CSE 5311 - Design and Analysis of Algorithms

Introduction to Graphs

Alex Dillhoff

University of Texas at Arlington

What are graphs?

What are graphs?

A **graph** is a data structure that is used to represent pairwise relationships between objects.

Graphs are used in many applications, such as social networks, maps, and routing algorithms.

What are graphs?

Before moving to applications, we need to define some terminology.

Definitions

A **directed graph** G is represented as a pair (V, E) of a set of vertices V and edges E.

Edges are represented as ordered pairs.

Definitions

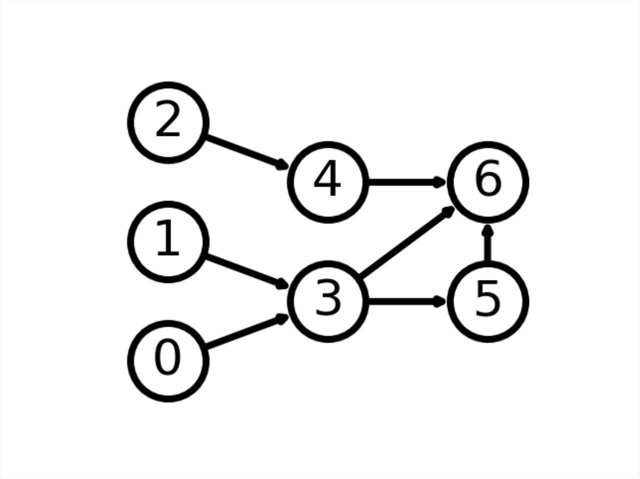


Figure 1: A directed graph with 7 vertices and 7 edges.

Definitions

An **undirected graph** G is represented as a pair (V, E) of a set of vertices V and edges E.

The edges are represented as unordered pairs, as it does not matter which direction the edge is going.

Definitions

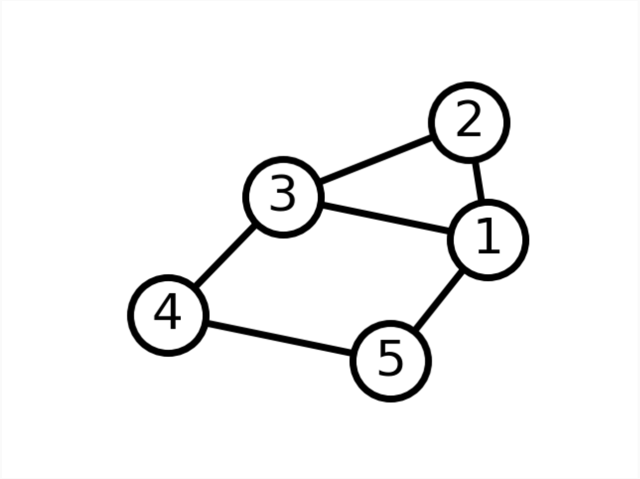


Figure 2: An undirected graph with 5 vertices and 6 edges.

Definitions

Let (u, v) be an edge in a graph G.

If G is a directed graph, then the edge is **incident from** u and is **incident to** v.

In this case, v is also **adjacent** to u.

If G is an undirected graph, then the edge is **incident on** u and v.

For undirected graphs, the **adjacency** relation is symmetric.

Definitions

The **degree** is a graph is the number of edges incident on a vertex.

For directed graphs, the **in-degree** is the number of edges incident to a vertex, and the **out-degree** is the number of edges incident from a vertex.

Definitions

A **path** from a vertex *u* to another vertex *v* is a sequence of edges that starts at *u* and ends at *v*. This definition can include duplicates.

A **simple path** is a path that does not repeat any vertices.

A **cycle** is a path that starts and ends at the same vertex.

If a path exists from *u* to *v*, then *u* is **reachable** from *v*.

Definitions

A **connected graph** is a graph where there is a path between every pair of vertices.

A **strongly connected graph** is a directed graph where there is a path between every pair of vertices.

The **connected components** of a graph are the subgraphs in which each pair of nodes is connected by a path.

In image processing, connected-component labeling is used to find regions of connected pixels in a binary image.

Definitions

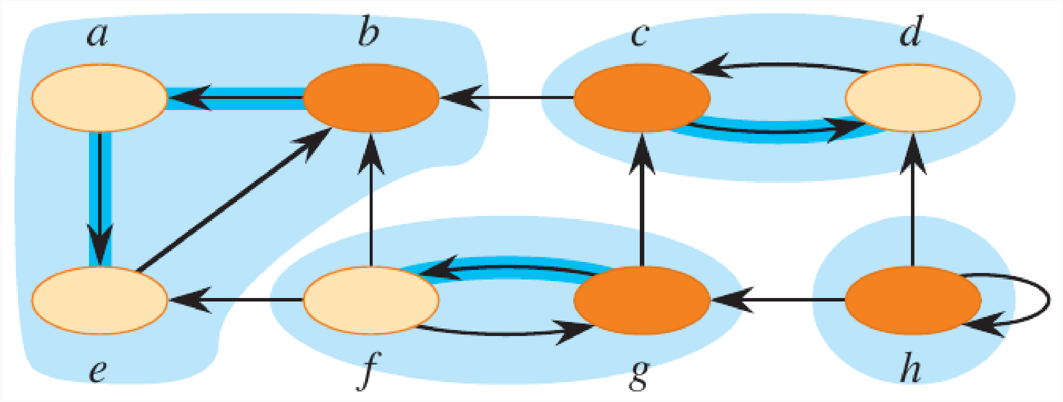


Figure 3: An example of connected components from *Introduction to Algorithms*.

Definitions

Isomorphism

Let $G = (V, E)$ and $G' = (V', E')$. G and G' are **isomorphic** if there is a bijection between their vertices such that $(u, v) \in E$ if and only if $(f(u), f(v)) \in E'$.

Definitions

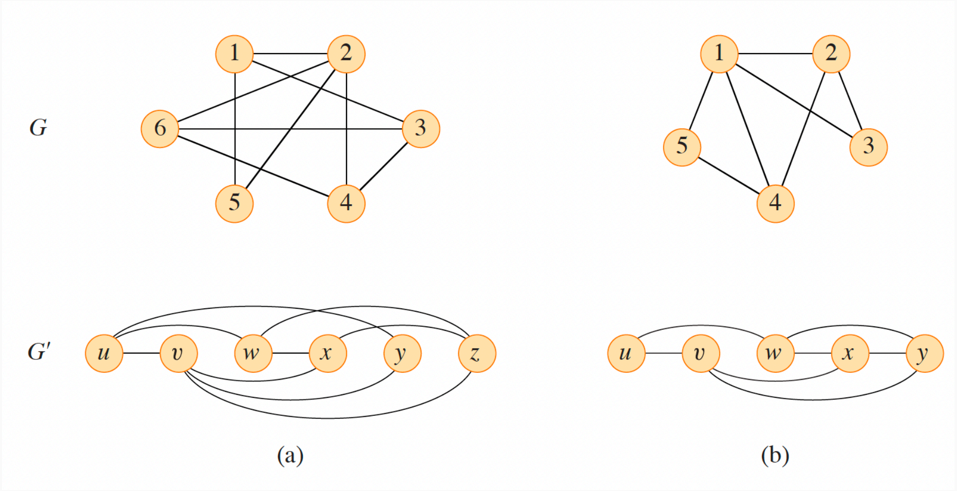


Figure 4: A pair of isomorphic graphs from *Introduction to Algorithms*.

Definitions

A **complete graph** is an undirected graph in which every pair of vertices is adjacent.

A **bipartite graph** is an undirected graph in which the vertices can be partitioned into two sets such that every edge connects a vertex in one set to a vertex in the other set.

Definitions

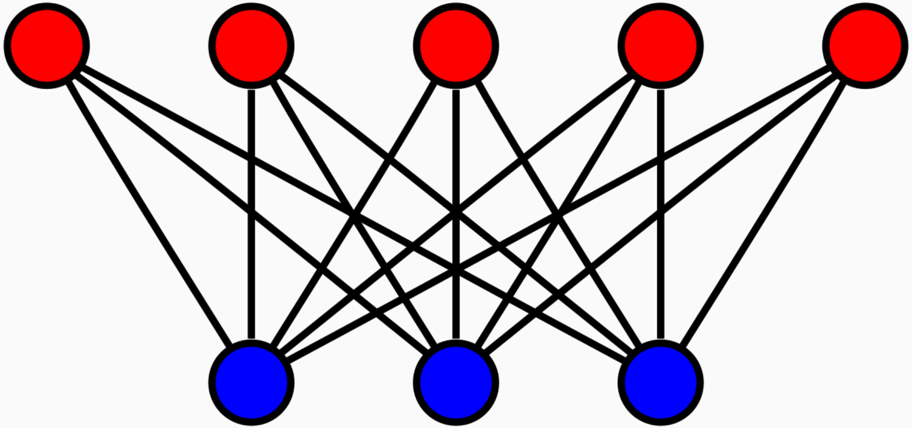


Figure 5: A bipartite graph with 4 vertices and 4 edges (Wikipedia).

Definitions

A **multi-graph** is a graph that allows multiple edges between the same pair of vertices.

These are commonly in social network analysis, where multiple edges between two people can represent different types of relationships.

Representing Graphs

Representing Graphs

Graphs can be represented in many different ways.

The most common representations are adjacency lists and adjacency matrices.

Adjacency lists are more space-efficient for sparse graphs, while adjacency matrices are more space-efficient for dense graphs.

Representing Graphs

Adjacency lists are also more efficient for finding the neighbors of a vertex.

A couple representations for adjacency lists are:

- A an array of linked lists, where each vertex has a linked list of its neighbors.
- A hash table that maps each vertex to an array of its neighbors.

Representing Graphs

Adjacency matrices are more efficient for checking if an edge exists between two vertices.

An adjacency matrix is formed by creating a $|V| \times |V|$ matrix, where $|V|$ is the number of vertices in the graph.

Each entry in the matrix is a boolean value that indicates whether an edge exists between the two vertices.

Representing Graphs

Consider the graph in the following figure. How would you represent it as an adjacency matrix?

Representing Graphs

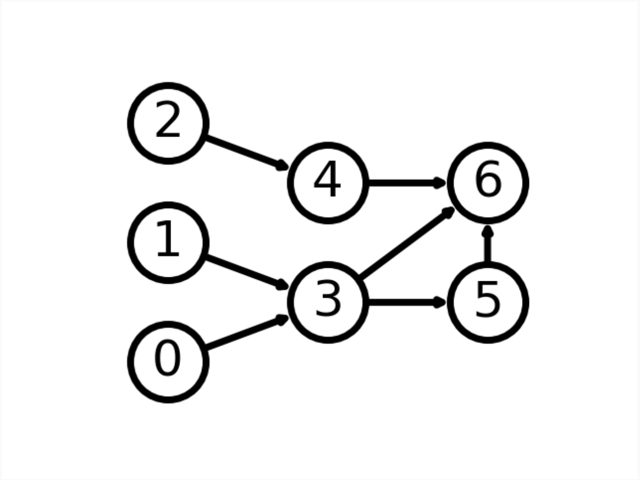


Figure 6: A directed graph.

Representing Graphs

The adjacency matrix for this graph is:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Representing Graphs

Let A be the adjacency matrix for a graph G.

The matrix A^k represents the number of paths of length k between each pair of vertices.

For example, what is A^2?

Representing Graphs

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The value at row *i* and column *j* is the number of paths of length 2 from vertex *i* to vertex *j*.

For example, is a path from vertex 0 to vertex 6 via $0 \to 3 \to 6$.

Representing Graphs

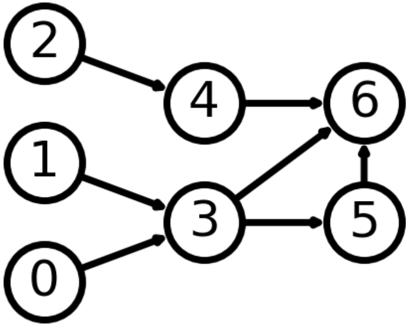


Figure 7: A directed graph.

Breadth-First Search

Breadth-First Search

We previously studied breadth-first search in the context of binary search trees.

The algorithm is the same when applied on general graphs, but our perspective is slightly different now.

The function studied before did not use node coloring.

Let’s investigate the algorithm given by Cormen et al. in *Introduction to Algorithms*.

Breadth-First Search

The algorithm adds a color to each node to keep track of its state. The colors are:

- WHITE: The node has not been discovered yet.
- GRAY: The node has been discovered, but not all of its neighbors have been discovered.
- BLACK: The node has been discovered, and all of its neighbors have been discovered.

Breadth-First Search

First, every vertex is painted white and the distance is set to ∞.

The first node `s` is immediately set to have 0 distance.

The queue then starts with `s`.

While there are any grey vertices, dequeue the next available node and add its adjacent vertices to the queue.

Breadth-First Search

The distance of each adjacent vertex is set to the distance of the current vertex plus one.

Once all of its neighbors have been discovered, the current vertex is painted black.

Breadth-First Search

```
def bfs(G, s):
    for u in G.V:
        u.color = WHITE
        u.d = inf
        u.pi = None
    s.color = GRAY
    s.d = 0
    s.pi = None
    Q = Queue()
    Q.enqueue(s)
    while not Q.empty():
        u = Q.dequeue()
        for v in G.adj[u]:
            if v.color == WHITE:
                v.color = GRAY
                v.d = u.d + 1
                v.pi = u
                Q.enqueue(v)
        u.color = BLACK
```

Breadth-First Search

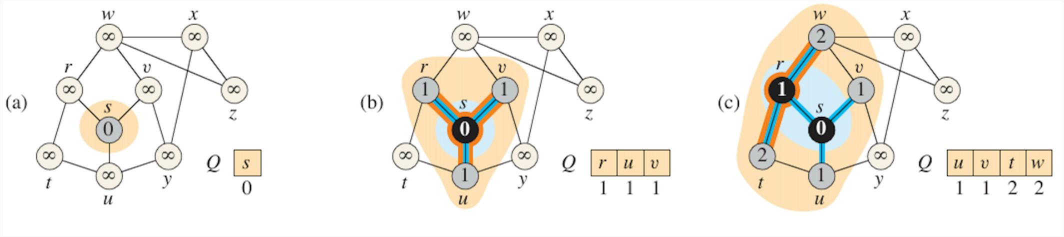


Figure 8: An example of BFS from *Introduction to Algorithms*

.

Breadth-First Search

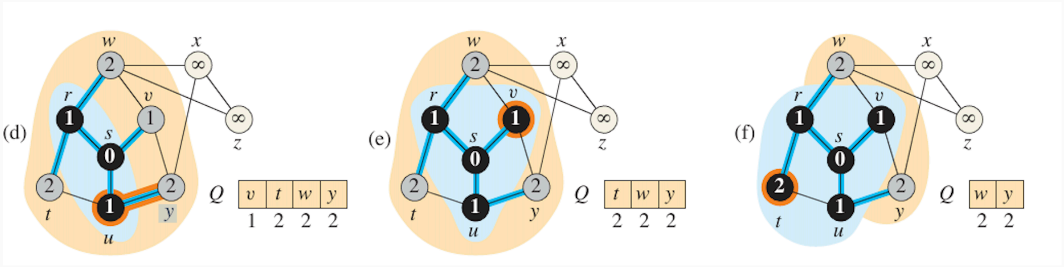


Figure 9: An example of BFS from *Introduction to Algorithms*

.

Breadth-First Search

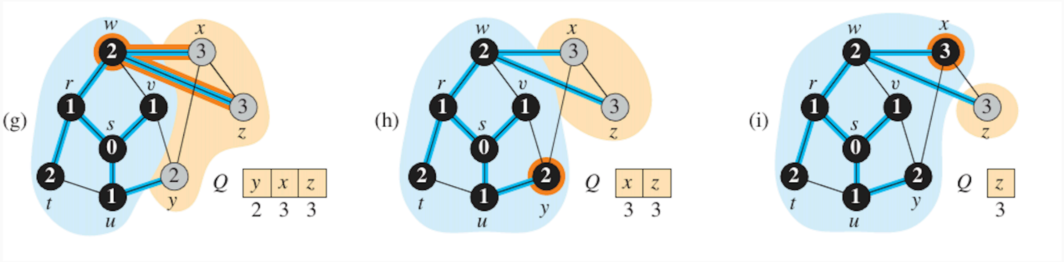


Figure 10: An example of BFS from *Introduction to Algorithms*

.

Breadth-First Search

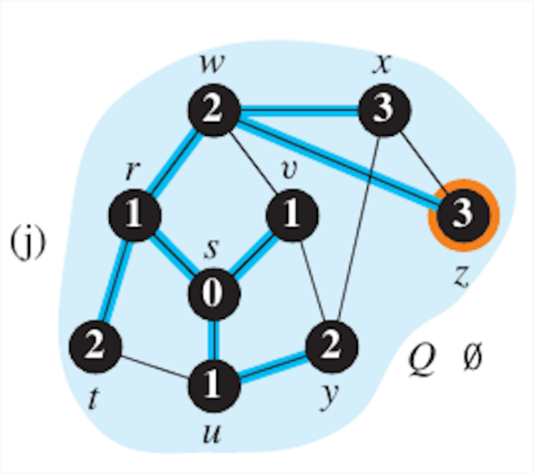


Figure 11: Final step of BFS from *Introduction to Algorithms*

BFS Analysis

The running time of BFS is $O(V+E)$, where V is the number of vertices and E is the number of edges.

Each vertex is queued and dequeued once, so the queue operations take $O(V)$ time.

Each edge is examined once, so the edge operations take $O(E)$ time.

The total running time is $O(V+E)$.

Breadth-First Trees

- The blue lines in the previous example depict a **breadth-first tree** which was built by BFS.
- The tree is defined by the π values updated throughout the course of the algorithm.
- These can also be used to reconstruct the shortest path from s to any other vertex v.

Breadth-First Trees

Breadth-first trees can be defined from a **predecessor subgraph**.

Breadth-First Trees

Breadth-first trees can be defined from a **predecessor subgraph**.

A **predecessor subgraph** is a graph $G_\pi = (V_\pi, E_\pi)$, where

$$V_\pi = \{v \in V : v.\pi \neq \text{None}\} \cup \{s\} \text{ and}$$
$$E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\}.$$

Breadth-First Trees

Such a graph is a breadth-first tree if V_π consists of the vertices reachable from s.

For all $v \in V_\pi$, the subgraph G_π contains a unique simple path from s to v that is also a shortest path from s to v in G.

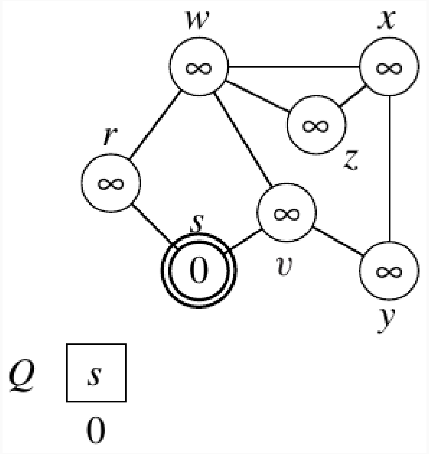
Breadth-First Trees

To print the vertices on a shortest path from *s* to *v*:

```
def print_path(G, s, v):
    if v == s:
        print(s)
    elif v.pi == None:
        print("No path from", s, "to", v, "exists.")
    else:
        print_path(G, s, v.pi)
        print(v)
```

Exercise

Run BFS on the following graph, starting from *s*.



Depth-First Search

Depth-First Search

Like the BFS algorithm presented in *Introduction to Algorithms* by Cormen et al., the DFS algorithm also uses colors to keep track of the state of each node.

The colors are similar to the BFS algorithm, but the meaning is slightly different:

- WHITE: The node has not been discovered yet.
- GRAY: The node has been visited for the first time.
- BLACK: The adjacency list of the node has been examined completely.

Depth-First Search

First, all vertices are colored white.

The time is set to 0.

The function `dfs_visit` is called on each vertex.

Depth-First Search

```
def dfs(G):
    for u in G.V:
        u.color = WHITE
        u.pi = None
    time = 0
    for u in G.V:
        if u.color == WHITE:
            dfs_visit(G, u)
```

Depth-First Search

```
def dfs_visit(G, u):
    time += 1
    u.d = time
    u.color = GRAY
    for v in G.adj[u]:
        if v.color == WHITE:
            v.pi = u
            dfs_visit(G, v)
    u.color = BLACK
    time += 1
    u.f = time
```

Depth-First Search

When a node is discovered via `dfs_visit`, the time is recorded and the color is changed to gray.

The start and finish times are useful in understanding the structure of the graph.

After all of the node's neighbors have been discovered, the color is changed to black and the finish time is recorded.

That is, the depth from the current node must be fully explored before it is considered finished.

Depth-First Search

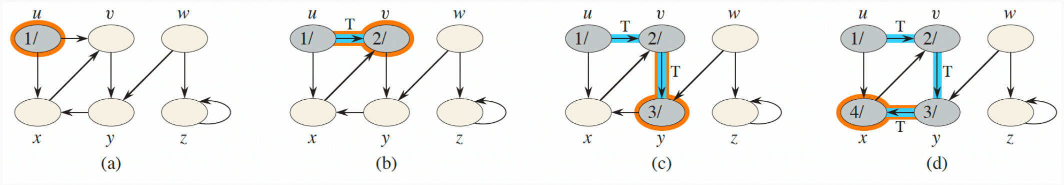


Figure 12: An example of DFS from *Introduction to Algorithms*

.

Depth-First Search

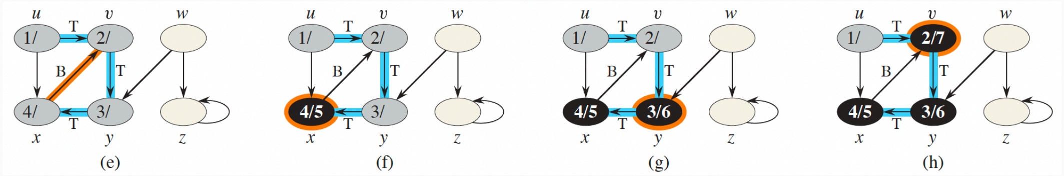


Figure 13: An example of DFS from *Introduction to Algorithms*

.

Depth-First Search

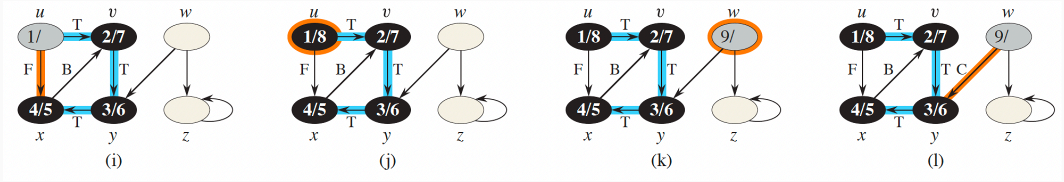


Figure 14: An example of DFS from *Introduction to Algorithms*

.

Depth-First Search

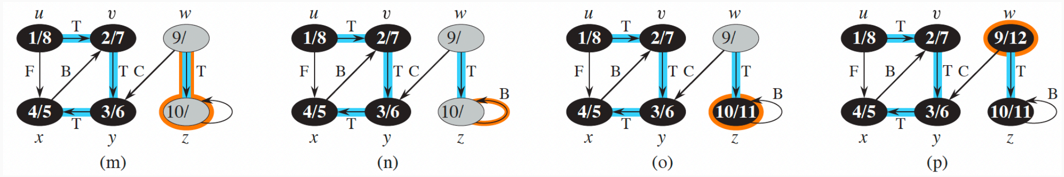


Figure 15: An example of DFS from *Introduction to Algorithms*

.

Depth-First Search

Class Exercise: Analyze the Running Time

Properties of DFS

The predecessor subgraph G_π is a forest of trees.

It creates a collection of depth first trees.

Properties of DFS

A vertex $u = v.\pi$ if and only if DFS-VISIT(G, v) was called during a search of u's adjacency list.

Vertex v is a descendant of vertex u in the depth-first forest if and only if v is discovered during the time in which u is gray.

Properties of DFS

Each call to `dfs_visit` from `dfs` finds a new tree.

Exercise: Identify the forest in the previous graph.

Parenthesis Theorem

In a DFS, the discovery and finish times have **parenthesis structure**. For all u, v, exactly only of the following holds:

1. the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint and neither u nor v is a descendant of the other in the depth-first forest.

Parenthesis Theorem

In a DFS, the discovery and finish times have **parenthesis structure**. For all u, v, exactly only of the following holds:

1. the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint and neither u nor v is a descendant of the other in the depth-first forest.
2. the interval $[u.d, u.f]$ is entirely contained within the interval $[v.d, v.f]$ and u is a descendant of v in the depth-first forest.

Parenthesis Theorem

In a DFS, the discovery and finish times have **parenthesis structure**. For all u, v, exactly only of the following holds:

1. the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint and neither u nor v is a descendant of the other in the depth-first forest.
2. the interval $[u.d, u.f]$ is entirely contained within the interval $[v.d, v.f]$ and u is a descendant of v in the depth-first forest.
3. the interval $[v.d, v.f]$ is entirely contained within the interval $[u.d, u.f]$ and v is a descendant of u in the depth-first forest.

Parenthesis Theorem

It is called the **parenthesis theorem** because if `dfs_visit` printed ”(u” when it first encountered u and printed ”u)” when it finished, the expression would be well formed.

Parenthesis Theorem

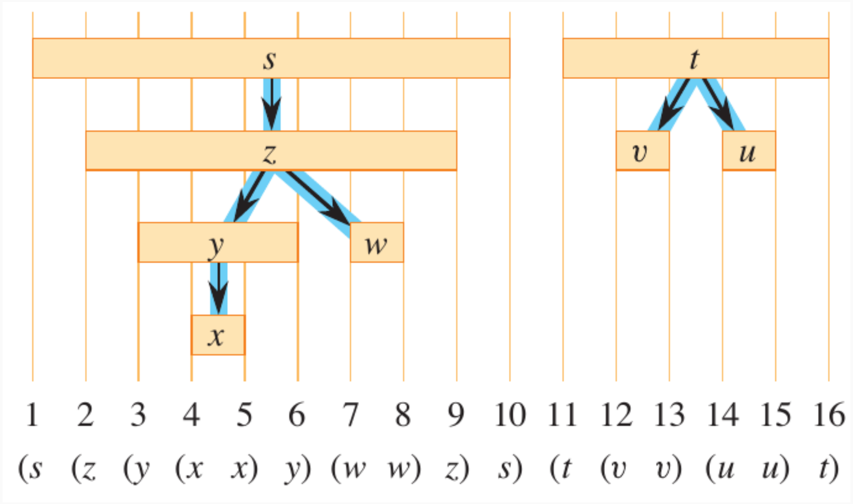


Figure 16: An example of the parenthesis theorem from *Introduction to Algorithms*

White-Path Theorem

White-path theorem

In a depth-first forest of a graph G, vertex v is a descendant of vertex u if and only if at the time $u.d$ that `dfs_visit` is called on u, there is a path of white vertices from u to v in G.

More on DFS Forests

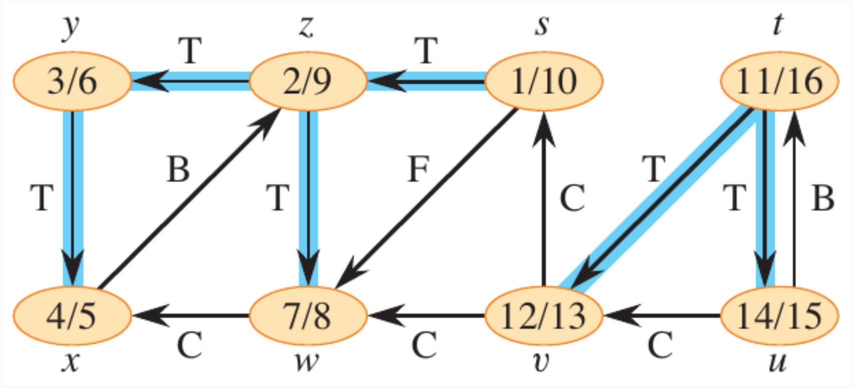


Figure 17: A completed DFS with edges labeled *Introduction to Algorithms*

More on DFS Forests

The edges are labeled as either

1. **tree edges**: edges in the depth-first forest.
2. **back edges**: edges that point from a vertex to an ancestor in the depth-first forest.
3. **forward edges**: edges that point from a vertex to a descendant in the depth-first forest.

More on DFS Forests

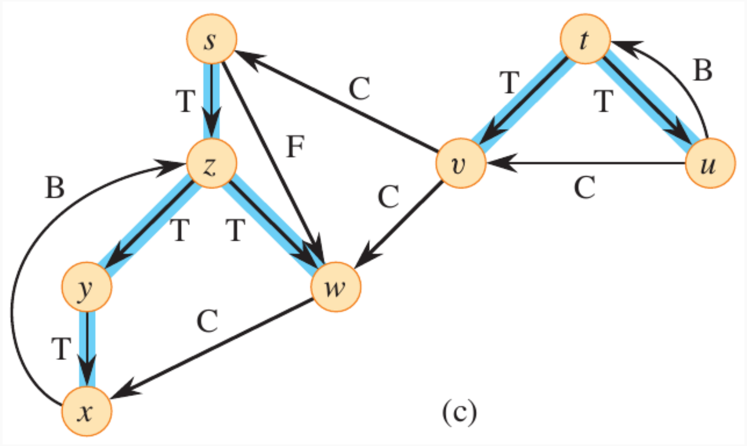


Figure 18: An example of a DFS forest from *Introduction to Algorithms*