{dist}[v] > \text{dist}[u] + 1\)
      - **BLACK**
        - It was already removed from the queue which means \(\text{dist}[v] \leq \text{dist}[u]\)  
          → contradiction
          - Recall: if \(v\) is dequeued before \(u\) then \(\text{dist}[v] \leq \text{dist}[u]\)
      - **GRAY**
        - It was painted gray upon dequeuing some vertex \(w\)
        - \(w\) was removed from \(Q\) before \(u\) and \(\text{dist}[v] = \text{dist}[w] + 1\)
        - \(\text{dist}[w] \leq \text{dist}[u]\) (was removed from queue before \(u\))
        - \(\text{dist}[v] = \text{dist}[w] + 1 \leq \text{dist}[u] + 1\) → contradiction with our assumption
Bipartite Graphs

- $G$ is bipartite if $V$ can be partitioned into two sets such that every edge has one end in $V_1$ and one end in $V_2$
- How to test if a graph is bipartite?
  - Similarly, how to test if a graph is 2-colorable
Bipartite Graphs

Is the following graph bipartite (2-colorable)?
Bipartite Graphs

Is the following graph bipartite (2-colorable)?
Bipartite Graphs

- G is bipartite if V can be partitioned into two sets such that every edge has one end in $V_1$ and one end in $V_2$.
- How to test if a graph is bipartite?
Checking if a graph is bipartite

- Run BFS \((G, s)\) algorithm

\[
V_1 = \{ v \in V | \text{dist}(s, v) \text{ is even} \}
\]

\[
V_2 = \{ v \in V | \text{dist}(s, v) \text{ is odd} \}
\]

- For each edge \((u, v) \in E\) check whether \((u \in V_1 \text{ and } v \in V_2)\) or \((v \in V_1 \text{ and } u \in V_2)\)
  - If YES, then G is bipartite
  - If No, then G is not bipartite
Checking if a graph is bipartite: proof of correctness

- If the algorithm determines that a graph is not bipartite, then there is an edge \((u, v)\) with both ends in one partition: \(u, v \in V_1\) or \(u, v \in V_2\)
  - Recall: for any edge \((u, v)\), \(\text{dist}(u)\) and \(\text{dist}(v)\) differs by 1 or 0
  - If \(\text{dist}(u)\) and \(\text{dist}(v)\) differ by 1, then one vertex is in \(V_1\) and the other is in \(V_2\) which cannot be true in this case
  - Therefore \(u\) and \(v\) have the same distance to \(s\)
- Now consider the BFS tree of \(T\) and let \(w\) be the lowest common ancestor of \(u\) and \(v\) in \(T\)
  - \(P_{uw} = P_{wv} = K\)
  - Length of cycle formed by \(P_{uw}, P_{wv}\), and \((u,v)\) is:
    - \(2K+1 \rightarrow \) it is an odd cycle
Bipartite Graphs

- A graph is bipartite if and only if it does not contain an odd length cycle.
  - The proof is via an algorithm that finds a bipartition OR an odd cycle.
Depth-First Search (DFS)
Depth-First Search (DFS)

- **Input:** A graph $G = (V, E)$ and a vertex $s$
- **Output:**
  - Explores all the nodes in the graph reachable from $s$
  - Generates a tree rooted at $s$
- **Algorithm:**
  - It starts from $s$, and follows the first path it finds and goes as deep as possible
  - During the execution of the algorithm, vertices are in one of the three following states:
    - UnDiscovered (white)
    - Discovered (gray)
    - Fully explored (black)
- **Algorithm is implemented recursively**
  - Can be implemented non-recursively using a stack
DFS: Basic Version

DFS(G)
  for each vertex u in G:
    color[u] = WHITE    #Mark w as undiscovered
  for each vertex u in G:
    if color[u] == WHITE    #if u is undiscovered
      DFS(G, u)

DFS(G, s)
  color[s] = GRAY    #discover s
  for each neighbor v of s:
    if (color[v] == WHITE)    #v is undiscovered
      DFS(G, v)
  color[v] = BLACK    #Mark s as fully-discovered(BLACK)
DFS Example
DFS Example
DFS Example
DFS Example
DFS Example
DFS Example
DFS Example
DFS Example
DFS Example
DFS Example
DFS Example
DFS Example
DFS Example

![Graph Diagram]

- **u** -> **v**
- **v** -> **w**
- **x** -> **y**
- **y** -> **z**
- **B**
- **F**

In a DFS traversal, the nodes are visited in the order shown by the arrows.
DFS Example
DFS Example

Diagram showing a graph with nodes labeled U, V, W, X, Y, and Z, and edges connecting them in a specific order.
DFS: Basic Version

\[ DFS(G) \]
\[
\text{for each vertex } u \text{ in } G:\n\]
\[ \text{color}[u] = \text{WHITE} \quad \# \text{Mark } w \text{ as undiscovered} \]
\[
\text{for each vertex } u \text{ in } G:\n\]
\[ \text{if } \text{color}[u] == \text{WHITE} \quad \# \text{if } u \text{ is undiscovered} \]
\[ \text{DFS}(G, u) \]
\[
\text{DFS}(G, s) \]
\[ \text{color}[s] = \text{GRAY} \quad \# \text{discover } s \]
\[
\text{for each neighbor } v \text{ of } s:\n\]
\[ \text{if } (\text{color}[v] == \text{WHITE}) \quad \# v \text{ is undiscovered} \]
\[ \text{DFS}(v) \]
\[ \text{color}[v] = \text{BLACK} \quad \# \text{Mark } s \text{ as fully-discovered(BLACK)} \]

\[ \Theta(V) \] in total:
- The loop iterates \( \deg(u) \) times for each node \( s \)
- In total the runtime of for loop is \( \Theta(\sum\deg(s)) = \Theta(E) \)
DFS Application
DFS: Proof of correctness of DFS(G, s)

Lemma 1. There is a path from $s$ to $v$ if and only if $v$ is discovered by DFS

- $\iff$ if $v$ is discovered, then there is a path from $s$ to $v$
  - **proof**: by induction on the number of vertices discovered
    - **Base case**: $s$ is discovered
    - **Induction hypothesis**: Statement is true for the first $i-1$ vertices discovered
    - **Induction step**: Now, we need to prove it is true after $i$ vertices are discovered. When the $i$-th vertex is discovered, it must have had a parent node that is already discovered. By induction hypothesis there is a path from $s$ to the parent node (since it is one of those $i-1$ vertices). There is an edge connecting vertex $i-1$ and $i$. Therefore, there is a path from $s$ to $i$

- $\implies$ if there is a path from $s$ to $v$, then $v$ is discovered
  - In other words, if $v$ is not discovered then there is no path from $s$ to $v$
DFS vs. BFS

● Similarities
  ○ Both BFS and DFS could be used to
    - Check graph connectivity
    - Find the connected components containing s
      - Both start at an arbitrary node and explores the whole connected component
    - Check s-t connectivity
  ○ Both BFS(G, s) and DFS(G, s) generate a tree rooted at s

● Differences
  ○ BFS(G, s) finds the shortest path from s to all vertices reachable from s, but DFS(G, s) does not
  ○ BFS is implemented using a queue, but not DFS
DFS Tree / Forest

- Similar to BFS, we can construct a DFS tree
  - It could be used to find the path from $s$ to all the reachable vertices
- When a vertex $v$ is first discovered when exploring vertex $u$, we say vertex $u$ is the parent of vertex $v$
- The edges $(v, \text{parent}(v))$ form a tree, and we can use them to find a path to $s$.
- A graph could have many different DFS trees depending on the order of exploring the neighbors of vertices
DFS Example

The same graph on the left: re-drawn to specify different classes of edges: B: Backward, F: Forward, C: Cross, and tree edges.
DFS: classification of edges in the graph

- DFS classifies the edges of the input Graph $G=(V, E)$.
- In a directed graph, an edge $(u, v)$ is in one of the following classes:
  - **Tree edge**
    - If it is in the depth-first forest generated by the algorithm
  - **Back edge**
    - If $v$ is ancestor of $u$ in a depth-first tree
  - **Forward edge**
    - If $v$ is a descendant of $u$ in a depth-first tree
  - **Cross edges**
    - If it is not in any of the above category. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees
Discovery and finishing time

- We modify the algorithm to record the time when a vertex is first visited and the time when its exploring is finished
- These information will be very useful in design and analysis of algorithms
DFS: Enhanced Version

\[
\text{time} = 0 \\
\text{DFS}(G) \\
\quad \text{for each vertex } u \text{ in } G: \\
\qquad \text{Color}[u] = \text{WHITE} \\
\qquad \text{Pred}[u] = \text{NULL} \\
\quad \text{time} = 0 \\
\quad \text{for each vertex } u \text{ in } G: \\
\qquad \text{if color}[u] == \text{WHITE} \\
\qquad \quad \text{DFS}(G, u) \\
\]

\[
\text{DFS}(G, u) \\
\quad \text{time} = \text{time} + 1 \\
\quad \text{discover}[u] = \text{time} \\
\quad \text{color}[u] = \text{GRAY} \\
\quad \text{for each neighbor } v \text{ of } u \\
\qquad \text{if (color}[v] == \text{WHITE}) \\
\qquad \quad \text{pred}[v] = u \\
\qquad \quad \text{DFS}(G, v) \\
\quad \text{color}[u] = \text{BLACK} \\
\quad \text{time} = \text{time} + 1 \\
\quad \text{finish}[u] = \text{time} \\
\]

Enhanced DFS

- keeps track of whether a node is undiscovered, discovered, fully explored.
- keeps track of the time it started and finished with a vertex

Output:

- a depth-first forest
- Timestamp each vertex:
  - d[u], when vertex u was discovered
  - f[u]: when u was fully explored (all its neighbors are discovered)
DFS Example
DFS Example
DFS Example
DFS Example

![DFS Example Diagram]
DFS Example
DFS Example
DFS Example

```
DFS Example

u → v → w
x → y → z
```

DFS Example
DFS Example
DFS Example

```
1/8 -> 2/7
4/5 -> 3/6
U -> V
F -> B
```

```
W -> Z
```

DFS Example

Graph:

- Nodes: U, V, W, X, Y, Z
- Edges:
  - U to V
  - V to X
  - X to U
  - V to Y
  - Y to V
  - W to Z
  - Z to W
  - U to F
  - F to B
  - B to U
  - V to 2/7
  - 2/7 to 3/6
  - 3/6 to 4/5
  - 4/5 to 1/8
  - W to 9/
DFS Example
DFS Example

The diagram illustrates a Directed Acyclic Graph (DAG) with the following nodes: U, V, X, Y, W, and Z. The edges between the nodes are represented with arrows, indicating the direction of the connection. The labels on the edges include fractions and numbers, which may represent costs or weights.

- U is connected to V with an edge labeled 1/8.
- X is connected to Y with an edge labeled 4/5.
- V has an edge to Y labeled 2/7, and another to W labeled 3/6.
- W has an edge to Z labeled 9/.
- Y has an edge to Z labeled 10/.
- There is also a dotted edge from W back to V labeled C.

The diagram shows a possible depth-first search (DFS) traversal path, starting from a root node and exploring as far as possible along each branch before backtracking. The traversal path and order of visitation are critical in understanding the flow and connectivity of the graph.
DFS Example

Diagram showing nodes and edges labeled with fractions.
DFS Example
DFS Example

The diagram illustrates a Depth-First Search (DFS) example with nodes labeled as U, V, W, X, Y, and Z. The edges and node labels are as follows:

- U -> V
- V -> W
- W -> X
- X -> Y
- Y -> Z
- Z -> W

The labels on the edges are fractions indicating the order of traversal:

- 1/8 (U to V)
- 2/7 (V to W)
- 3/6 (X to Y)
- 4/5 (X to U)
- 9/12 (W to Z)
- 10/11 (Z to W)
DFS properties: Parenthesis theorem

- In any DFS of a graph G, for any two vertices u and v, the intervals \([d[u], f[u]]\) and \([d[v], f[v]]\) are either nested or disjoint
  - If two intervals overlap, then one is nested within the other: no partial overlap
  - Vertex corresponding to smaller interval is a descendant of the vertex corresponding to the larger one
DFS properties

- Why is it called *parentheses* theorem
  - discovery time of a vertex: (
  - finish time of a vertex: )
  - The history of discoveries and finishing times makes a well-formed expression
    - Parenthesis are properly nested
DFS Properties

- Vertex \( v \) is a proper descendant of vertex \( u \) in the depth-first forest of graph \( G \) if and only if \( d[u] < d[v] < f[v] < f[u] \)
  - Follows from parenthesis theorem
DFS: Classification of edges: **directed** graph

- When we first explore an edge \((u, v)\), the color of vertex \(v\) determine edge class
- **WHITE**: a tree edge
- **GRAY**: a back edge
- **BLACK**: a forward or cross edge
  - \(d[u] < d[v]\)
    - Forward edge
  - \(d[u] > d[v]\)
    - Cross edge
DFS: Classification of edges: **undirected** graph

- In a DFS of an undirected graph G, every edge of the tree:
  - Tree edge
  - Back edge
- There are no cross or forward edges
DFS: Classification of edges: **undirected** graph

- In a DFS of an undirected graph $G$, every edge of the tree:
  - Tree edge
  - Back edge
- There are no cross or forward edges
- Proof. Let $(u, v)$ be an edge in $G$ and assume WLOG $d[u] < d[v]$
  - $d[u] < d[v]$  
    - $u$ becomes gray first, vertex $v$ is finished before $u$
    - Once $u$ becomes grays there will be two possibilities
      - The neighbor $v$ is discovered $\rightarrow$ $(u, v)$ becomes a tree edge
      - Some other neighbor $w$ will be discovered and that neighbor discovers $v$ $\rightarrow$ $(u, v)$ becomes a back edge
DFS: Classification of edges: undirected graph
DFS: Classification of edges: **undirected** graph

1. **Case 1**
   - $(u,v)$: tree edge

2. **Case 2**
   - $(u,v)$: back edge
Cut vertices and Cut Edges

- **Definitions**
  - A vertex $v$ is a cut vertex if $G - v$ is not a connected graph.
  - An edge $e$ is a cut edge if $G - e$ is not a connected graph.

- **Example:**
  - Cut vertices: b, e
  - Cut edges: (e, f)
Cut vertices and Cut Edges

- Problem. Designing an algorithm to identify cut vertices
  - **Input:** Graph $G = (V, E)$
  - **Output:** identify all cut vertices and cut edges of $G$

- Brute-force solution:
  - for all $v$ in $V$
    - Compute $G - \{v\}$, check whether $G - \{v\}$ connected or not
  - Time: $O(n \ (n+m) )$
  - $n=|V|$
  - $m=|E|$
Identifying cut vertices: Ideas and observations

**Claim.** The root is a cut vertex if and only if it has more than one child.
Identifying cut vertices: Ideas and observations

- **Lemma.** A non-root $v$ is a cut vertex if and only if $v$ has a subtree $T$ with no non-tree edge going to a proper ancestor of $v$.

- **Proof.**
  - $\Leftarrow$ removing $v$ separates $T$ from rest of graph.
  - $\Rightarrow$ since removing $v$ disconnects $G$, some subtree must get disconnected.

- **Lemma.** A non-root $v$ is a NOT cut vertex if and only if all subtrees below $v$ have an edge that goes above $v$. 

---

[Diagram showing a tree structure with vertices and edges illustrating the concepts.]
Algorithm

- **Idea:** need to check for each subtree rooted at $v$, how far up we can go?

  \[
  \text{low}(v) = \min \left\{ \text{start}[v], \text{start}[w] \mid \text{uw is a back edge with } u \text{ being descendant of } v \text{ or } u=v \right\}
  \]

  $\text{low}(v)$ records how far we can go from the subtree rooted at $v$.
Identify cut vertices using low[]

- To check whether vertex v (non-root) is a cut vertex, we check whether $\text{low}[u_i] < \text{start}[v]$ for all children $u_i$ of v
  - If yes, then v is not a cut vertex
  - If not, then v is a cut vertex and the subtree rooted at $u_i$ is a connected component
- The root vertex is handled separately
Computing the low[] in linear time

To have an efficient implementation, we process the DFS tree using a bottom-up ordering

- **Base case:**
  - v is a leaf
  - $\text{low}[v] = \min \{\text{start}[w] | vw \in E\}$
  - Time = $O(\text{deg}(v))$

- **Induction step:**
  - By I.H, computed $\text{low}(u_i)$ correctly
  - $\text{low}[v] = \min \{\text{low}[u_1], \text{low}[u_2], \text{low}[u_3], \min \{\text{start}[w] | vw \in E \} \}$
  - Time = $O(\text{deg}(v))$

Total time: $O(n + \sum_{v \in V} \text{deg}(v))$
Application of DFS

Detecting cycles in directed graphs

Lemma. A directed graph has a (directed) cycle iff DFS has a back edge.