Lecture 12 — Elementary Graph Algorithms (Part 2)

COMPSCI 308 — Design and Analysis of Algorithms

Xing Shi Cai https://newptcai.gitlab.io

Duke Kunshan University, Kunshan, China

Jan-Mar 2026

Summary

Assignments[1]

Practice makes perfect!

📖 Required Readings:

- Section 20.3.
- Section 20.4.
- Section 20.5.

✏️ Required Exercises:

- Exercises 20.3 — 1–3.
- Exercises 20.4 — 1–3.
- Exercises 20.5 — 1–3.

[1] 💣 Assignments will not be collected; however, quiz problems will be selected from them. (This includes both Readings and Exercises.)

ITA 20.3 Depth-First Search

A Visual Comparison With BFS

When solving a maze, the BFS strategy explores .

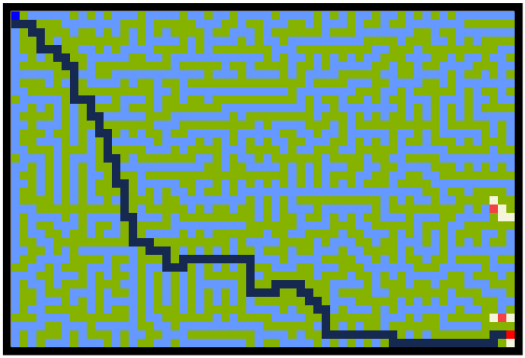


Figure 1: BFS solution

Try it out at https://brkwok.github.io/Maze-Solver/.

A Visual Comparison With bfs

When solving a maze, the DFS goes deep .

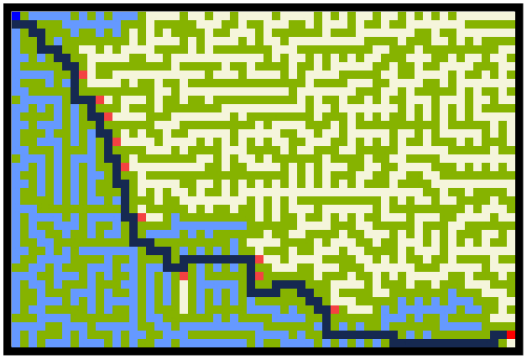


Figure 1: DFS solution

Try it out at https://brkwok.github.io/Maze-Solver/.

Example Running of dfs

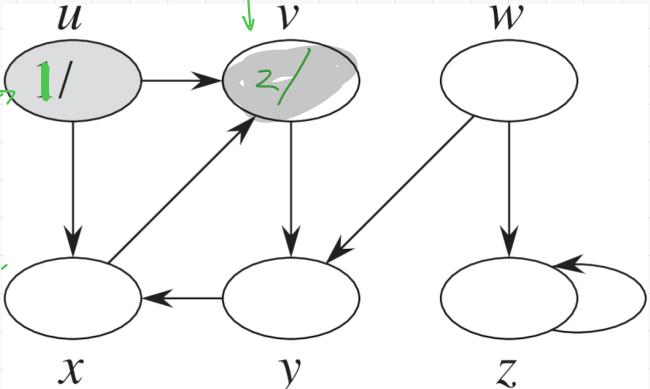


Figure 2: Running DFS

Example Running of dfs

This edge has been explored.

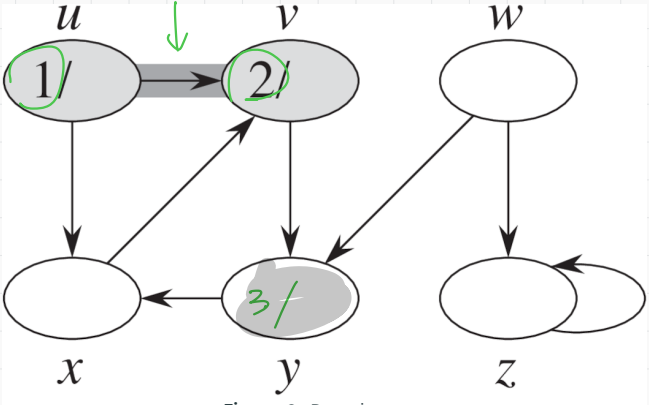


Figure 2: Running DFS

📌 Example Running of dfs

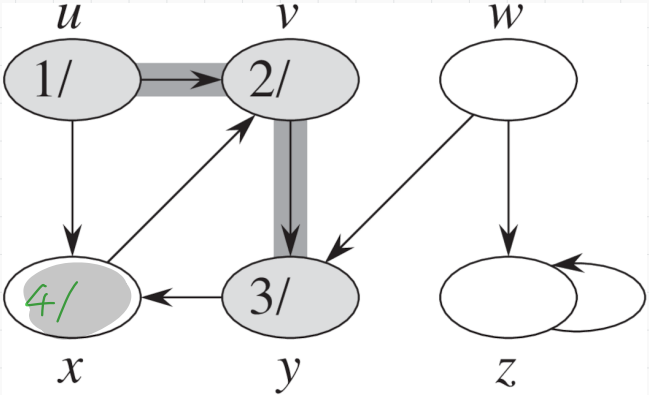


Figure 2: Running DFS

(c)

📌 Example Running of dfs

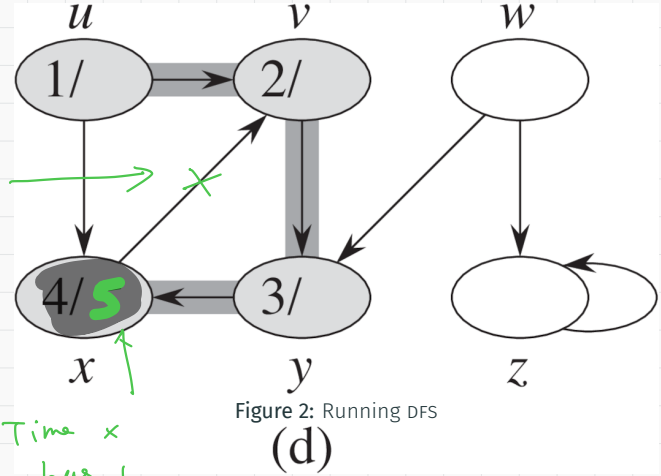


Figure 2: Running DFS

Back edge — Do not follow

Time x has been processed.

📌 Example Running of dfs

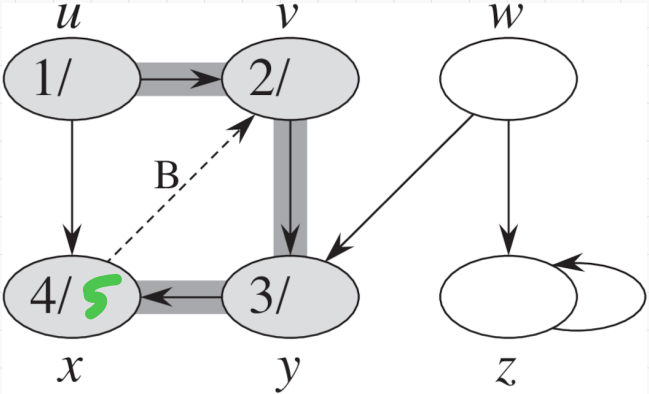


Figure 2: Running DFS

Example Running of dfs

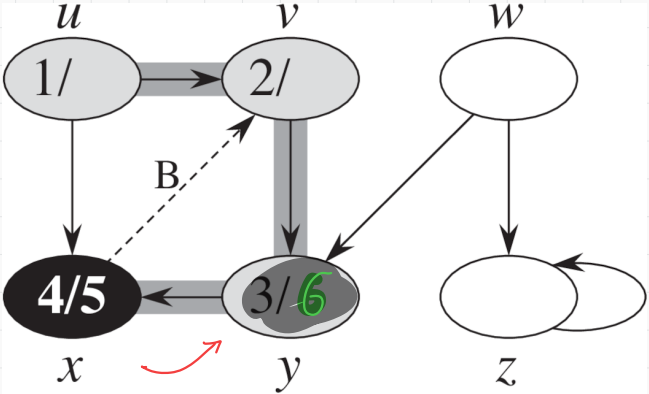


Figure 2: Running DFS

Has
to
go back

📌 Example Running of dfs

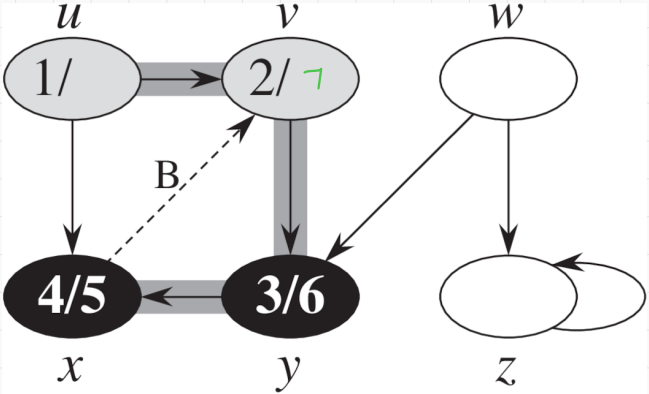


Figure 2: Running DFS

📌 Example Running of dfs

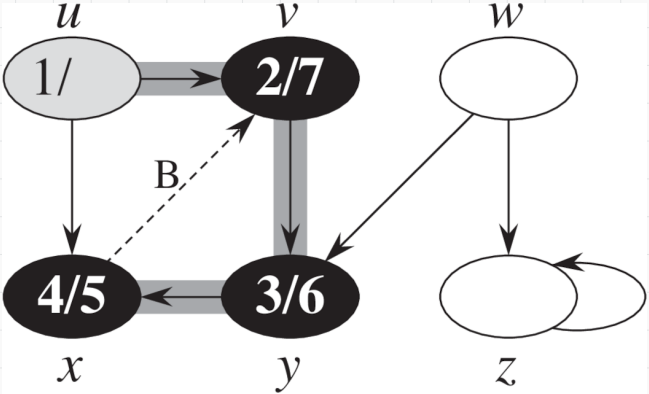


Figure 2: Running DFS

📌 Example Running of dfs

Forward edge:

Do not follow.

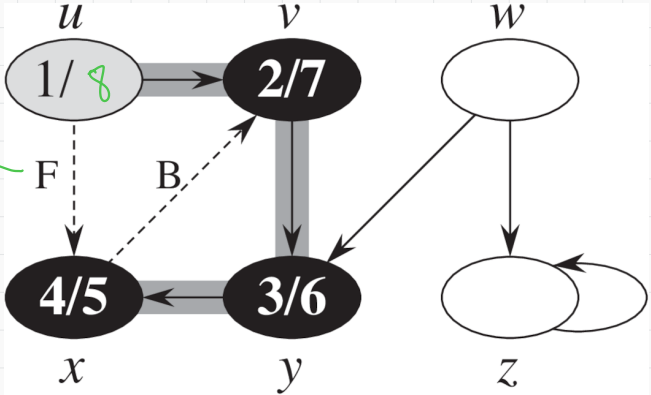


Figure 2: Running DFS

Example Running of dfs

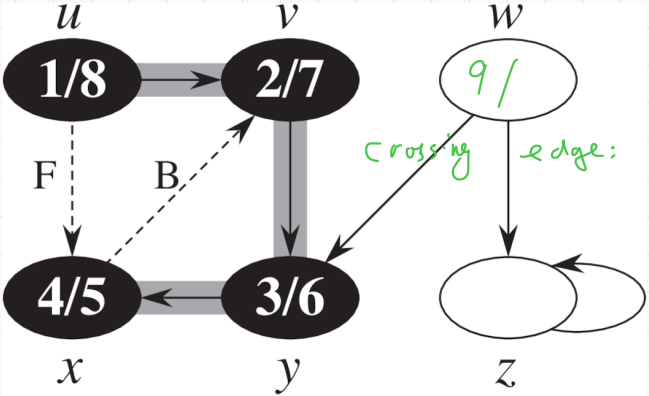


Figure 2: Running DFS

crossing edge: Not following.

📌 Example Running of dfs

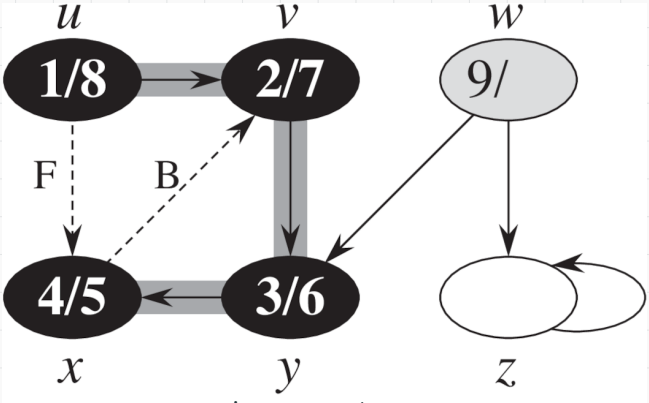


Figure 2: Running DFS

Example Running of dfs

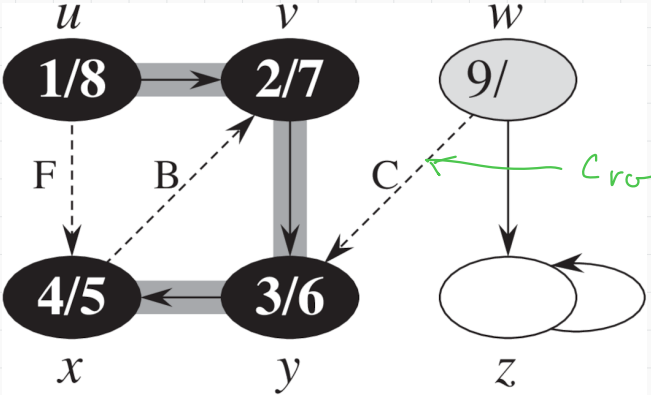


Figure 2: Running DFS

(1)

📌 Example Running of dfs

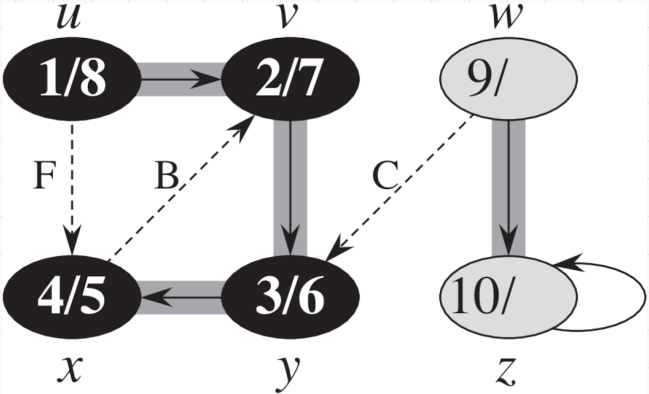


Figure 2: Running DFS

(m)

Example Running of dfs

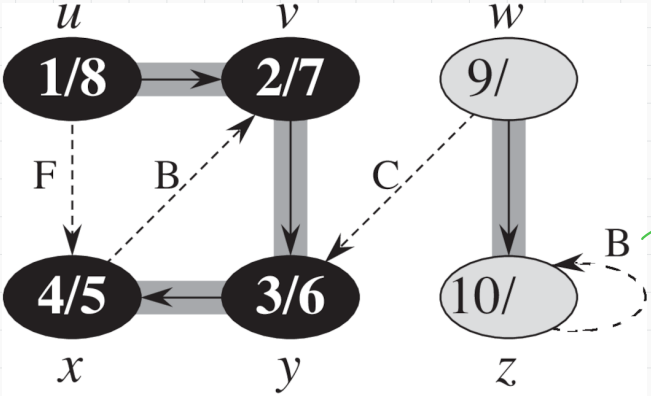


Figure 2: Running DFS

(n)

📌 Example Running of dfs

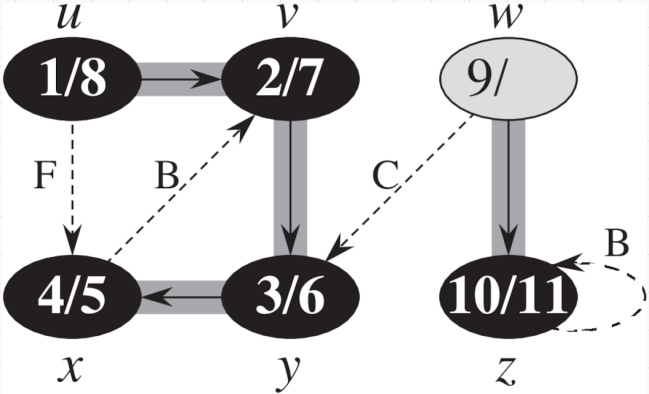


Figure 2: Running DFS

(o)

Example Running of dfs

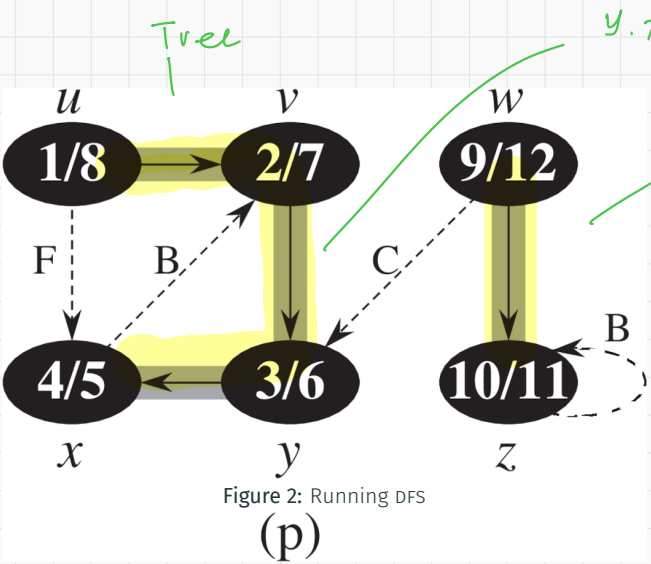


Figure 2: Running DFS

Pseudocode

```
1: DFS(G)
2: for each vertex u ∈ G.V do
3:     u.color ← WHITE   ◁ Not discovered.
4:     u.π ← NIL         ◁ It's parent in DFS forest
5: time ← 0
6: for each vertex u ∈ G.V do   ◁ Repeat DFS until all vertices are black.
7:     if u.color == WHITE then
8:         DFS-VISIT(G, u)   ◁ Do a DFS from u
```

🍰 The ⏱ complexity?

Pseudocode

```
1: DFS(G)
2: for each vertex u ∈ G.V do
3:     u.color ← WHITE
4:     u.π ← NIL
5: time ← 0
6: for each vertex u ∈ G.V do
7:     if u.color == WHITE then
8:         DFS-VISIT(G, u)
```

🍰 The ⏱ complexity?

```
1: DFS-VISIT(G, u)
2: time ← time + 1          ◁ The clock.
3: u.d ← time               ◁ The discover time
4: u.color ← GRAY           ▷ u has been discovered.
5: for each v ∈ G.Adj[u] do
6:     if v.color == WHITE then   ← Found a white node.
7:         v.π ← u                ← We discover v from u.
8:         DFS-VISIT(G, v)
9: u.color ← BLACK          ↖ u has been processed.
10: time ← time + 1
11: u.f ← time              ↖ Finishing time u.
```

🤔 The Parenthesis Structure

🍰 Do you see a pattern?

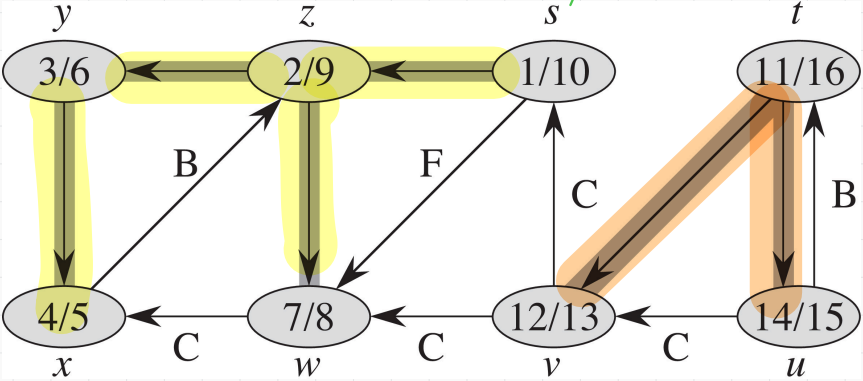


Figure 3: The result of running DFS

🍰 Do you see a pattern?

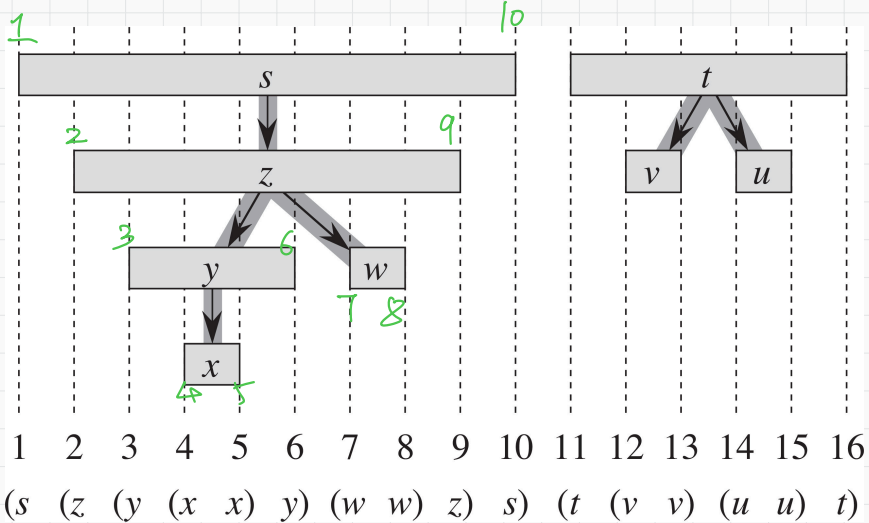


Figure 3: The intervals of discovery and finishing times

Predecessor Subgraph

For a graph $G = (V, E)$, we define the predecessor subgraph of a DFS as $G_\pi = (V_\pi, E_\pi)$, where

$$V_\pi = V$$

and

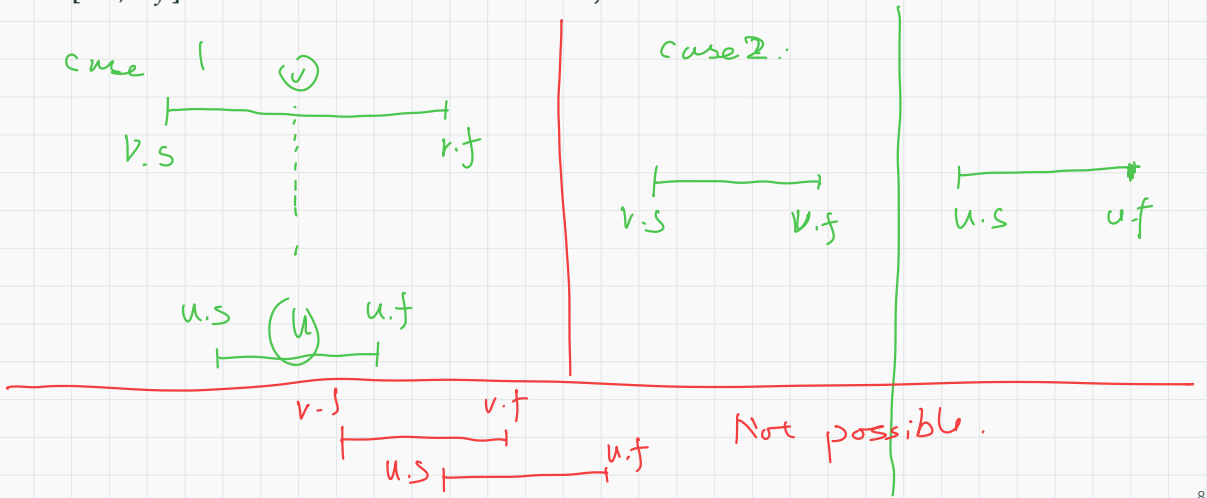
$$E_\pi = \{(v.\pi, v) : v \in V_\pi, v.\pi \neq \text{NIL}\}.$$

G_π for DFS forms a depth-first forest comprising depth-first trees.

💡 We get a forest in DFS instead of a tree in BFS.

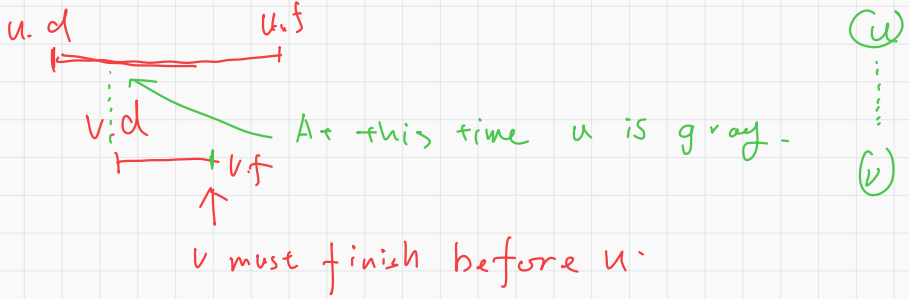
Theorem 20.7

In a DFS, for vertices u and v, exactly one holds. If u is a descendant of v, then $[u.d, u.f]$ lies within $[v.d, v.f]$. If v is a descendant of u, then $[v.d, v.f]$ lies within $[u.d, u.f]$. Otherwise the intervals are disjoint and neither is a descendant.



Theorem 20.7

Proof. Assume $u.d < v.d$. If $v.d < u.f$, then v is discovered while u is gray, so v is a descendant of u, and v finishes before u; hence $[v.d, v.f] \subseteq [u.d, u.f]$. If $u.f < v.d$, then u finishes before v is discovered, so the intervals are disjoint and neither is a descendant of the other. The case $v.d < u.d$ is symmetric. □



Theorem 20.7

Proof. Assume $u.d < v.d$. If $v.d < u.f$, then v is discovered while u is gray, so v is a descendant of u, and v finishes before u; hence $[v.d, v.f] \subseteq [u.d, u.f]$. If $u.f < v.d$, then u finishes before v is discovered, so the intervals are disjoint and neither is a descendant of the other. The case $v.d < u.d$ is symmetric. □

Corollary 20.8

Vertex v is a *proper* descendant of vertex u in the depth-first forest if and only if $u.d < v.d < v.f < u.f$.

$u \neq v$.

💡 In other words, by looking at the intervals, we will be able to reconstruct the predecessor graph.

Theorem 20.9 — White Path Theorem

In a depth-first forest, vertex v is a descendant of vertex u if and only if at the time $u.d$, there is a path from u to v consisting entirely of white vertices.

x, y, v will be descendants of u.

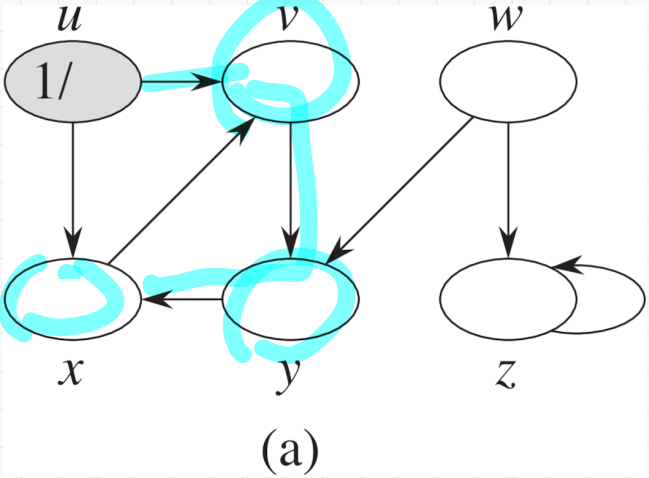


Figure 4: Which vertices will be descendants of s?

Theorem 20.9 — White Path Theorem

In a depth-first forest, vertex v is a descendant of vertex u if and only if at the time $u.d$, there is a path from u to v consisting entirely of white vertices.

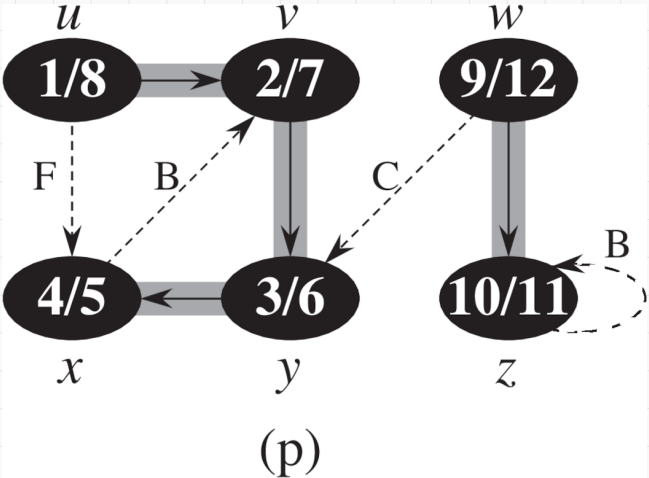


Figure 4: Which vertices will be descendants of s?

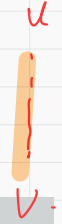
Theorem 20.9 — White Path Theorem

In a depth-first forest, vertex v is a descendant of vertex u if and only if at the time $u.d$, there is a path from u to v consisting entirely of white vertices.

Proof Outline of $\Rightarrow$ — Suppose v is a descendant of u in the depth-first forest. If $v = u$, the path from u to v is just u, which is white at time $u.d$.

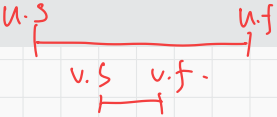
Otherwise, v is a proper descendant, so by Corollary 20.8 we have $u.d < v.d < v.f < u.f$, and v is white at time $u.d$.

Every vertex on the unique simple path from u to v in the depth-first forest is discovered after u, so all are white at time $u.d$.



Corollary 20.8

Vertex v is a *proper* descendant of vertex u in the depth-first forest for if and only if $u.d < v.d < v.f < u.f$.

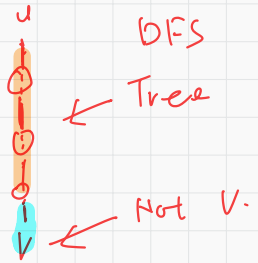


Theorem 20.9 — White Path Theorem

In a depth-first forest, vertex v is a descendant of vertex u if and only if at the time $u.d$, there is a path from u to v consisting entirely of white vertices.

Proof of $\Leftarrow$ — Suppose there exists a path of white vertices from u to v at time $u.d$, but v is not a descendant of u in the depth-first tree.

We can assume every vertex on this path, except v, becomes a descendant of u.



If not, pick one closer to u.

Theorem 20.9 — White Path Theorem

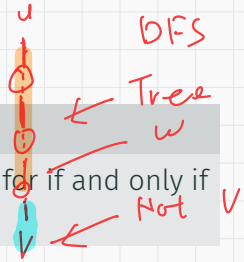
In a depth-first forest, vertex v is a descendant of vertex u if and only if at the time $u.d$, there is a path from u to v consisting entirely of white vertices.

Proof of $\Leftarrow$ — Suppose there exists a path of white vertices from u to v at time $u.d$, but v is not a descendant of u in the depth-first tree.

We can assume every vertex on this path, except v, becomes a descendant of u.

Let w be the closest vertex to v along the path.

Since w is a descendant of u, $w.f < u.f$ by Corollary 20.8.



Corollary 20.8

Vertex v is a *proper* descendant of vertex u in the depth-first forest for if and only if $u.d < v.d < v.f < u.f$.

Theorem 20.9 — White Path Theorem

In a depth-first forest, vertex v is a descendant of vertex u if and only if at the time $u.d$, there is a path from u to v consisting entirely of white vertices.

Proof of $\Leftarrow$ — Suppose there exists a path of white vertices from u to v at time $u.d$, but v is not a descendant of u in the depth-first tree.

We can assume every vertex on this path, except v, becomes a descendant of u.

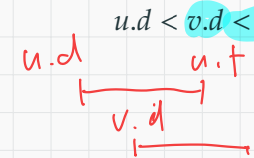
Let w be the closest vertex to v along the path.

Since w is a descendant of u, $w.f < u.f$ by Corollary 20.8.

As v is discovered after u and before w finishes, we have

$$u.d < v.d < w.f < u.f$$

Impossible



$\Rightarrow$ $u.d < v.d < v.f < u.f$.

$\rightarrow$ v is descedant of u.

Theorem 20.9 — White Path Theorem

In a depth-first forest, vertex v is a descendant of vertex u if and only if at the time $u.d$, there is a path from u to v consisting entirely of white vertices.

Proof of $\Leftarrow$ — Suppose there exists a path of white vertices from u to v at time $u.d$, but v is not a descendant of u in the depth-first tree.

We can assume every vertex on this path, except v, becomes a descendant of u.

Let w be the closest vertex to v along the path.

Since w is a descendant of u, $w.f < u.f$ by Corollary 20.8.

As v is discovered after u and before w finishes, we have

$$u.d < v.d < w.f < u.f$$

By Theorem 20.7, this implies that

$$u.d < v.d < v.f < u.f$$

implying v is indeed a descendant of u, a contradiction.

Four Types of Edges

We can categorize edges encountered in a DFS into four distinct types:

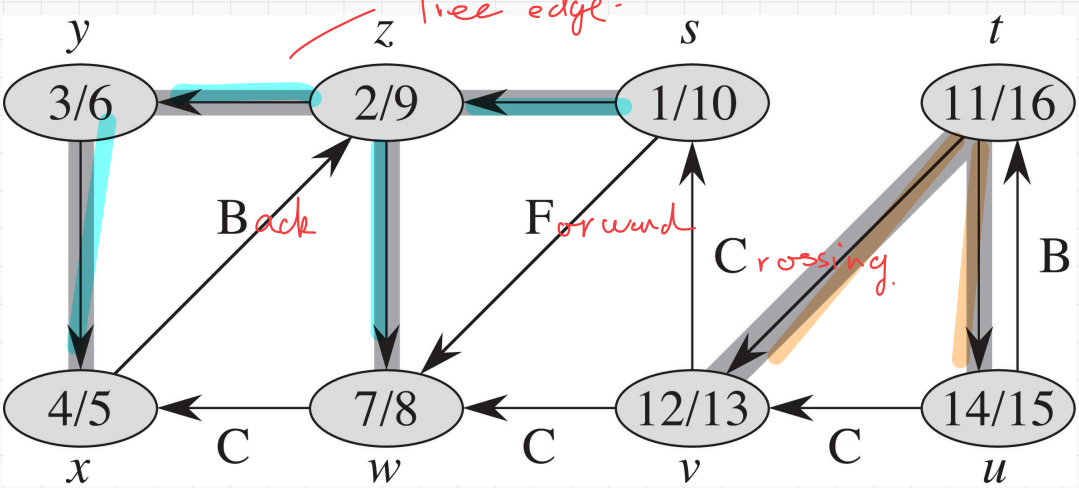


Figure 4: The four types of edges — tree, back, forward, and cross

Distinguish Types of Edges

When we first explore an edge (a, b), the color of vertex b tells us something about the edge —

- white indicates a tree edge,

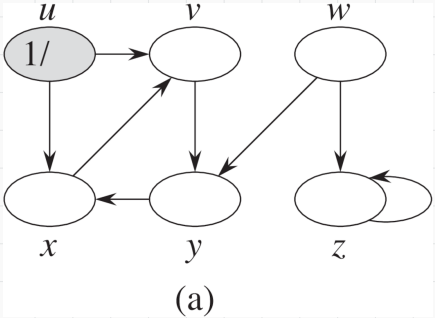


Figure 5: A tree edge

Distinguish Types of Edges

When we first explore an edge (a, b), the color of vertex b tells us something about the edge —

- white indicates a tree edge,
- gray indicates a back edge, and

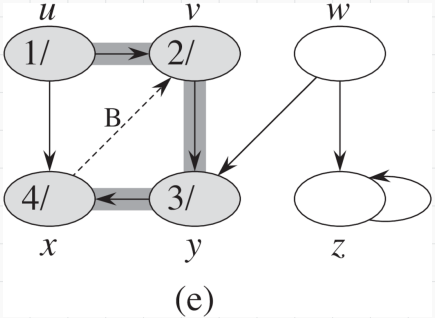


Figure 5: A back edge

Distinguish Types of Edges

When we first explore an edge (a, b), the color of vertex b tells us something about the edge —

- _____ indicates a tree edge,
- _____ indicates a back edge, and
- black indicates a forward edge.

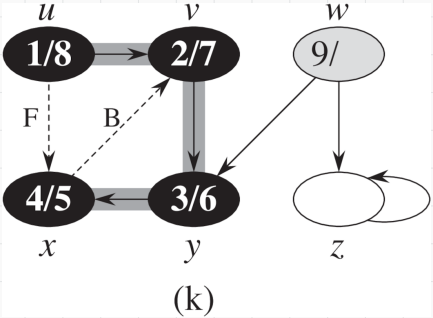


Figure 5: A forward edge

Distinguish Types of Edges

When we first explore an edge (a, b), the color of vertex b tells us something about the edge —

- _____ indicates a tree edge,
- _____ indicates a back edge, and
- black indicates a forward or cross edge.

By intervals.

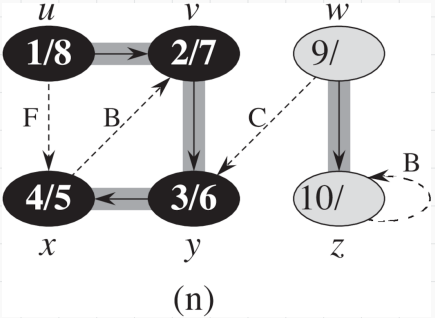


Figure 5: A cross edge

Theorem 20.10

In a DFS search of an undirected graph G, every edge is either a tree edge or a back edge.

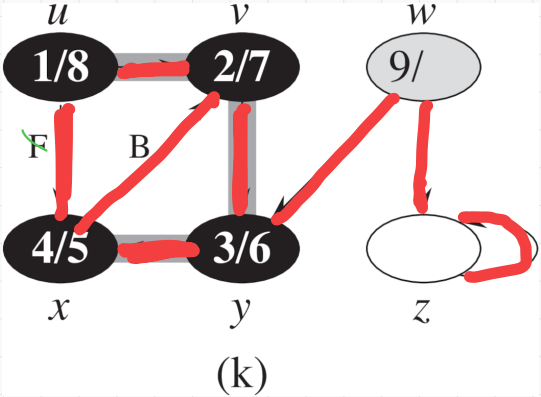


Figure 6: Not happening in an undirected graph

Theorem 20.10

In a DFS search of an undirected graph G, every edge is either a tree edge or a back edge.

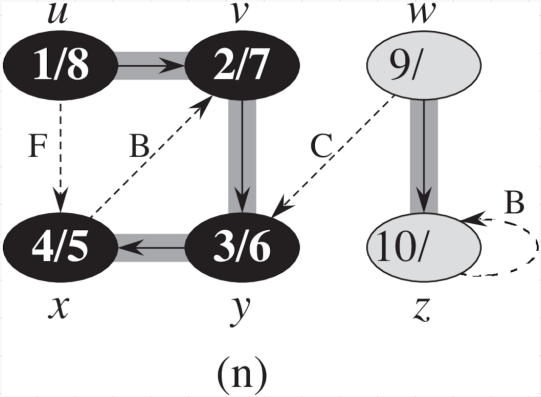
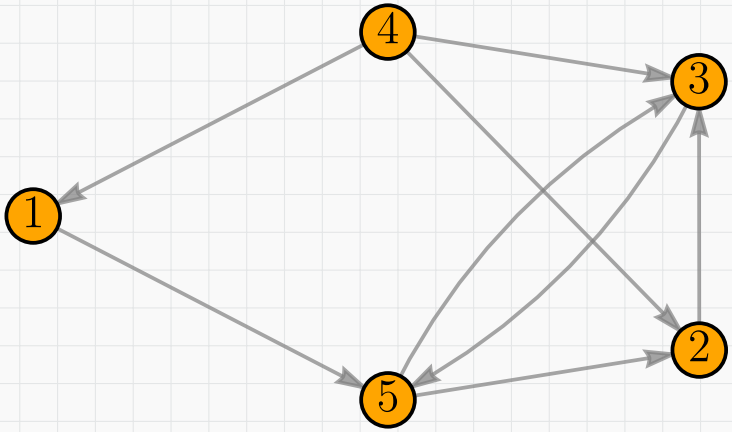


Figure 6: Not happening in an undirected graph

If we start with a DFS search on the following graph, starting from vertex 1, what is the finishing time of vertex 4?



ITA 20.4 Topological Sort

Sequencing Tasks — The Right Order Matters

Imagine a 🐰 that wakes up late for class.

The 🐰 must put on its clothes, but some items must be worn before others.

For instance, 🧦 should be on before 👞.

What sequence allows the 🐰 to dress in a correct order?

Figure 7: A bunny getting ready in haste

A Directed Acyclic Graph (DAG) is a digraph without cycles.

No cycle.

It can be used to represent the dependencies between tasks, as the dependencies of a task should not be circular.

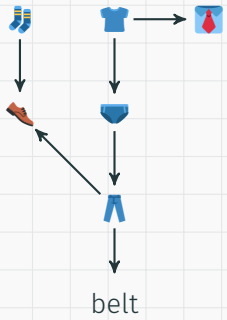


Figure 8: 📌 Example of a DAG

Topological Sorting via Depth-First Search

Topological sorting arranges a directed graph's vertices linearly, ensuring that for every directed edge $u \rightarrow v$, vertex u precedes v.

This ordering facilitates determining a sequence to execute tasks without violating dependencies.

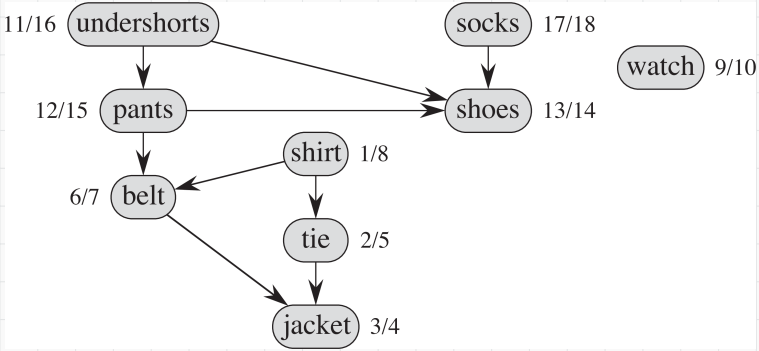


Figure 9: Illustration of Topological Sorting

Topological Sorting via Depth-First Search

Implementing this involves performing a DFS and positioning the vertices in descending order of their finish times.

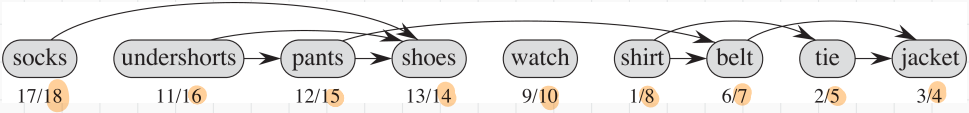


Figure 9: Illustration of Topological Sorting

Lemma 20.11

A directed graph G is acyclic if and only if a depth-first search of G yields no back edges.

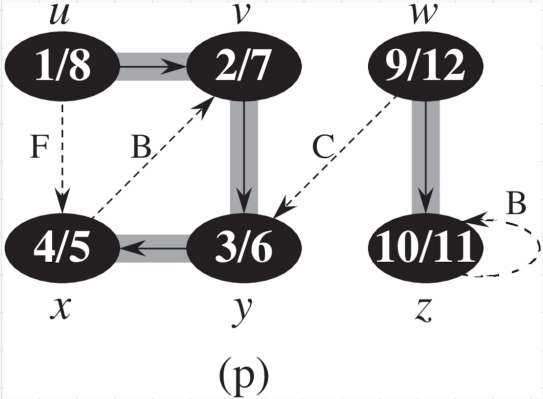


Figure 10: How to make this digraph a DAG

The Correctness of the Algorithm

Lemma 20.11

A directed graph G is acyclic if and only if a depth-first search of G yields no back edges.

Theorem 20.12

The following Topological-Sort produces a topological sort of G if it is a DAG.

Topological-Sort$((G))$
DFS(G) to compute finishing times $v.f$ for each vertex v
as each vertex is finished, insert it onto the front of a linked list
return the linked list of vertices

🤪 Proof not required.

ITA 20.5 Strongly Connected Components

Strongly Connected Components

An Strongly Connected Component (scc) is a maximal subgraph of a directed graph G in which every vertex is reachable from every other vertex.

Many graph algorithms first decompose a graph into sccs.

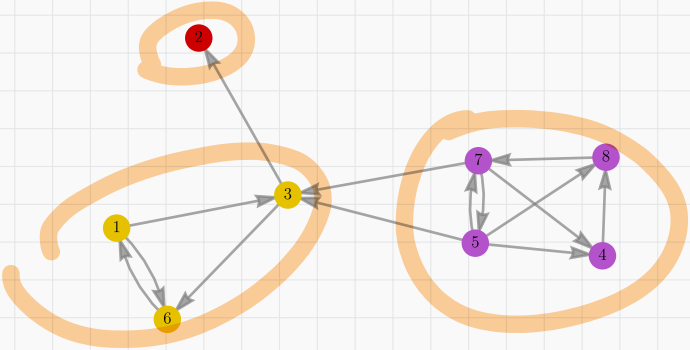


Figure 10: 📌 Example of sccs

The Transpose of a Directed Graph

Given a directed graph G, the transpose of G is the directed graph G^T in which every edge is reversed.

🍰 Why do G and G^T have the same sccs?

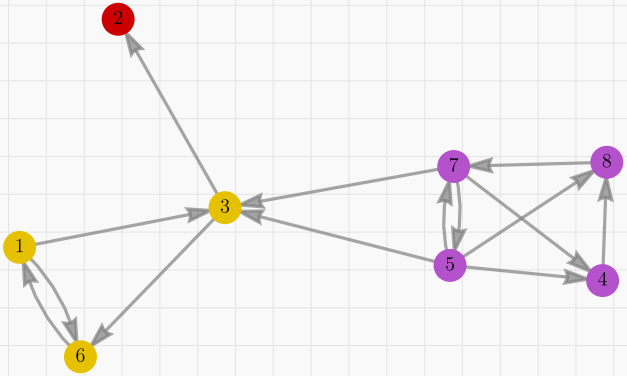


Figure 11: G

The Transpose of a Directed Graph

Given a directed graph G, the transpose of G is the directed graph G^T in which every edge is reversed.

🍰 Why do G and G^T have the same sccs?

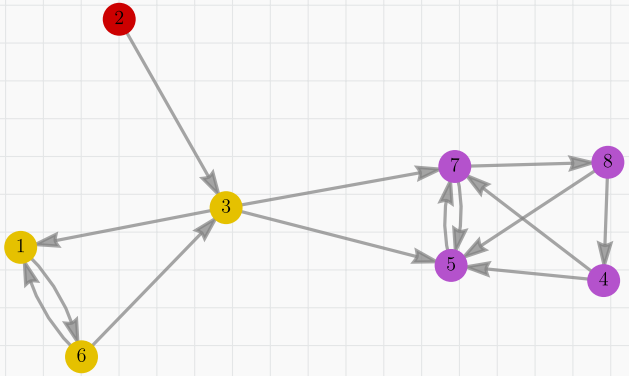
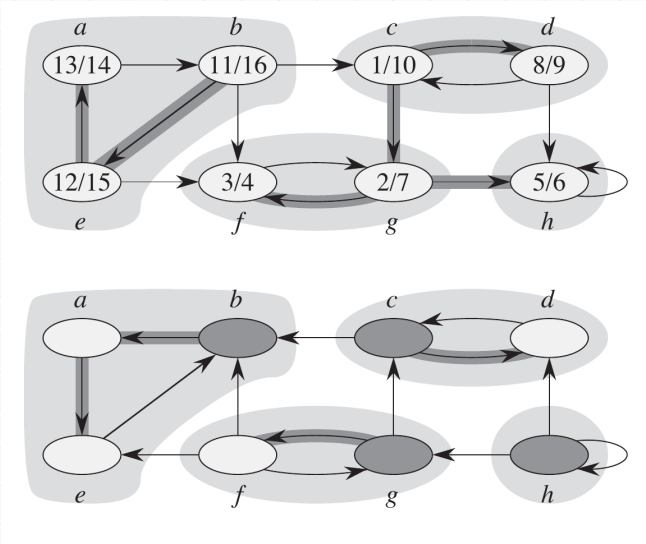


Figure 11: G^T

Example of Finding scc

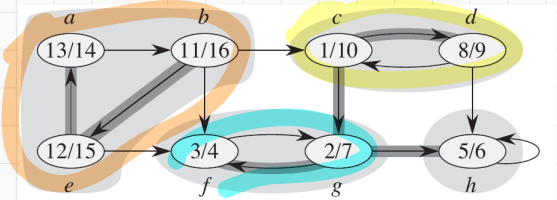


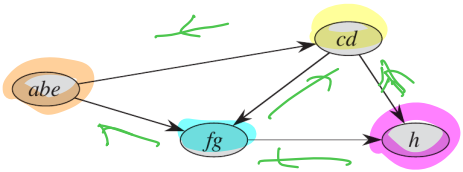
Idea of the Algorithm

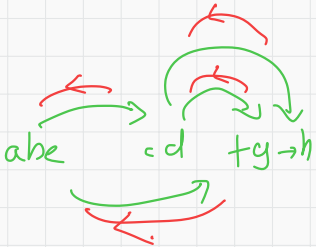
If we contract each SCC into a single vertex, then the result is a DAG, which we call the component graph of G and denote it by G^{scc}.

Why is G^{scc} a DAG?

Directed Acyclic Graph.

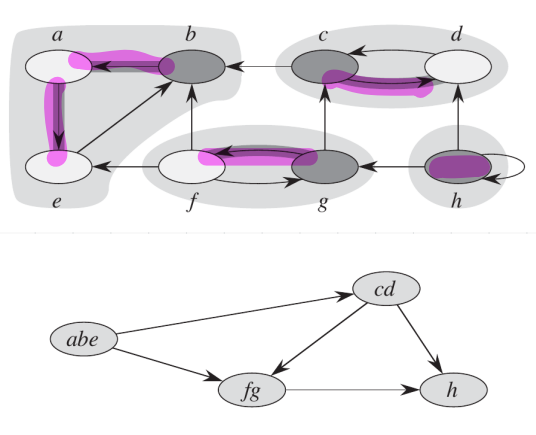






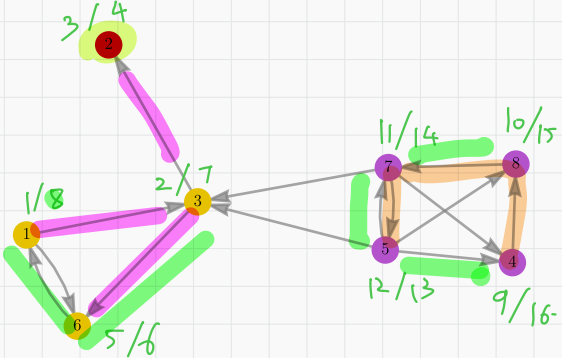
Idea of the Algorithm

😲 The second DFS essentially visits the SCCs in topologically sorted order of G^{scc}.



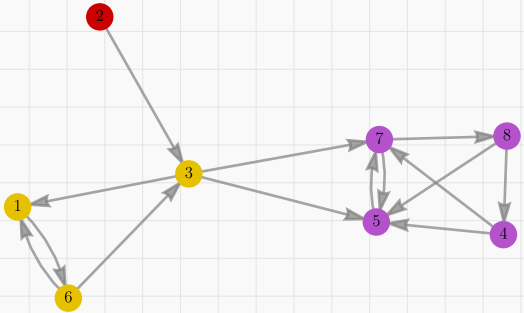
Using dfs to Find scc

1: **Strongly-Connected-Components**(G)
2: DFS(G) to compute finishing times $u.f$ for each vertex u
3: Compute G^T
4: DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 2)
5: Output the vertices of each tree in the depth-first forest formed in line 4 as a separate strongly connected component



Using dfs to Find scc

1: **Strongly-Connected-Components**(G)
2: DFS(G) to compute finishing times $u.f$ for each vertex u
3: Compute G^T
4: DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 2)
5: Output the vertices of each tree in the depth-first forest formed in line 4 as a separate strongly connected component



Lemma 20.13

Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$, let $u, v \in C$, let $u', v' \in C'$, and suppose that G contains a path $u \to u'$. Then G cannot also contain a path $v' \to v$.

We define finishing time for an scc C

$$f(C) = \max_{u \in C} u.f$$

Lemma 20.14

Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E$, where $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.

We define finishing time for an scc C

$$f(C) = \max_{u \in C} u.f$$

Lemma 20.14

Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E$, where $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.

Corollary 20.15

Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E^T$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.

Theorem 20.16

The procedure Strongly-Connected-Components correctly computes the strongly connected components of the directed graph G provided as its input.

Theorem 20.16

The procedure Strongly-Connected-Components correctly computes the strongly connected components of the directed graph G provided as its input.

The proof uses induction on the number of depth-first trees formed during the DFS of G^T.

The base case, when there are no trees, is trivial.

The inductive hypothesis assumes the first k trees each correspond to a scc.

Theorem 20.16

The procedure Strongly-Connected-Components correctly computes the strongly connected components of the directed graph G provided as its input.

In the inductive step, we consider the $(k+1)$-th tree, rooted at u in the component C.

By construction, u's finish time ensures that any other component C' processed later has vertices marked white when DFS reaches u.

Thus, all vertices in C are reachable from u in G^T and they have not been visited yet.

Theorem 20.16

The procedure STRONGLY-CONNECTED-COMPONENTS correctly computes the strongly connected components of the directed graph G provided as its input.

Corollary 20.15 guarantees that edges in G^T leaving from C lead only to completed SCC.

Therefore, no vertex outside C can become a descendant of u in the second run of DFS.

What are your main takeaways today? Any questions?