Algorithms and Data Structures

Christian J. Rudder

October 2024

Contents

This page is left intentionally blank.

— 1 —

Graphs and Trees

1.1 Paths and Connectivity

Graphs are similar to train networks or airline routes. They connect one location to another.

Definition 1.1: Graph

A **graph** is a collection of points, called **vertices** or **nodes**, connected by lines, called **edges**. Similarly to how a polygon has vertices connected by edges.

Definition 1.2: Undirected Graph

An **undirected graph** is a graph where the edges have no particular direction going both ways between nodes. A **degree** of a node is the number of edges connected to it.

Example: Figure (1.1) shows an undirected graph:

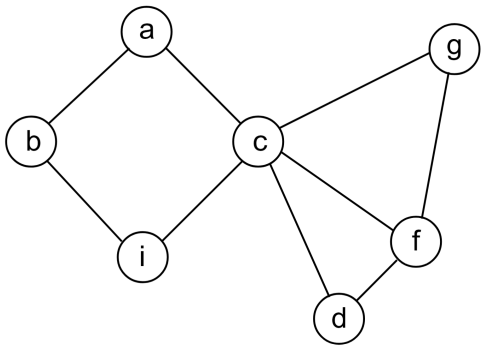


Figure 1.1: An undirected graph with 7 vertices and 9 edges.

Node a has a degree of 2, and node c has a degree of 5.

Definition 1.3: Directed Graph

A **directed graph** is where the edges have a specific direction from one node to another.

- The **indegree** of a node is the number of edges that point to it.
- The **outdegree** of a node is the number of edges that point from it.

Example: Figure (1.2) shows a directed graph:

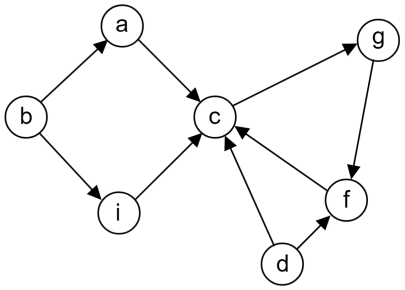


Figure 1.2: A directed graph with 7 vertices and 9 edges.

Node b has an outdegree of 2 and an indegree of 0. c has an indegree of 4 and an outdegree of 1.

Definition 1.4: Weighted Graph

A **weighted graph** is a graph where each edge has a numerical value assigned to it.

Example: Figure (1.3) shows a weighted graph:

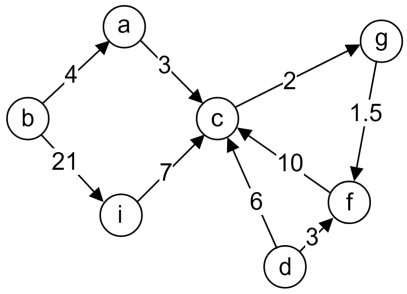


Figure 1.3: A weighted graph with 7 vertices and 9 edges.

Definition 1.5: Path

A **path** is a sequence of edges that connect a sequence of vertices. A path is **simple** if all nodes are distinct.

Example: In Figure (1.4), a simple path $h \leftrightarrow b \leftrightarrow i \leftrightarrow c \leftrightarrow d$ is shown:

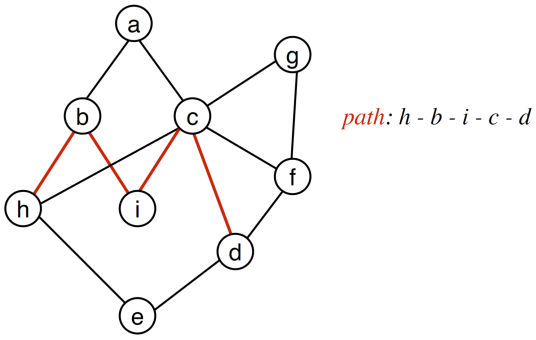


Figure 1.4: A graph with a simple path from h to d.

Definition 1.6: Connectivity

A graph is **connected** if there is a path between every pair of vertices.
A graph is **disconnected** if there are two vertices with no path between them.
Connected graphs of n nodes have at least $n - 1$ edges.

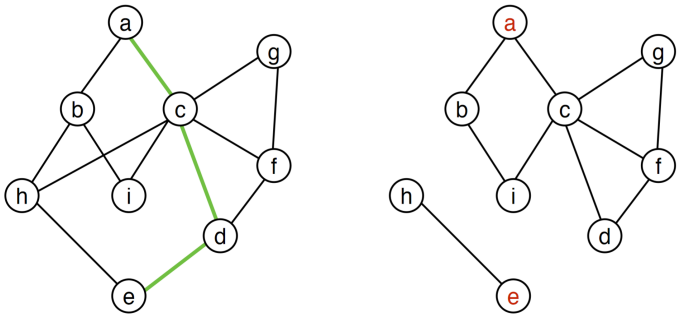


Figure 1.5: A connected graph $a \leftrightarrow c \leftrightarrow d \leftrightarrow e$ and disconnected graph.

Definition 1.7: Adjacency Matrix

An **adjacency matrix** is an $n \times n$ matrix where such that $A[i][j] = 1$ if there is an edge between nodes i and j.

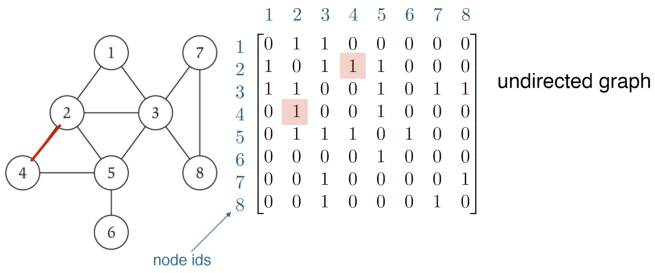


Figure 1.6: An adjacency matrix where the path $4 \leftrightarrow 2$ is highlighted ($A[2][4]$ or $A[4][2]$)

Theorem 1.1: Properties of Adjacency Matrix

The following properties hold for adjacency matrices:

- An undirected graph is symmetric about the diagonal.
- A directed graph is not symmetric about.
- A weighted graph has the weight of the edge instead of binary.

Space Complexity: $\Theta(n^2)$; **Time Complexities:**
Index edge $A[i][j]$**:** $\Theta(1)$; **List neighbors** $A[i]$: $\Theta(n)$; **List all edges:** $\Theta(n^2)$.

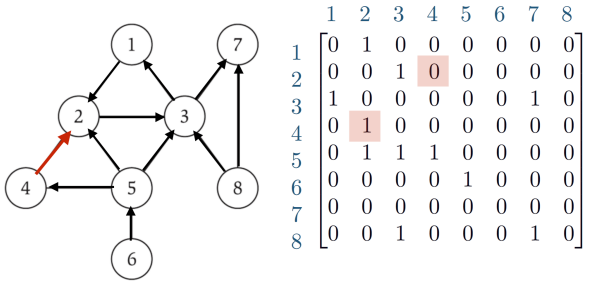


Figure 1.7: An adjacency matrix for a directed graph with path $4 \leftrightarrow 2$ highlighted ($A[2][4]$ and $A[4][2]$).

Definition 1.8: Adjacency List

An **adjacency list** is a list of keys where each key has a list of neighbors.

Often taking form as a dictionary or hash-table:
Space Complexity: $\Theta(n+m)$ for n nodes and m total edges; **Time Complexities:** **Index key:** $\Theta(1)$; **List key neighbors:** $\Theta(\#$ of outdegrees$)$; **List all edges:** $\Theta(n+m)$; **Insert edge:** $\Theta(1)$. **Note:** $m \leq n^2$ (all nodes connected to all nodes), though typically $m < n$.

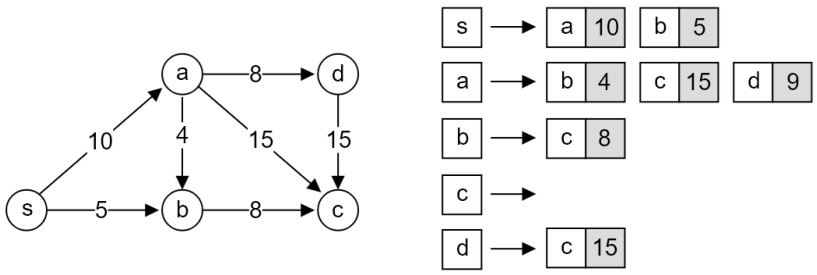


Figure 1.8: An adjacency list of a directed graph, notably c has no outdegrees.

1.2 Breath-First and Depth-First Search

High-Level Overview

Two general methods for traversing a graph are, **breadth-first search** and **depth-first search**.

Definition 2.1: Cycle

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

Example: In the above Figure (1.7), $1 \to 2 \to 3 \to 1$ form a cycle.

Definition 2.2: Tree

A **tree** is a connected graph with no cycles. A **leaf-nodes** is the outer-most nodes of a tree. A **branch** is a path from the root to a leaf.

Theorem 2.1: Tree Identity

Let G be an undirected graph of n nodes. Then any two statements imply the third:

(i.) G is connected.

(ii.) G has $n-1$ edges.

(iii.) G has no cycles.

Definition 2.3: Rooted Trees

Let T be a tree with a designated root node r. Then each degree is considered an outdegree, called a **child**, and its indegree its **parent**.

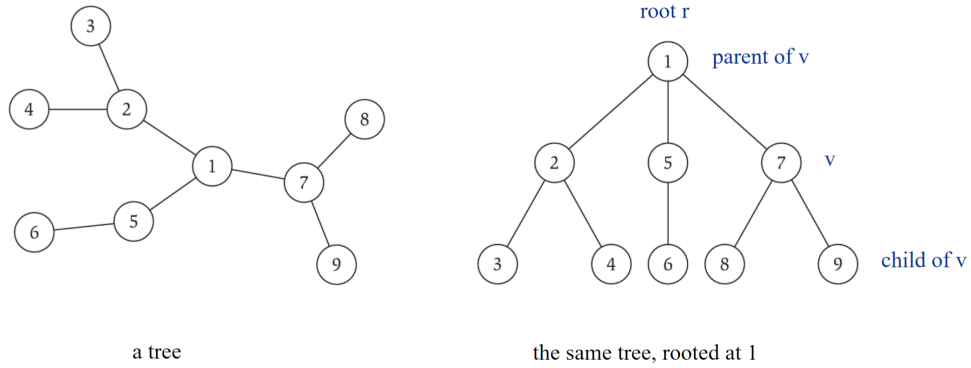


Figure 1.9: A rooted tree with root 1 and children $\{2, 5, 7\}$

Definition 2.4: Levels and Heights

The **level** of a node is the number of edges from the root. The **height** of a tree is the maximum level of any node.

Example: In Figure (1.9)'s rooted tree , node 5 has a level of 1, and the height of the tree is 2.

Definition 2.5: Breadth-First Search (BFS)

In a **breadth-first search**, we start at a node's children first before moving onto their children's children in level order.

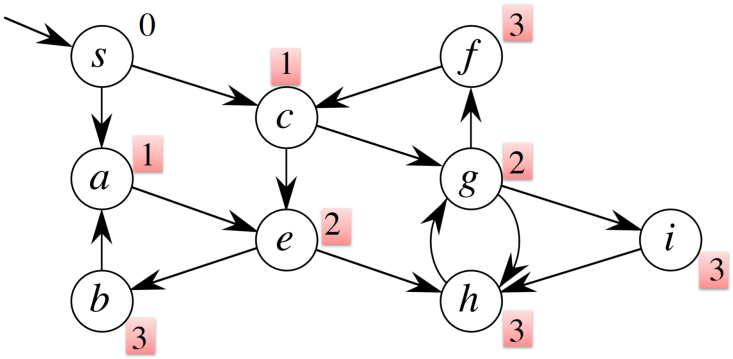


Figure 1.10: A BFS tree traversal preformed on a graph with each level enumerated

Theorem 2.2: Properties of BFS

BFS run on any graph T produces a tree T' with the following properties:

(i.) T' is a tree.

(ii.) T' is a rooted tree with the starting node as the root.

(iii.) The height of T' is the shortest path from the root to any node.

(iv.) Any sub-paths of T' are also shortest paths.

Proof 2.1: Proof of BFS

(i.) and (ii.) follow from the definition of a tree. (iii.) and (iv.) follow that since a tree contains a direct path to any given node in our parent child relationship, that path must be the shortest. ■

Tip: In a family tree, there is only one path from each ancestor to each descendant.

We create a BFS algorithm from what we know, though not the best implementation:

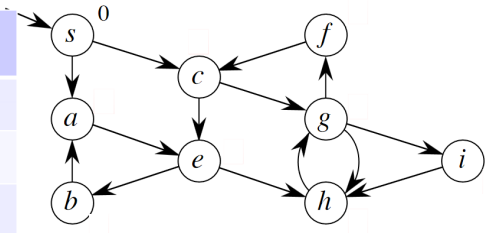
Function 2.1: BFS Algorithm - `BFS(s)`

Breadth-First Search starting from node s.

Input: Graph $G = (V, E)$ and starting node s.
Output: Levels of each vertex from s.

```
1  Function BFS(s):
2      for each v ∈ V do
3          Level[v] ← ∞;
4      Level[s] ← 0;
5      Add s to Q;
6      while Q not empty do
7          u ← Q.Dequeue();
8          for each v ∈ G[u] do
9              if Level[v] = ∞ then
10                 Add edge (u, v) to tree T (parent[v] = u);
11                 Add v to Q;
12                 Level[v] ← Level[u] + 1;
```

u	Queue contents before exploring u	… after exploring u
s	(empty)	*[a,c]*
a	*[c]*	*[c,e]*
c	*[e]*	*[e,g]*
e	*[g]*	*[g,b,h]*
g	*[b,h]*	*[b,h,f,i]*
b	*[h,f,i]*	*[h,f,i]*
h	*[f,i]*	*[f,i]*
f	*[i]*	*[i]*
i	(empty)	(empty)



Loop invariant: If the first node in the queue has level i, then the queue consists of nodes of level i possibly followed by nodes of level $i + 1$.

Consequence: Nodes are explored in increasing order of level.

Figure 1.11: A table showing the queue at each level of iteration

We analyze the time and space complexity in the below Figure (1.12):

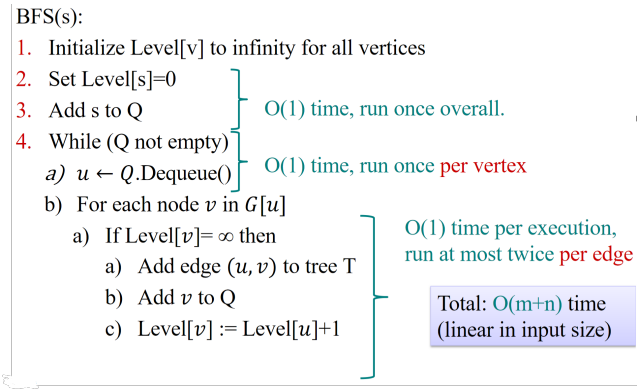


Figure 1.12: An analysis showing $O(m+n)$ for both time and space complexity

Proof 2.2: Claim 1 for BFS

Let s be the root of the BFS tree, then: **Proof:** Induction on the distance from s to u.
Base case ($u = s$): The code sets Level$[s] = 0$, and there is no path to find since the path has length 0.
Induction hypothesis: For every node u at distance $\leq i$, Claim 1 holds.
Induction step:

- Let v be a node at distance exactly $i+1$ from s. Let u be its parent in the BFS tree.
- The code sets Level$[v]$ = Level$[u] + 1$.
- Let x be the last node before v on a shortest path from s to v. Since v is at distance $i+1$, then x must be at distance i, and so Level$[x] = i$ (by induction hypothesis).
- If $u = x$, we are done!
- If $u \neq x$, then it must be that u was explored before x, since otherwise x would be the parent of u.
- Since we explore nodes in order of level, Level$[u] \leq$ Level$[x] = i$.
- If Level$[u] = i$, then we are done.
- If Level$[u] < i$, then the path $s \sim u \rightarrow v$ has length at most i, which contradicts the assumption that the distance of v is $i+1$.

We conclude that Level$[u] = i$, Level$[v] = i+1$, and the path in the BFS tree that goes from s to u to v has length $i+1$. ■

Definition 2.6: Depth-First Search (DFS)

In a **depth-first search**, we recursively explore each an entire branch before moving onto the next.

Function 2.2: DFS Algorithm - `DFS(`G`)`

Depth-First Search on graph G (recursive).

Input: Graph $G = (V, E)$.
Output: Discovery and finishing times for each vertex.

```
1  Function DFS(G):
2      for each u ∈ G do
3          u.state ← unvisited;
4      time ← 0;
5      for each u ∈ G do
6          if u.state == unvisited then
7              DFS-Visit(u);
8  Function DFS-Visit(u):
9      time ← time + 1;
10     u.d ← time ;
11     // record discovery time;
12     u.state ← discovered;
13     for each v ∈ G[u] do
14         if v.state == unvisited then
15             DFS-Visit(v);
16     u.state ← finished;
17     time ← time + 1;
18     u.f ← time;
19     // record finishing time;
```

Time and Space Complexity: $O(n + m)$ where n is the number of vertices and m the number of edges.

Proof 2.3: DFS Correctness

Case 1: When s is discovered, there is a path of unvisited vertices from s to y. We use induction on the length L of the unvisited path from x to y.

- **Base case:** There is an edge (x, y), and y is unvisited.
- **Induction hypothesis:** Assume that the claim is true for all nodes reachable via k unvisited nodes.
- **Induction step:**
  - Consider u reachable via $k + 1$ unvisited nodes.
  - Let z be the last node on the path before y, and z is discovered from x (by I.H.).
  - The edge (z, y) will be explored from x, ensuring that y is eventually visited.

Thus, DFS-Visit(x) explores all nodes reachable from x through a path of unvisited nodes, as required. ■

Edge Classifications – Directed Graphs

To build our intuition around edge classifications, we'll follow a walkthrough of DFS & BFS on directed graphs to build our intuition. We'll follow the below format:

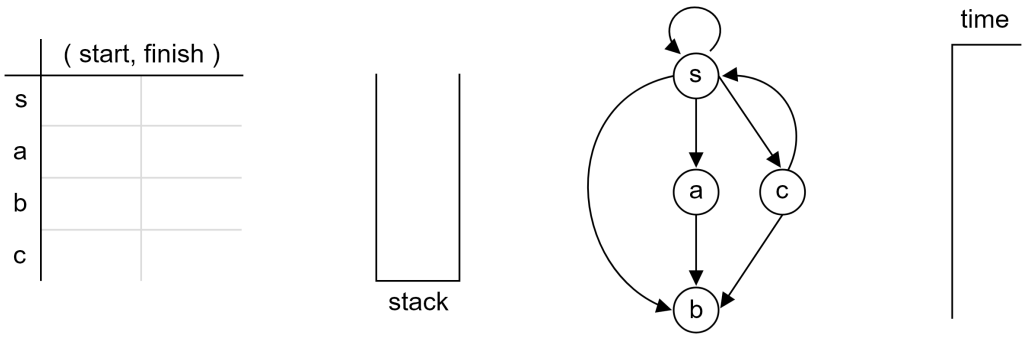


Figure 1.13: A table for start and finish times of each visited nodes, our stack to keep track of traversals, the graph that shall be traversed, and a timer for DFS (BFS will use a queue).

Let's start by filling out a partial DFS traversal:

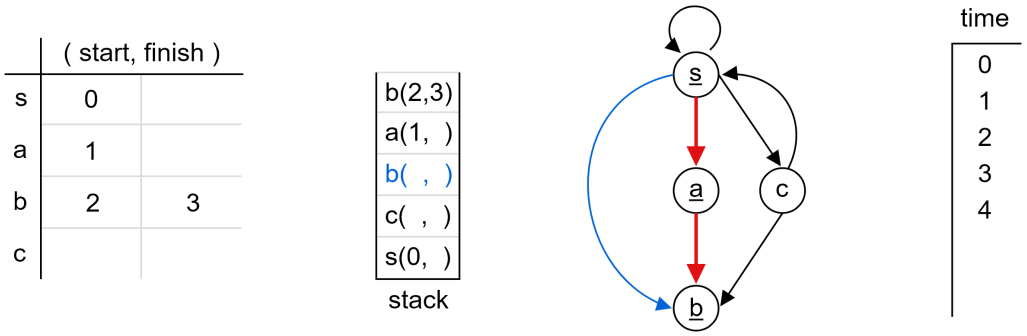


Figure 1.14: So far we have asked s to put all their children on the stack at timestamp 0. We first investigate the first child a's branch at time 1. It gives us b, for which we time stamp (2,3) for start and finish. We highlight the first the alternative b branch in blue for distinction. The edges that are in our (start, finish) table are **Tree Edges**.

Definition 2.7: Forward Edge

A **non-tree edge** that connects an already discovered node to a descendant in the DFS tree is called a **forward edge**. Concretely, (u, v) is a forward edge if given $u(s_1, f_1)$ and $v(s_2, f_2)$ of node id and a start finish time tuple, we have: $s_1 < s_2 < f_2 < f_1$.
Where the **start time** of u is less than v, and the **finish time** of u is greater than v.

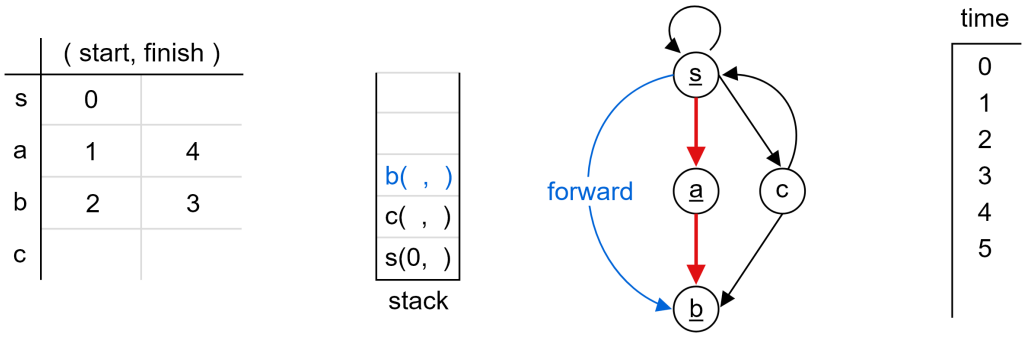


Figure 1.15: Continuing from Figure (1.14), b and a are taken off the stack. The next b on the stack is a **forward edge**, as s has a smaller start time, and a foreseeable finish time than the table's b.

The next two are **cross and back edges**:

Definition 2.8: Cross Edge

A **non-tree edge** that connects branches of a tree is called a **cross edge**. In particular, (u, v) is a cross edge if v is not a descendant or ancestor of u. Concretely, given $u(s_1, f_1)$ and $v(s_2, f_2)$ of node id and a start finish time tuple, we have: $s_2 < s_1 < f_2 < f_1$.
Where the **start and finish time** of u is greater than v.

Definition 2.9: Back Edge

A **non-tree edge** that connects a node to an ancestor in the tree is called a **back edge**. Concretely, given $u(s_1, f_1)$ and $v(s_2, f_2)$ of node id and a start finish time tuple, we have: $s_2 < s_1 < f_1 < f_2$.
Where the **start time** of u is greater than v, but the **finish time** of is less than v.

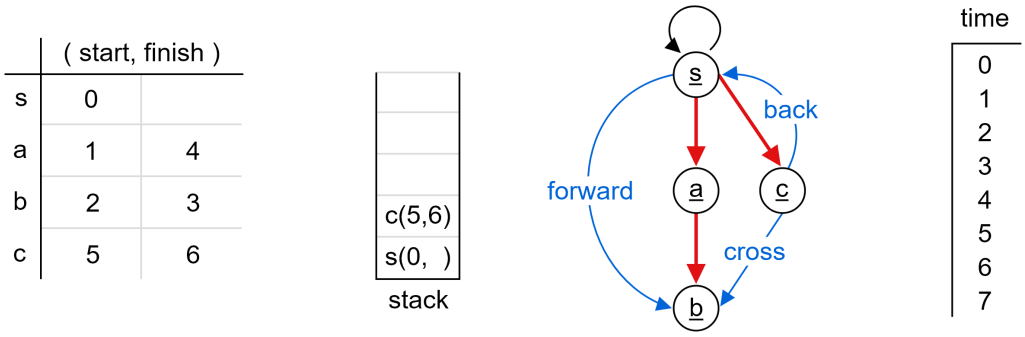


Figure 1.16: Continuing from Figure (1.15), b and a are taken off the stack. Next on the stack is c, which reveals children b and s; Here, b has already been processed (cross edge), and s discovered, but yet to be finished (back edge). There is nothing else to explore, so we finish c. **Note:** There is no classifications for self referential edges.

Next we'll see how our classifications work on BFS.

Starting with the same graph as before we instead run BFS:

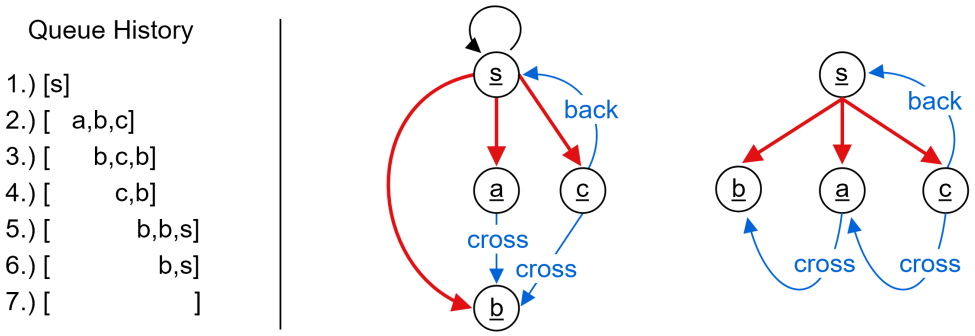


Figure 1.17: We start with s modeling the queue history on the left. The first graph is our original graph finished, and on the right it is redrawn for clarity.

Theorem 2.3: DFS & BFS Directed Graphs

Between DFS and BFS, the following holds:

- **DFS**: Tree, back, forward, and cross edges (all edges).
- **BFS**: Tree, cross, and back edges (no forward edges).

Additionally, one may find the following theorem helpful:

Theorem 2.4: DFS and Cycles

Let DFS run on graph G, then:

$$(G \text{ has a cycle}) \iff (\text{DFS run reveals back edges})$$

Proof 2.4: Proof of Cycles and Back Edges

Proving (G has a cycle) $\Longleftarrow$ (DFS run reveals back edges): Every back edge creates a cycle. **Proving** (G has a cycle) $\Longrightarrow$ (DFS run reveals back edges) Suppose G has a cycle: Let u_1 be the first discovered vertex in the cycle, and let u_k be its predecessor in the cycle. u_k will be discovered while exploring u_1. The edge (u_k, u_1) will be a back edge. ■

Next we deal with undirected graphs briefly and give a summary:

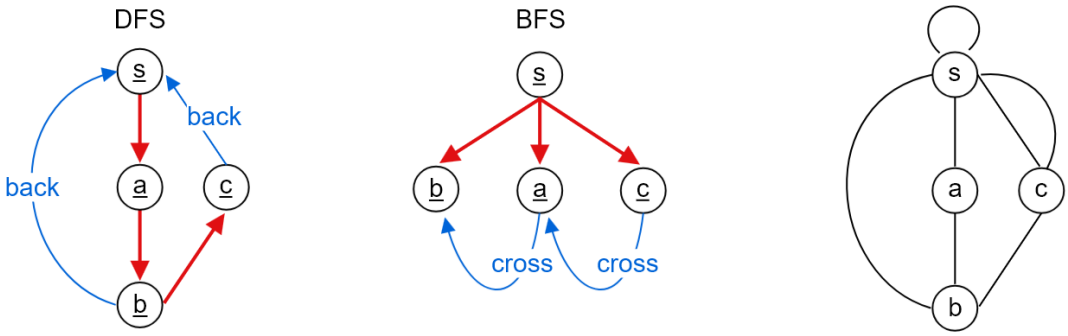


Figure 1.18: Here DFS and BFS are shown, with the original graph on the far right. Here it's quite peculiar to have a double-edge in an undirected graph. In our case it's redundant and may be counted as a single edge.

Summary:
This allows us to summarize the edge classifications for both directed and undirected graphs in the below table:

Graph Type	DFS Edge Types	BFS Edge Types
Directed Graphs	Tree, Back, Forward, Cross	Tree, Back, Cross
Undirected Graphs	Tree, Back	Tree, Cross

Table 1.1: Edge classifications for DFS and BFS in directed and undirected graphs.

Edge Type	Start Condition	Finish Condition
Forward Edge	less	greater
Cross Edge	greater	greater
Back Edge	greater	less

Table 1.2: Summary of start and finish conditions for edge types by comparing the starting node with its endpoint. E.g., in a forward edge (u, v), the start time of u is less than v, but the finish time is greater.

1.3 Directed-Acyclic Graphs & Topological Ordering

Graphs may represent a variety of relationships, such as dependencies between tasks or the flow of information.

Definition 3.1: Directed-Acyclic Graph (DAG)

A **directed-acyclic graph** is a directed graph with no cycles.

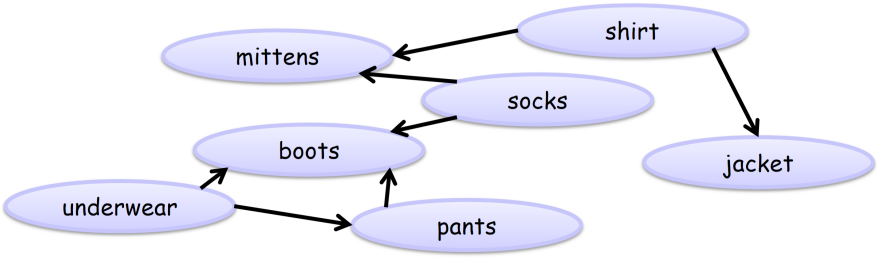


Figure 1.19: A DAG depicted by getting dressed for winter. For each node to be processed its **dependencies** or parents must be processed first.

Definition 3.2: Topological Ordering

Given a graph, a **topological ordering** is a linear ordering of nodes such that for every edge (u, v), u comes before v.

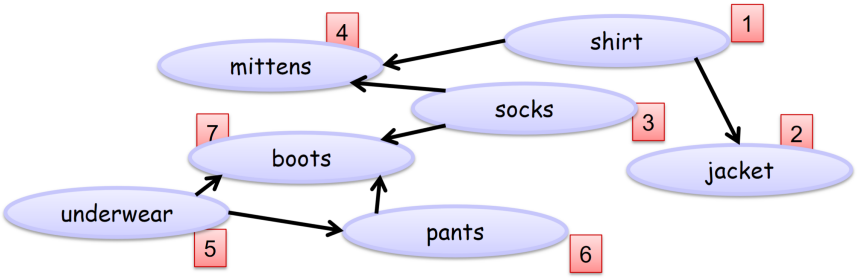


Figure 1.20: A topological ordering of the DAG in Figure (1.19) enumerated in red. Another possible ordering of $[1, 2, 3, 4, 5, 6, 7]$ is $[5, 6, 1, 2, 3, 4, 7]$, as $5 \to 6$ is independent.

Theorem 3.1: Topological Sort

Given a DAG, a topological ordering can be found.

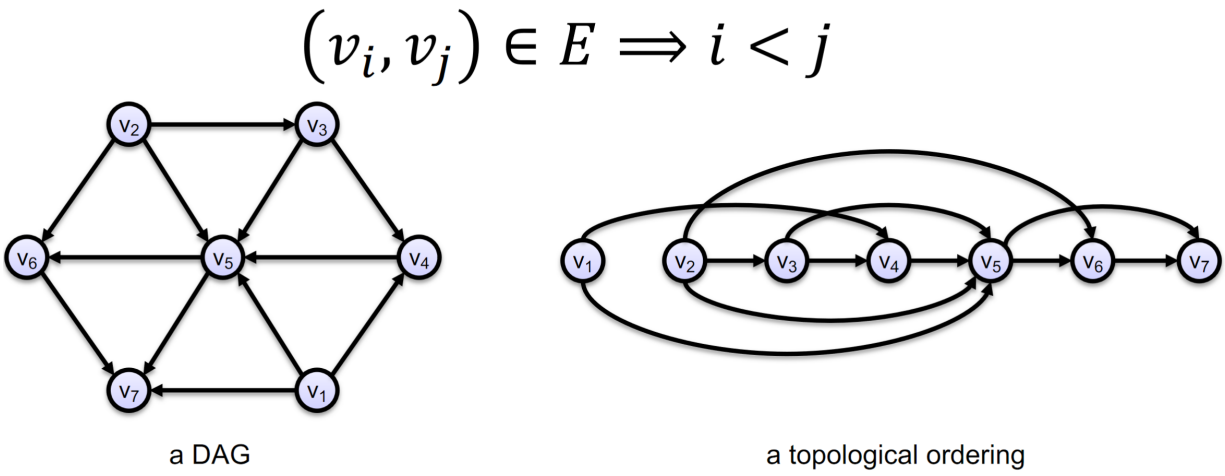


Figure 1.21: A topological sorting of a DAG E and v nodes

Proof 3.1: Topological Sort via DFS

Lemma: In a directed graph G, if (note necessarily acyclic):

- (u, v) is an edge, and
- v is not reachable from u,

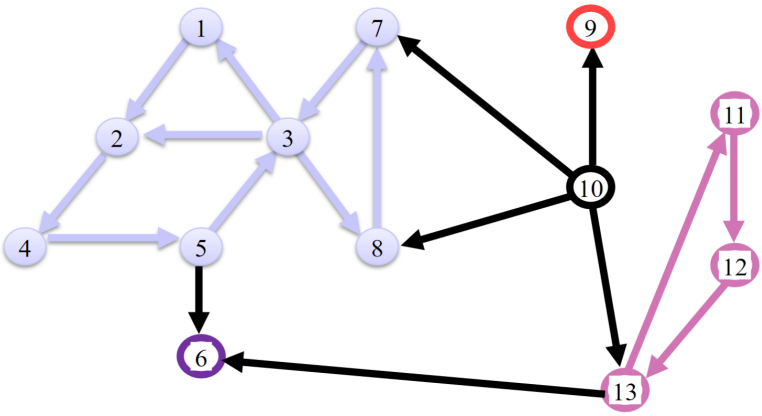
Then in every run of DFS, $u.f > v.f$.
Proof:

- If v is started before u, then the DFS-Visit(v) will terminate without reaching u (because there is no path to u).
- If u is started before v, then the edge (u, v) will be explored before u is finished.

Therefore, in all cases, $u.f > v.f$. ■

Definition 3.3: Strongly Connected Components

A **strongly connected component** is a subgraph where every node is reachable from every other node. Then we say $u \rightsquigarrow v$ and $v \rightsquigarrow u$ are **mutually reachable**.



- Observation. Two SCCs are either disjoint or equal.
- If we contract the SC components in one node we get an *acyclic* graph.

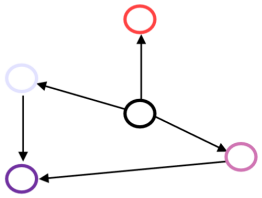
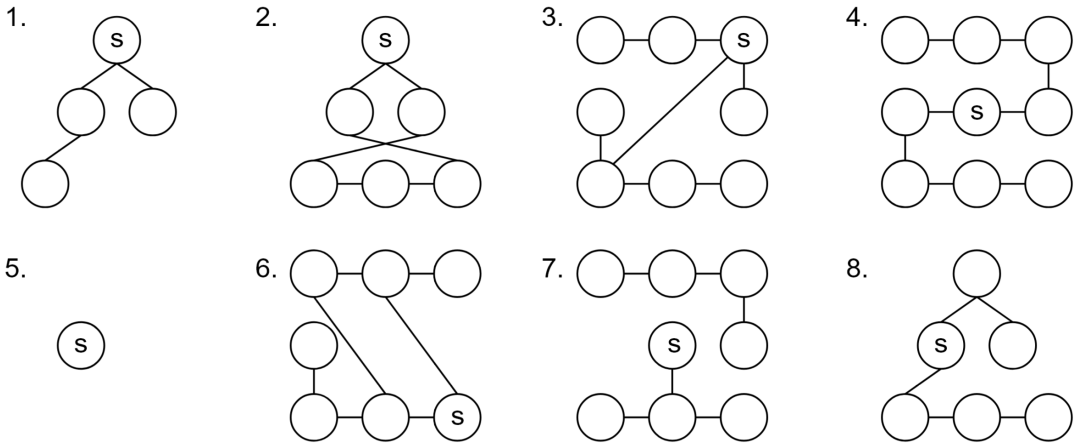


Figure 1.22: A graph with 5 strongly connected components.

Exercise 3.1: Based on the images which are trees, and if not, why?



Exercise 3.2: Based on Exercise (3.1) above, starting with the s nodes, what are possible results of BFS and DFS?

Exercise 3.3: What type of edges do BFS and DFS traversals produce on directed and undirected graphs?

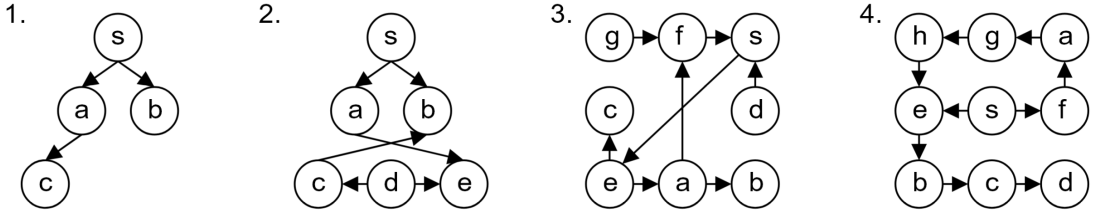
Exercise 3.4: BFS can find shortest paths, does DFS find longest paths (True/False)?

Exercise 3.5: To find a path and return such path from node s to node t in a graph, should we use BFS or DFS?

Exercise 3.6: To detect cycles in a graph, should we use BFS or DFS, or both?

Exercise 3.7: Say we wanted to find the shortest round-trip path, that is, the minimal path starting from origin node s visiting a set of nodes exactly once returning to s. How should we use BFS or DFS?

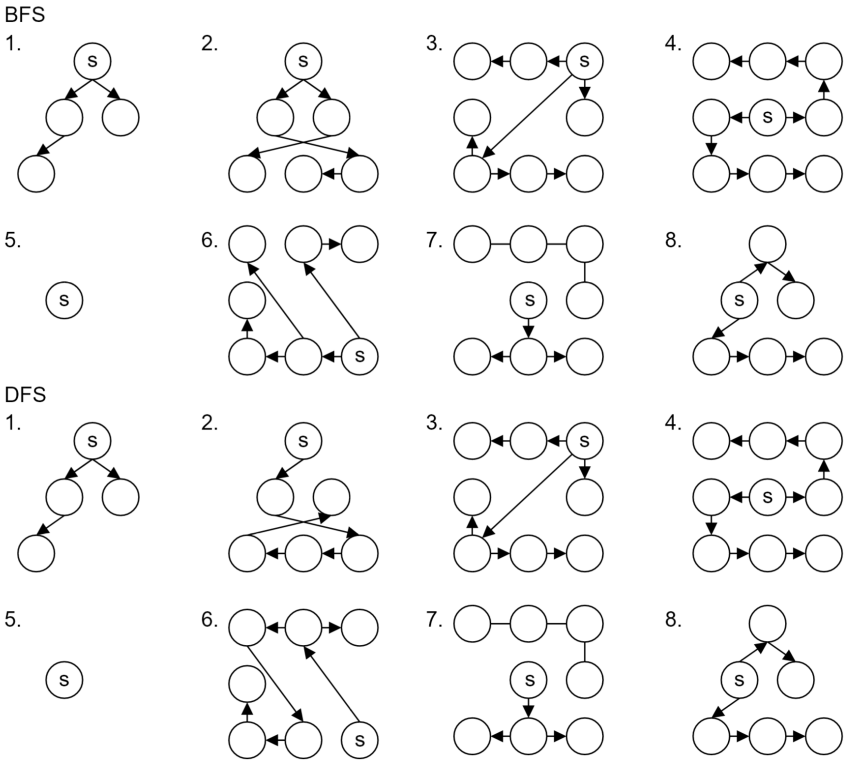
Exercise 3.8: Which of the below images are DAGs.



Exercise 3.9: Based on Exercise (3.8) above, what are possible topological orderings?

Answer 3.1: 1, 3, 4, 5, 8 are trees. 2 has a cycle, 6 has a cycle, 7 is disconnected.

Answer 3.2: There are other possible solutions depending on order of traversal. For 7, BFS and DFS don't alone handle disconnected graphs. Though modifications could be made to handle such.



Answer 3.3: Undirected - BFS: Tree edges, DFS: Tree edges, Back edges. Directed - BFS: Tree edges, Back edges, Cross edges, DFS: Tree edges, Back edges, Forward edges, Cross edges [1, 3, 2]

Answer 3.4: False, as of today, there is no known algorithm to find the longest path in a graph.

Answer 3.5: BFS, it finds shortest child-parent connections, giving us a direct path from s to t.

Answer 3.6: Both, as DFS when it encounters a node that has already been visited, it has found a cycle. BFS when two branches meet, a cycle is found.

Answer 3.7: BFS, one implementation is to run BFS from s to all nodes, the first cycle found is the shortest round-trip path.

Answer 3.8: 1, 2, and 4 are DAGs. 3 has a cycle (f,s,e,a).

Answer 3.9:

1. $[s, a, b, c]$, $[s, b, a, c]$, $[s, a, c, b]$.
2. $[s, a, d, c, b, e]$, $[d, c, s, a, e, b]$ or $[d, c, s, a, b, e]$
4. $[s, f, a, g, h, e, b, c, d]$

Bibliography

[1] Dfs, graph modeling. https://courses.cs.washington.edu/courses/cse417/21wi/lecture/06-DFS-model.pdf, 2021. Lecture 6.

[2] P.-Y. Chen. Chapter 22.1: Breadth-first search – clrs solutions. https://walkccc.me/CLRS/Chap22/Problems/22-1/, 2017–2024. Built by P.-Y. Chen, made with Material for MkDocs. Accessed: 2024-06-15.

[3] Rong Ge and Will Long (Scribe). Graph algorithms: Types of edges. https://courses.cs.duke.edu/fall17/compsci330/lecture12note.pdf, 2017. Lecture 12.