Algorithms and Datastructures

Depth-First Search
September 19, 2023

iCIS | Software Science
Radboud University

Outline

Recap

Depth-First Search

Topological Sorts

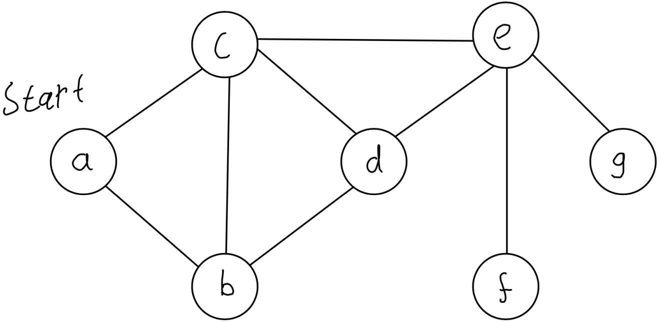
Checking for cycles

Strongly Connected Components

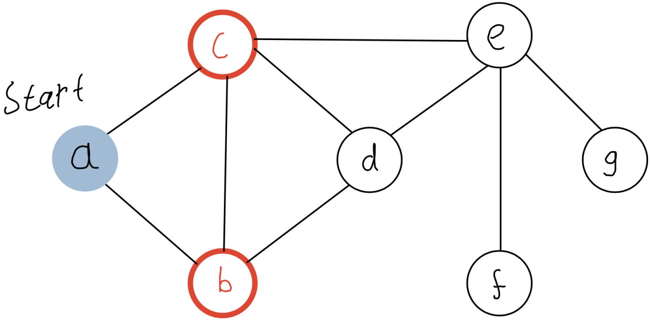
What did we do last week?

- We discussed graphs and applications
- We discussed the basic terminology: vertex, edge, adjacent, source, target, (un)directed, weighted
- We discussed different ways of representing graphs: adjacency lists, adjacency matrix
- We discussed breadth-first search: functional correctness and complexity

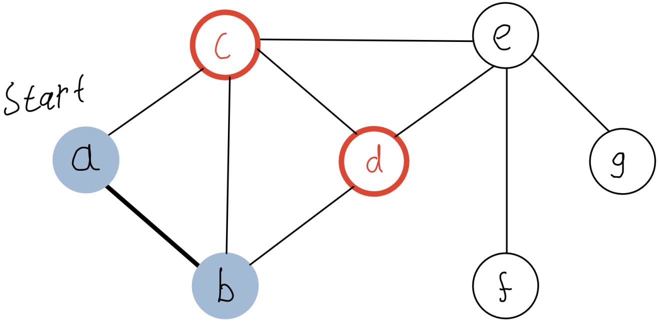
Recap: breadth-first search



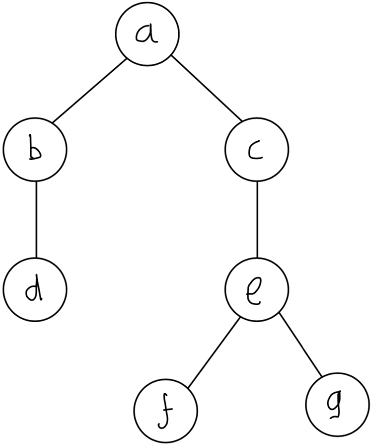
Recap: breadth-first search



Recap: breadth-first search



Search order



The algorithm of today: depth-first search

- Today, we look at **depth-first search**
- Again a search algorithm, quite similar to breadth first search
- We also look at three applications.

Outline

Recap

Depth-First Search

Topological Sorts

Checking for cycles

Strongly Connected Components

Stacks

For stacks, we have the following operations:

- Return the empty stack
- Determine whether the stack is empty
- Push: add an element to the front of the stack
- Pop: return and remove the front element from the stack

Stacks

For stacks, we have the following operations:

- Return the empty stack
- Determine whether the stack is empty
- Push: add an element to the front of the stack
- Pop: return and remove the front element from the stack

For example:

Pushing 1 to $[2, 3]$ gives $[1, 2, 3]$

If we pop $[1, 2, 3]$, we get 1 and the stacks becomes $[2, 3]$

Stacks

For stacks, we have the following operations:

- Return the empty stack
- Determine whether the stack is empty
- Push: add an element to the front of the stack
- Pop: return and remove the front element from the stack

For example:

Pushing 1 to $[2, 3]$ gives $[1, 2, 3]$

If we pop $[1, 2, 3]$, we get 1 and the stacks becomes $[2, 3]$

Basically a **last in-first out queue**.
We can implement them via linked lists.

First Implementation: Iterative

```
enum State := { UNDISCOVERED, DISCOVERED }

void dfs(G, v)
    // initialize
    for each u in vertex(G) unequal v
        explored[u] := UNDISCOVERED
        predecessor[u] := null
    explored[v] := DISCOVERED
    predecessor[v] := null
    S := emptyStack
    push(S, v)
    // main loop
    while (!isEmpty(S))
        u := pop(S)
        for each w in adjacent(u)
            if (explored[w] == UNDISCOVERED)
                explored[w] := DISCOVERED
                predecessor[w] := u
                push(S, w)
```

Second Implementation: Recursive

```
enum State := { UNDISCOVERED, DISCOVERED }

void dfs-init(G, v)
    for each u in vertex(G) unequal v
        explored[u] := UNDISCOVERED
        predecessor[u] := null
    explored[v] := DISCOVERED
    predecessor[v] := null
    dfs-visit(G, v)

void dfs-visit(G, v)
    for each u in adjacent(v)
        if (explored[u] == UNDISCOVERED)
            explored[u] := DISCOVERED
            predecessor[u] := v
            dfs-visit(G, u)
```

Remark

- The two versions of depth-first search presented here, do **not** consider the vertices in the same order.

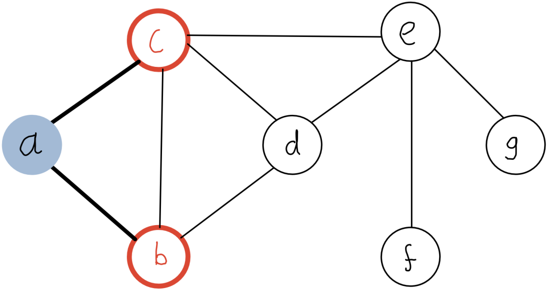
Remark

- The two versions of depth-first search presented here, do **not** consider the vertices in the same order.
- **Iterative version**: add all neighbors to the stack, then continue searching
- So: you continue searching from the **last** neighbor

Remark

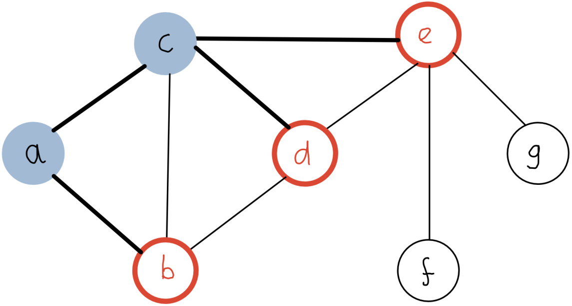
- The two versions of depth-first search presented here, do **not** consider the vertices in the same order.
- **Iterative version**: add all neighbors to the stack, then continue searching
- So: you continue searching from the **last** neighbor
- **Recursive version**: continue searching from the first neighbor

Example: depth-first search



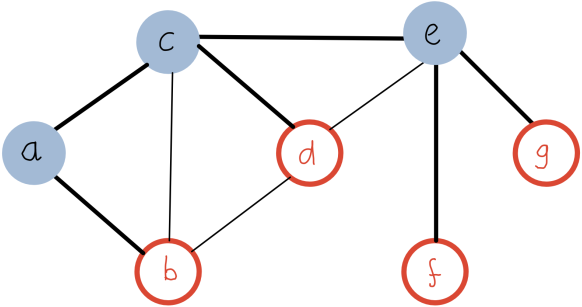
This is the iterative version

Example: depth-first search



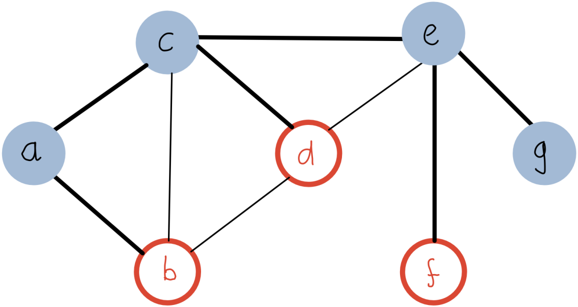
This is the iterative version

Example: depth-first search



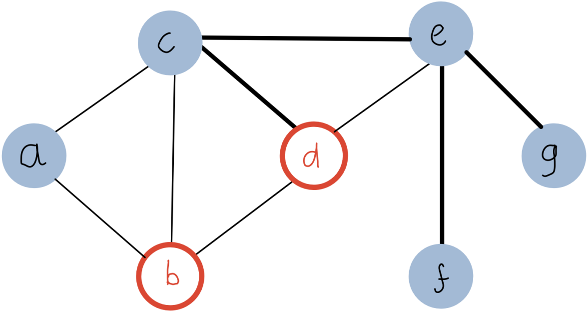
This is the iterative version

Example: depth-first search



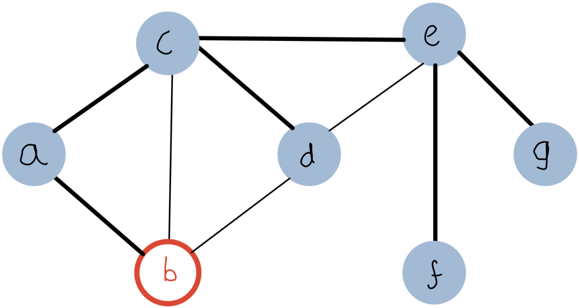
This is the iterative version

Example: depth-first search



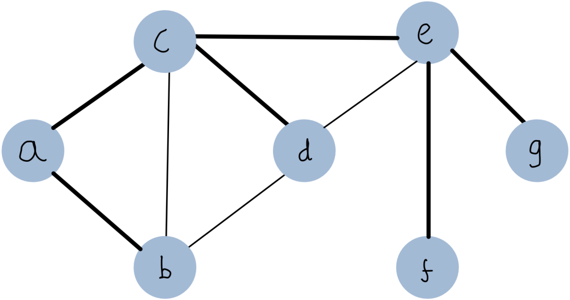
This is the iterative version

Example: depth-first search



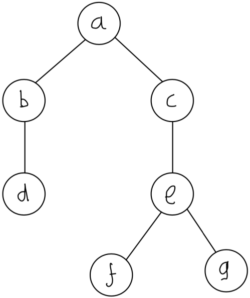
This is the iterative version

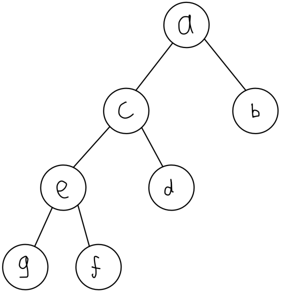
Example: depth-first search



This is the iterative version

The search order





Left: breadth-first search

Right: depth-first search

Complexity of depth-first search: initialization

```
enum State := { UNDISCOVERED, DISCOVERED }

void dfs(G, v)
    // initialize
    for each u in vertex(G) unequal v
        explored[u] := UNDISCOVERED
        predecessor[u] := null
    explored[v] := DISCOVERED
    predecessor[v] := null
    S := emptyStack
    push(S, v)
```

Done in $\mathcal{O}(|V|)$

Complexity of depth-first search: main loop

```
// main loop
while (!isEmpty(S))
   u := pop(S)
   for each w in adjacent(u)
      if (explored[w] == UNDISCOVERED)
         explored[w] := DISCOVERED
         predecessor[w] := u
         push(S, w)
```

Each individual line happens in $\mathcal{O}(1)$
So: we need to count the number of repetitions per line

Complexity of depth-first search: main loop

```
// main loop
while (!isEmpty(S))                                  // ? repetitions
   u := pop(S)                                       // ? repetitions
   for each w in adjacent(u)
      if (explored[w] == UNDISCOVERED)               // ? repetitions
         explored[w] := DISCOVERED                   // ? repetitions
         predecessor[w] := u                         // ? repetitions
         push(S, w)                                  // ? repetitions
```

Complexity of depth-first search: main loop

```
// main loop
while (!isEmpty(S))                              // ? repetitions
   u := pop(S)                                   // ? repetitions
   for each w in adjacent(u)
      if (explored[w] == UNDISCOVERED)           // ? repetitions
         explored[w] := DISCOVERED               // ? repetitions
         predecessor[w] := u                     // ? repetitions
         push(S, w)                              // ? repetitions
```

Observation 1: a vertex can enter the stack **at most once**

Complexity of depth-first search: main loop

```
// main loop
while (!isEmpty(S))                                   // |V| repetitions
    u := pop(S)                                       // |V| repetitions
    for each w in adjacent(u)
        if (explored[w] == UNDISCOVERED)              // ? repetitions
            explored[w] := DISCOVERED                 // ? repetitions
            predecessor[w] := u                       // ? repetitions
            push(S, w)                                // ? repetitions
```

Complexity of depth-first search: main loop

```
// main loop
while (!isEmpty(S))                                    // |V| repetitions
   u := pop(S)                                         // |V| repetitions
   for each w in adjacent(u)
      if (explored[w] == UNDISCOVERED)                 // ? repetitions
         explored[w] := DISCOVERED                     // ? repetitions
         predecessor[w] := u                           // ? repetitions
         push(S, w)                                    // ? repetitions
```

Observation 2: an edge can be explored **at most once**

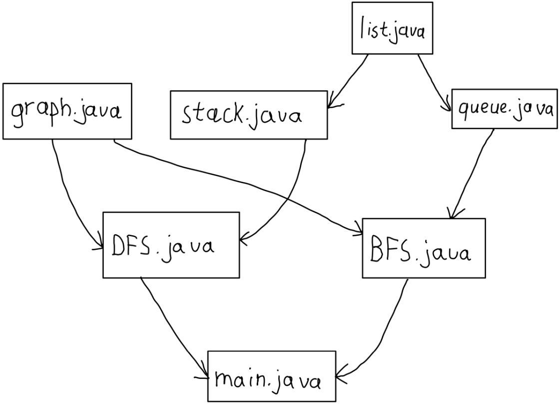
Complexity of depth-first search: main loop

```
    // main loop
    while (!isEmpty(S))                              // |V| repetitions
       u := pop(S)                                   // |V| repetitions
       for each w in adjacent(u)
          if (explored[w] == UNDISCOVERED)           // |E| repetitions
             explored[w] := DISCOVERED               // |E| repetitions
             predecessor[w] := u                     // |E| repetitions
             push(S, w)                              // |E| repetitions
```

Total: $\mathcal{O}(|V| + |E|)$

Outline

Tasks



Suppose, we have these files and dependencies. In which order should we link them?

Topological Sorts

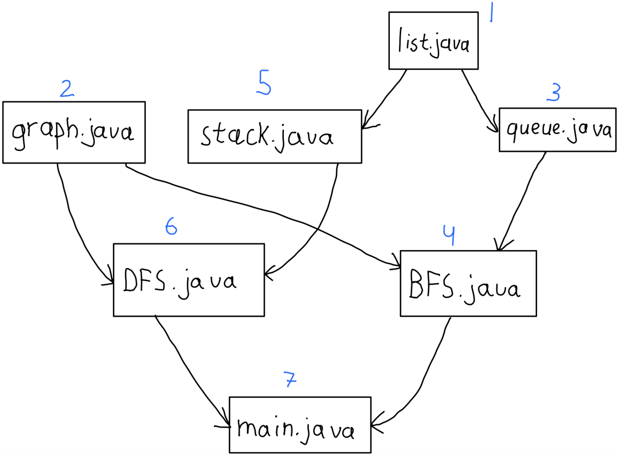
Suppose, we have a graph.
Goal: assign to every vertex v a number $f(v)$ such that if we have an edge from v_1 to v_2, then we have $f(v_1) < f(v_2)$.

Topological Sorts

Suppose, we have a graph.
Goal: assign to every vertex v a number $f(v)$ such that if we have an edge from v_1 to v_2, then we have $f(v_1) < f(v_2)$.
Applications of topological sorting:

- Compiling files
- Scheduling jobs

A topological sort



Cyclic Dependencies?

Let's say we have two jobs: l and o

- Job l needs to be finished before job o
- Job o needs to be finished before job l

We can't schedule this!

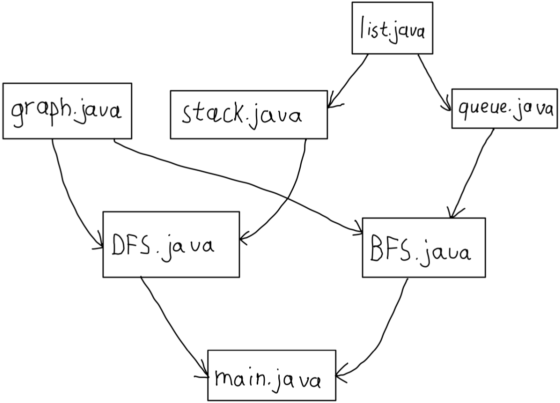
Directed Acyclic Graphs

A graph is called a **directed acyclic graph** (DAG) if

- it is directed
- it has no cycles

A **cycle** is a list of edges $v \to w_1 \to \ldots \to w_n \to v$.

Example of a DAG



Computing Topological Sorts: Idea

Observation:

- Let's say, we are running depth-first search on a DAG
- We are exploring some node v

Computing Topological Sorts: Idea

Observation:

- Let's say, we are running depth-first search on a DAG
- We are exploring some node v
- Before we finish exploring v, we first explore all neighbors of v
- So: the earlier we finish exploring a vertex, the *later* it should be in a topological order
- Main idea: keep track of the finishing time going from high to low

Computing Topological Sorts: Idea

Observation:

- Let's say, we are running depth-first search on a DAG
- We are exploring some node v
- Before we finish exploring v, we first explore all neighbors of v
- So: the earlier we finish exploring a vertex, the *later* it should be in a topological order
- Main idea: keep track of the finishing time going from high to low
- For this particular problem, we do not care about finding a path.
- We do not keep track of predecessors

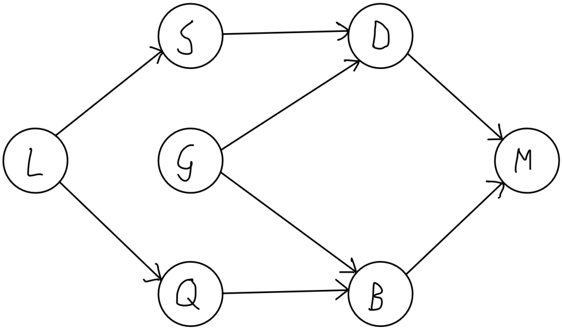
Finding Topological Sorts

```
enum State := { UNDISCOVERED, EXPLORED }

void top-init(G)
    for each u in vertex(G)
        explored[u] := UNDISCOVERED
    time := size(vertex(G))
    for each v in vertex(G)
        if (explored[v] == UNDISCOVERED)
            top-visit(G, v)
```

Finding Topological Sorts

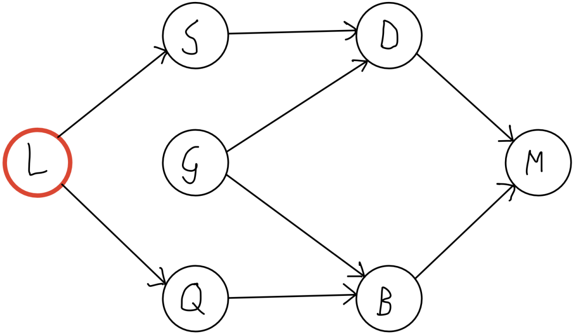
```
void top-visit(G, v)
    explored[v] := EXPLORED

    for each u in adjacent(v)
        if (explored[u] == UNDISCOVERED)
            top-visit(G, u)

    f[v] := time
    time := time - 1
```

Example of Topological sort



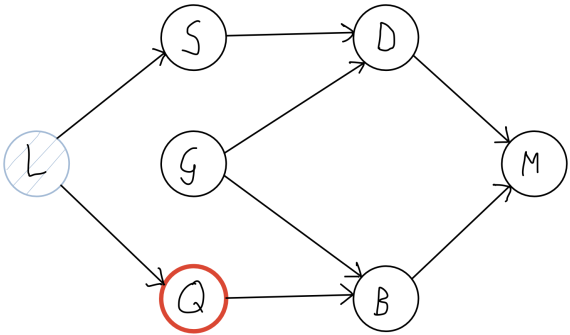
Blue, lines: discovered   Blue, filled: explored   Red: exploring.

Example of Topological sort



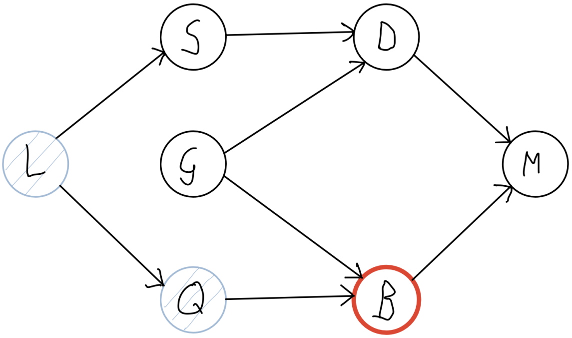
Blue, lines: discovered   Blue, filled: explored   Red: exploring.

Example of Topological sort



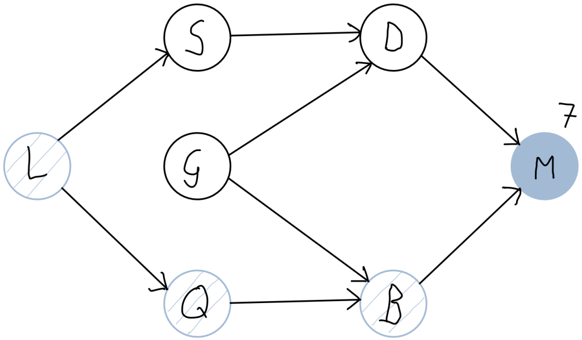
Blue, lines: discovered Blue, filled: explored Red: exploring.

Example of Topological sort



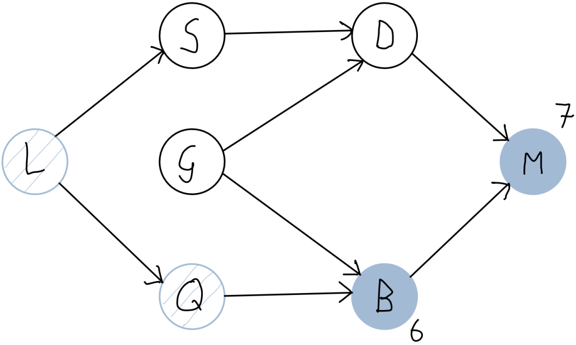
Blue, lines: discovered   Blue, filled: explored   Red: exploring.

Example of Topological sort



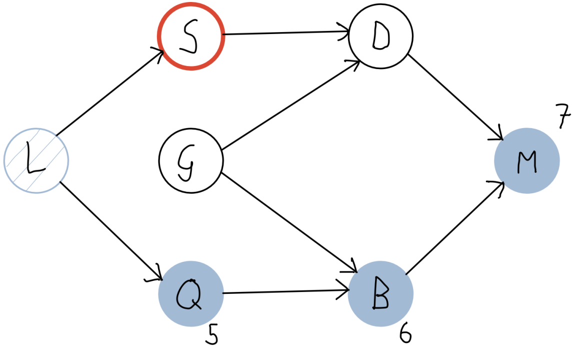
Blue, lines: discovered   Blue, filled: explored   Red: exploring.

Example of Topological sort



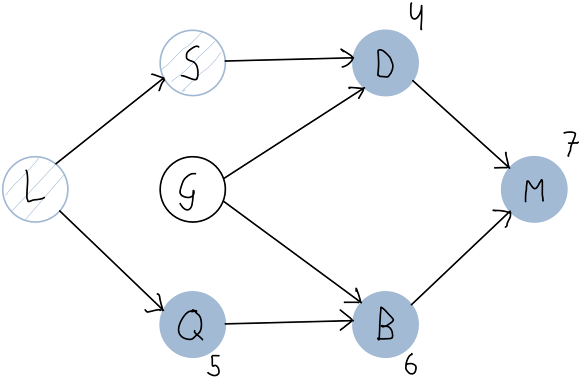
Blue, lines: discovered   Blue, filled: explored   Red: exploring.

Example of Topological sort



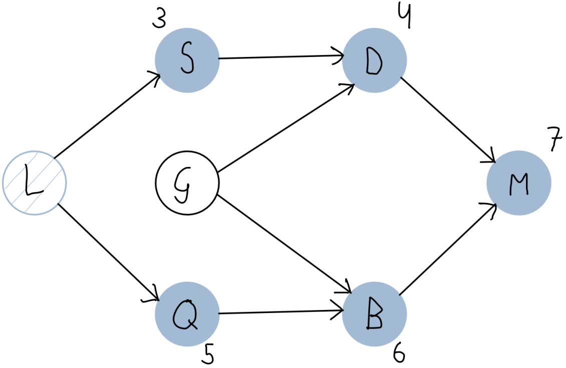
Blue, lines: discovered Blue, filled: explored Red: exploring.

Example of Topological sort



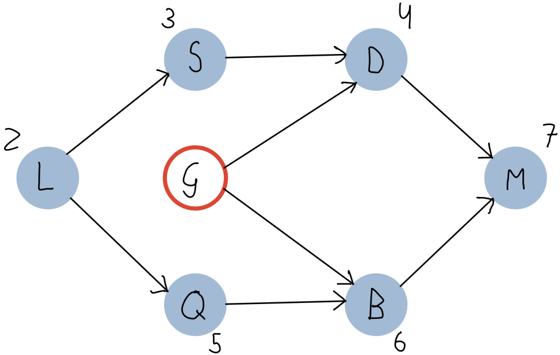
Blue, lines: discovered   Blue, filled: explored   Red: exploring.

Example of Topological sort



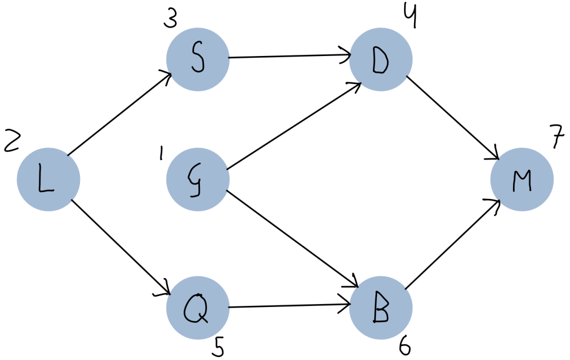
Blue, lines: discovered   Blue, filled: explored   Red: exploring.

Example of Topological sort



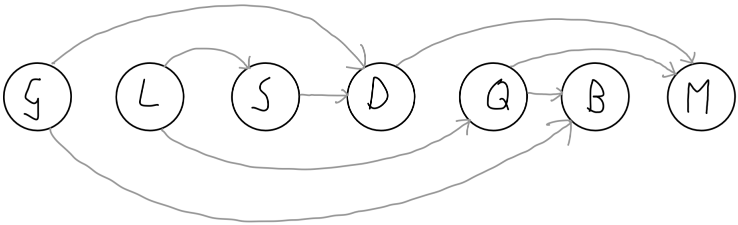
Blue, lines: discovered Blue, filled: explored Red: exploring.

Example of Topological sort



Blue, lines: discovered   Blue, filled: explored   Red: exploring.

The resulting topological sort



Correctness and Complexity

- We know that DFS runs in $\mathcal{O}(|V| + |E|)$, and the same can be said for topological sorts
- So, we only need to prove correctness

Correctness and Complexity

- We know that DFS runs in $\mathcal{O}(|V| + |E|)$, and the same can be said for topological sorts
- So, we only need to prove correctness
- We need to show: if G is a directed acyclic graph, then the algorithm computes a topological sort
- We assume that G has no cycles and we need to prove that whenever we have an edge from v to w, then $f(v) < f(w)$

Proof of Correctness

- Suppose, G does not have any cycles
- Suppose, we have vertices v, w and an edge from v to w
- To show: $f(v) < f(w)$

Proof of Correctness

- Suppose, G does not have any cycles
- Suppose, we have vertices v, w and an edge from v to w
- To show: $f(v) < f(w)$
- **Key observation**: there is no path from w to v, because this would create a cycle (*violates the assumption that G is a DAG*)
- **So**: if **top-visit**(G, w) gets executed, the algorithm will not visit v

Proof of Correctness

- Suppose, G does not have any cycles
- Suppose, we have vertices v, w and an edge from v to w
- To show: $f(v) < f(w)$
- **Key observation**: there is no path from w to v, because this would create a cycle (*violates the assumption that G is a DAG*)
- **So**: if **top-visit**(G, w) gets executed, the algorithm will not visit v
- There are two options: the algorithm either executes **top-visit**(G, w) before **top-visit**(G, v) or the other way around

Proof of Correctness

- Suppose, G does not have any cycles
- Suppose, we have vertices v, w and an edge from v to w
- To show: $f(v) < f(w)$
- **Key observation**: there is no path from w to v, because this would create a cycle (*violates the assumption that G is a DAG*)
- **So**: if **top-visit**(G, w) gets executed, the algorithm will not visit v
- There are two options: the algorithm either executes **top-visit**(G, w) before **top-visit**(G, v) or the other way around
- First case: w will be finished before v (*no path from w to v*). So: $f(v) < f(w)$
- Second case: before **top-visit**(G, v) is finished, **top-visit**(G, w) must be finished.
- So: we have $f(v) < f(w)$ in both cases

Outline

Question

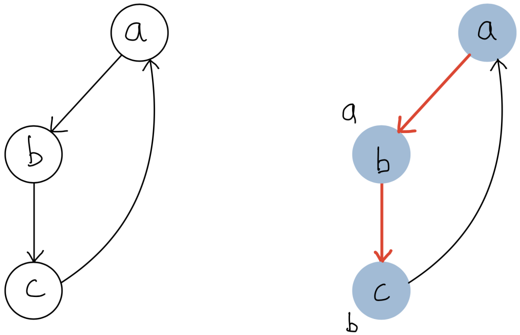
How to check whether a graph is a directed acyclic graph?

Question

How to check whether a graph is a directed acyclic graph?
Again we can use DFS!

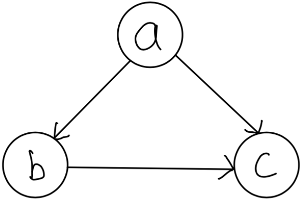
Cycles and Depth-First Search

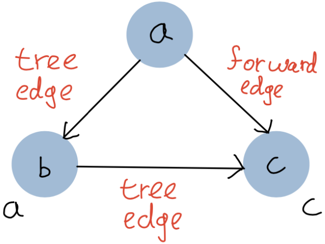
If the graph has a cycle, then the following happens during DFS



There is an edge from a descendant of some v to v that's not in the depth-first search tree.

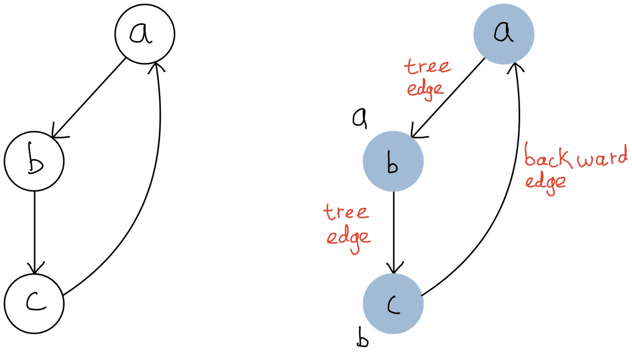
Identification of Edges: forward edge





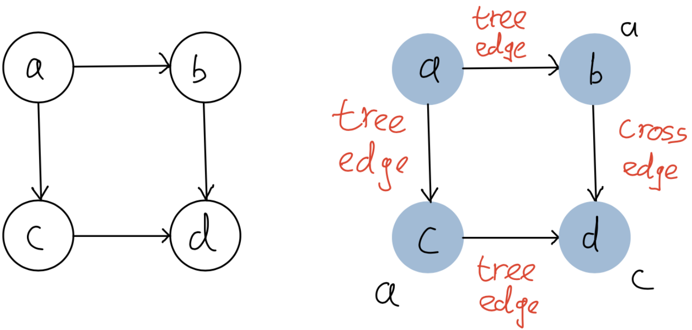
A **tree edge** is an edge that occurs in the depth first search tree.
A **forward edge** is an edge (u, v) that is not a tree edge and such that v is a descendant of u.

Identification of Edges: backward edge



A **backward edge** is an edge (u, v) that is not a tree edge and such that u is a descendant of v.

Identification of Edges: cross edge



A **cross edge** is any other edge.

Idea

We add the following.

- **Colors**: is a node undiscovered, discovered or explored?
- **Discovery time**: at which step was the node discovered?
- **Finishing time**: at which step are all neighbors explored?
- Perform it on all vertices

Note: for thsi particular application (finding cycles), we don't care about predecessors.

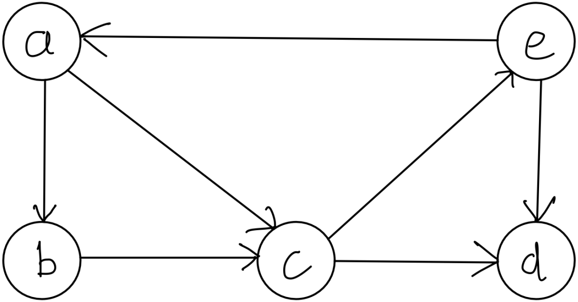
Implementation of Depth First Search

```
enum State := { UNDISCOVERED, DISCOVERED , EXPLORED }

void dfs-init(G)
    for each u in vertex(G)
        explored[u] := UNDISCOVERED
    time := 0
    for each v in vertex(G)
        if (explored[v] == UNDISCOVERED)
            dfs-visit(G, v)
```

Implementation of Depth First Search

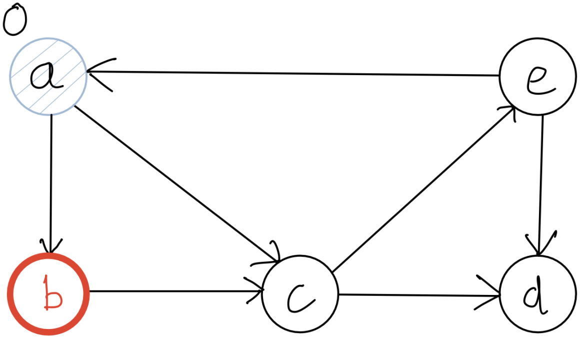
```
void dfs-visit(G, v)
    explored[v] := DISCOVERED
    d[v] := time
    time := time + 1

    for each u in adjacent(v)
        if (explored[u] == UNDISCOVERED)
            dfs-visit(G, u)

    explored[v] := EXPLORED
    f[v] := time
    time := time + 1
```

Example of Depth-First Search



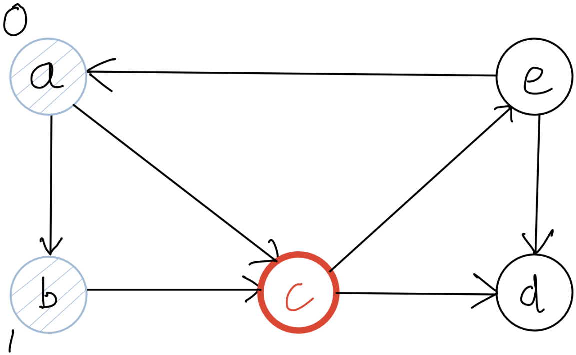
Blue, lines: discovered Blue, filled: explored Red: exploring.

Example of Depth-First Search



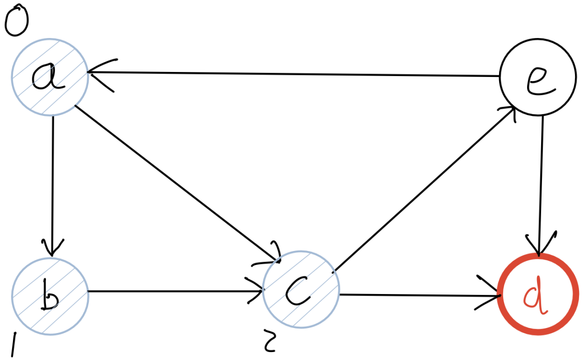
Blue, lines: discovered　　Blue, filled: explored　　Red: exploring.

Example of Depth-First Search



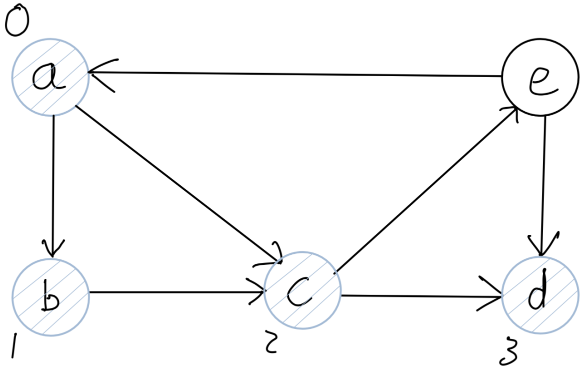
Blue, lines: discovered Blue, filled: explored Red: exploring.

Example of Depth-First Search



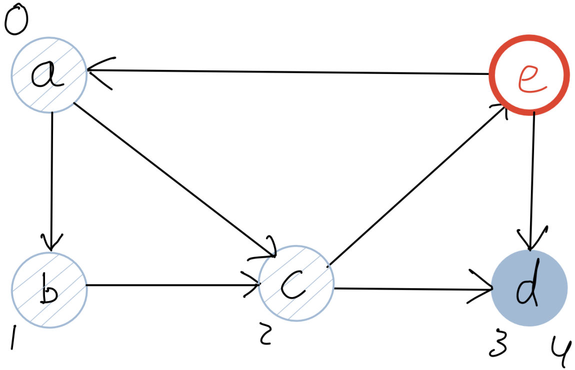
Blue, lines: discovered   Blue, filled: explored   Red: exploring.

Example of Depth-First Search



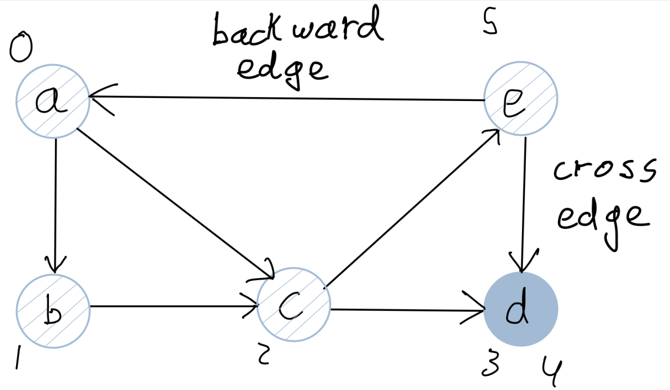
Blue, lines: discovered Blue, filled: explored Red: exploring.

Example of Depth-First Search



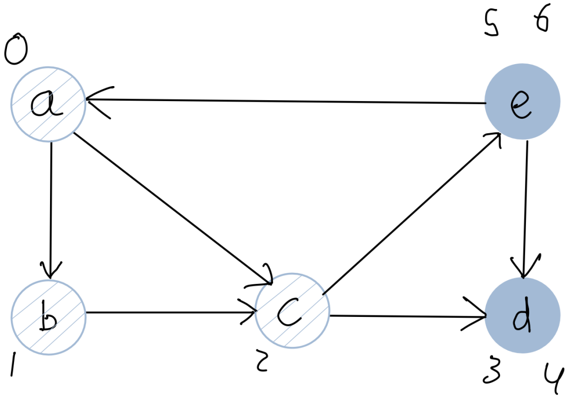
Blue, lines: discovered   Blue, filled: explored   Red: exploring.

Example of Depth-First Search



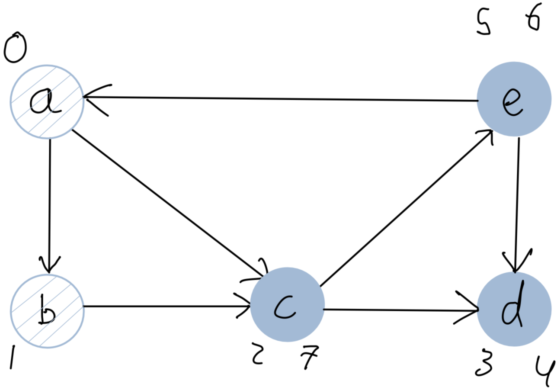
Blue, lines: discovered Blue, filled: explored Red: exploring.

Example of Depth-First Search



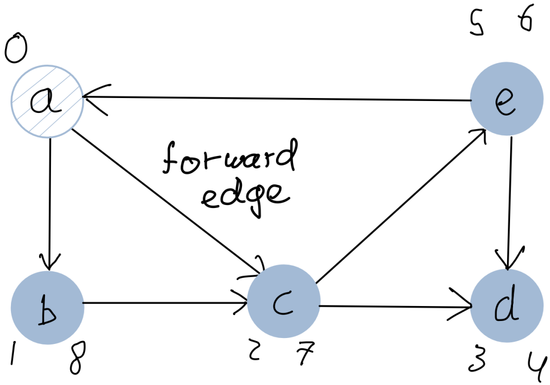
Blue, lines: discovered   Blue, filled: explored   Red: exploring.

Example of Depth-First Search



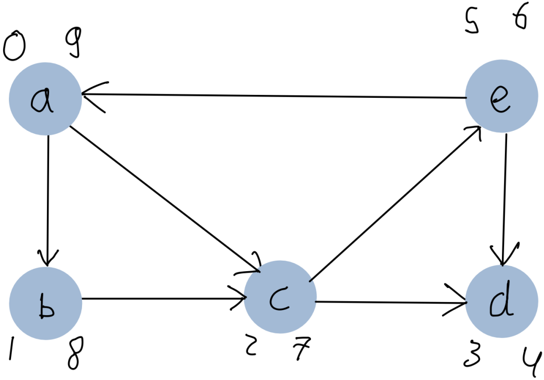
Blue, lines: discovered Blue, filled: explored Red: exploring.

Example of Depth-First Search



Blue, lines: discovered   Blue, filled: explored   Red: exploring.

Example of Depth-First Search



Blue, lines: discovered   Blue, filled: explored   Red: exploring.

Observations

By performing depth-first search, we can recognize the types of edges.
When we discover an edge (u, v), then it is a

- a tree edge if v is undiscovered
- a backward edge if v is discovered
- a forward edge if v is explored and $d[u] < d[v]$
- a cross edge if v is explored and $d[v] < d[u]$

In addition, we also note

- for all v, we have $\mathrm{d[v]} < \mathrm{f[v]}$
- if $v \neq w$, then we also have $\mathrm{d[v]} \neq \mathrm{d[w]}$ and $\mathrm{f[v]} \neq \mathrm{f[w]}$

Cycles via DFS

We can determine whether a graph has a cycle as follows

- Perform DFS
- If at some step a backward edge is detected, then there is a cycle
- If no backward edges are detected, then there is no cycle

This allows us to check whether a graph is a DAG.

Outline

Strongly Connected Components

- A graph is called **strongly connected** if for all v and w we have a path from v to w and from w to v.
- The **strongly connected component** of a vertex v consists of all the vertices w such that we have a path from v to w and a path from w to v.

Strongly Connected Components

- A graph is called **strongly connected** if for all v and w we have a path from v to w and from w to v.
- The **strongly connected component** of a vertex v consists of all the vertices w such that we have a path from v to w and a path from w to v.

Applications of strongly connected components

- Communities in social networks
- Mutual admiration societies in citation graphs

Strongly Connected Components

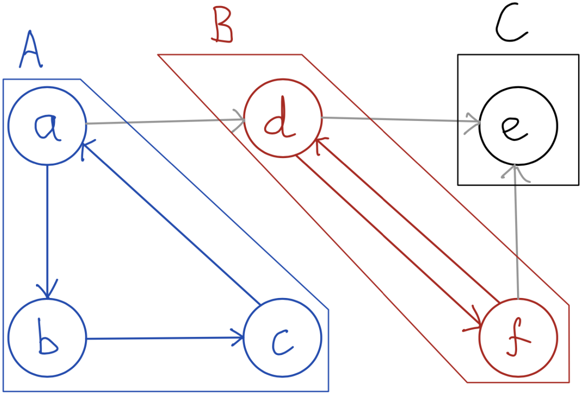
- A graph is called **strongly connected** if for all v and w we have a path from v to w and from w to v.
- The **strongly connected component** of a vertex v consists of all the vertices w such that we have a path from v to w and a path from w to v.

Applications of strongly connected components

- Communities in social networks
- Mutual admiration societies in citation graphs

Goal: given a graph, determine its strongly connected components

Example Strongly Connected Components



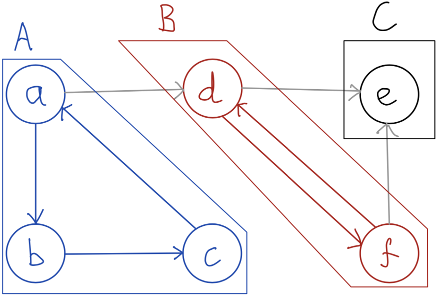
This graph has three strongly connected components.

Component Graph

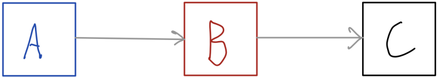
Given a graph G, define a graph G^{SCC} as follows:

- Vertices: strongly connected components C
- We have an edge from C_1 to C_2 if there are vertices $v \in C_1$ and $w \in C_2$ and an edge from v to w.

Example



Its component graph:



The transpose of directed graph

Let G be a graph. Define G^{T}

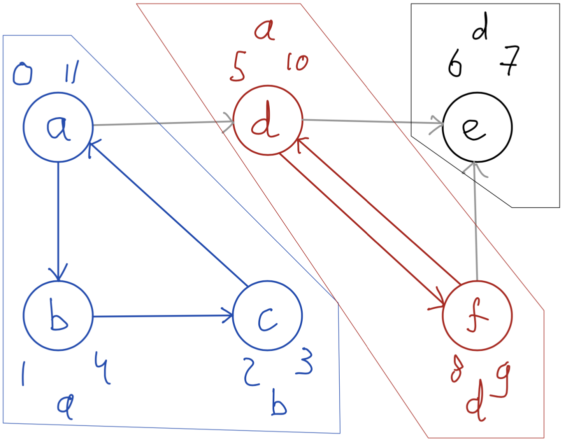
- The vertices are vertices of G
- We have an edge from v to w in G^{T} if and only if we have an edge from w to v in G

Algorithm for determining SCCs

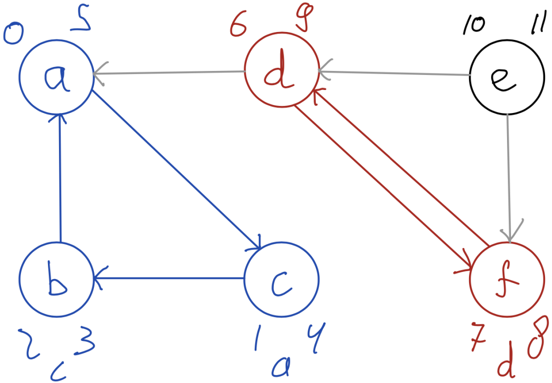
We start with a graph G

- Perform DFS
- Perform DFS on G^T. Go through the vertices in decreasing order based on the highest finishing time from the previous step
- This gives a forest
- The strongly connected components are the trees in that forest

Example



Example



Conclusion

- DFS: similar to BFS, but a different search order
- Iterative implementation using stacks or recursive implementation
- Important application of DFS: finding topological sorts.
- Other applications: we can use DFS to determine whether a graph is a DAG and to compute strongly connected components

Reading material: 8.4, 8.5, 8.6, and 8.7 in Roughgarden