CSCI3160 Design and Analysis of Algorithms (2025 Fall)

Bridging DP and Graphs: DAGs and Topological Sorting

Instructor: Xiao Liang

Department of Computer Science and Engineering
The Chinese University of Hong Kong

The order of subprolems

Let's recall the recursive formulas for the three DP problems we have learned:

- Rod cutting:

$$opt(n) = \max_{i=1}^{n}(P[i] + opt(n - i))$$

- Longest Increasing Subsequence:

$$len_{\mathrm{LIS}}(i) = 1 + \max_{\{j \mid j<i \text{ and } A[j]<A[i]\}} \left\{ len_{\mathrm{LIS}}(j) \right\}$$

- Edit Distance:

$$D(i, j) = \min \left\{ D(i, j - 1) + 1, \ D(i - 1, j) + 1, \ D(i - 1, j - 1) + [a_i == b_j] \right\}$$

DP could work because:

1. the subproblems can be solved in **a nice order** (i.e., previous solutions contribute to later subproblems)

2. such a nice order can be efficiently computed.

Dependency Graph of Dynamic-Programming Subproblems

Let's try to illustrated the relation between subproblems using the language of directed graphs:

- Consider a graph in which each node represents a subproblem.
- If solving subproblem sub_2 requires first solving subproblem sub_1, we draw a directed edge from sub_1 to sub_2.

This graph is known as the *dependency graph* of the subproblems in a dynamic programming (DP) algorithm.

Let's see some examples

For Rod Cutting

Recursion: $opt(n) = \max_{i=1}^{n}(P[i] + opt(n - i))$.
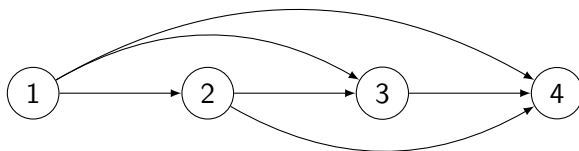
Figure: Exemplary Dependency Graph ($n = 4$)

For $i \in \{1, 2, 3, 4\}$, the i-th node represents the subproblem of computing $opt(i)$.

For Longest Increasing Subsequence

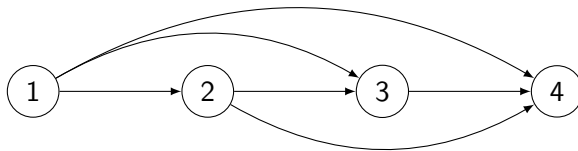**Recursion:** $len_{\text{LIS}}(i) = 1 + \max_{\{j \mid j < i \text{ and } A[j] < A[i]\}} \left\{ len_{\text{LIS}}(j) \right\}$.

Figure: Exemplary Dependency Graph ($n = 4$)

(It happens to be the same graph as for Rod Cutting)

For $i \in \{1, 2, 3, 4\}$, the i-th node represents the subproblem of computing $len_{\text{LIS}}(i)$.

For Edit Distance: 2D table of subproblem solutions

Recursion: $D(i,j) = \min\big\{D(i, j-1) + 1,\ D(i-1, j) + 1,\ D(i-1, j-1) + [a_i == b_j]\big\}.$

Figure: Exemplary 2-D Table of Subproblems ($n = 4$)

For $i, j \in \{0, 1, 2, 3, 4\}$, the (i, j)-th cell represents the subproblem of computing $D(i, j)$.

For Edit Distance: the dependency graph

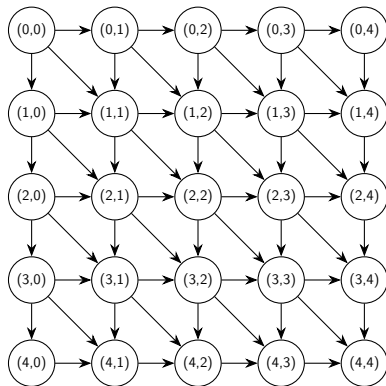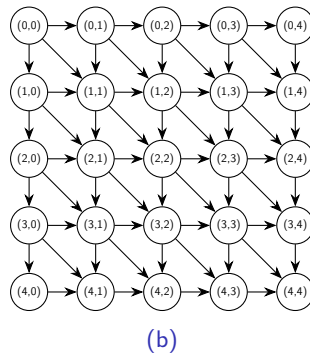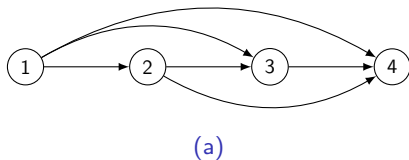**Recursion:** $D(i,j) = \min\left\{D(i,j-1) + 1,\ D(i-1,j) + 1,\ D(i-1,j-1) + [a_i == b_j]\right\}.$

Figure: Exemplary Dependency Graph ($n = 4$)

For $i, j \in \{0, 1, 2, 3, 4\}$, the (i,j)-th node represents the subproblem of computing $D(i,j)$.

On Dependency Graphs

(a)

(b)

A key feature: these dependency graphs do not contain cycles.

- Necessary for (efficiently) finding a nice order to solve these subproblems.

Such cycle-free directed graphs are called **Directed Acyclic Graphs** (DAGs).

DAGs and Topological Sorting

Definition

A **Directed Acyclic Graph (DAG)** is a finite directed graph with no directed cycles. That is, it consists of vertices connected by directed edges (arrows), and it is not possible to start at any vertex and follow a sequence of edges that eventually loops back to the starting vertex.

Moreover, the **nice order** we keep talking about can be translated as a way to arrange the nodes of a DAG into a sequence such that:

- for every directed edge $u \to v$, vertex u appears **before** v in the sequence.

This type of ordering is called a **topological ordering**.

The process of computing such an ordering is known as **topological sorting** (or **linearization**), where the goal is to take a DAG as input and produce a sequence of its vertices that satisfies the topological ordering condition.

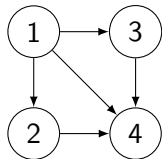# DAGs and Topological Sorting
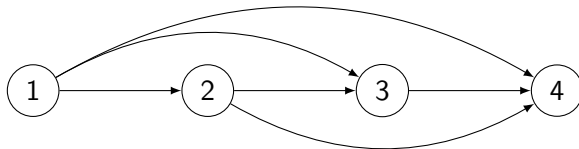
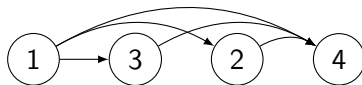It is clear to see that the topological order for this graph is just

(node 1, node 2, node 3, node 4)

It's less clear what is the topological order for the dependency graph of the **edit distance problem**.
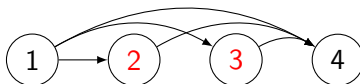
Let's start with a simpler example with a 2-by-2 grid graph:
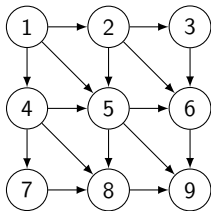
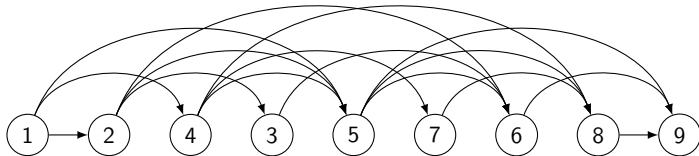

(a) Original graph

(b) Linearized (i)

(c) Linearized (ii)

Important: The topological ordering may **NOT** be unique!

A slightly larger example:

(a) Original graph

(b) Linearized (i)

Two Important Questions

Question 1: What kind of DAGs can be linearized (aka topologically sorted)?

- Answer: all of them! (think: why?)

Question 2: How to efficiently linearize a DAG?

- Answer: Depth First Search!

Next, we'll first recall the DFS algorithm, and then show how it can be utilized to do topological sorting.

Depth First Search (Review)

1

We now review the **Depth First Search** (DFS) algorithm (which was covered in CSCI2100). The algorithm is deceptively simple and has numerous non-trivial properties.

In this lecture, we will focus on a **black descendant property**[2] of DFS, which will help us perform topological sorting.

In a future lecture, we will introduce a **white path theorem**[3] for DFS, which will help us find the so-called **strongly connected components**.

---

[2]This term was coined by the course instructor and is not commonly used.
[3]This term was used in our textbook [CRLS].

The DFS Algorithm (1/2)

Algorithm description:

- Let $G = (V, E)$ be a directed simple[4] graph.

- In the beginning, color all vertices in the graph **white**.

- Create an empty tree T.   /* this will be called a "DFS tree" */

- Create a stack S, and then:
    - Pick an arbitrary vertex v
    - Push v into S, and color it **gray**   /* gray means "in the stack" */
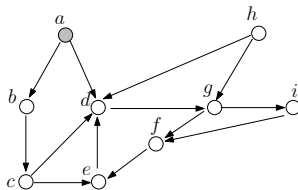   - Make v the root of T

(to be continued ...)

---

[4]Here, "simple" means no self-loops — i.e., no edge from a vertex to itself.

Example

Suppose that we start from *a*.

DFS tree
a

$S = (a)$.

The DFS Algorithm (2/2)

Repeat the following until S is empty.

1. Let v be the vertex that currently tops the stack S /* do not remove v from S yet */

2. Does v still have a white out-neighbor?

    2.1 If **YES**: let it be u.
- Push u into S and color u **gray**   /* gray means "in the stack" */
- Make u a child of v in the DFS-tree T.

    2.2 If **NO**: pop v from S and color v **black**   /* black means "this node is done" */

If there are still white vertices, repeat the above by restarting from an arbitrary white vertex v', creating a new DFS-tree rooted at v'.

(end of description)

Running time: DFS finishes in $O(|V| + |E|)$ time. (Think: why?)

Top of stack: a, which has white out-neighbors b, d. Suppose we access b first. Push b into S.
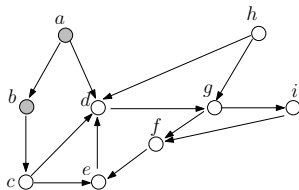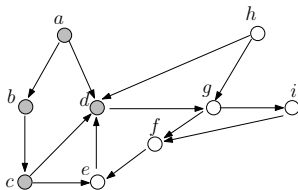

$S = (a, b)$.

Running Example

After pushing c into S:
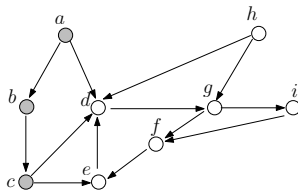
DFS tree
```
a
|
b
|
c
```

$S = (a, b, c)$.

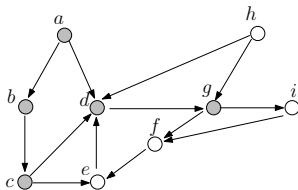Now c tops the stack. It has white out-neighbors d and e. Suppose we visit d first. Push d into S.

$S = (a, b, c, d)$.

After pushing g into S:

DFS tree
a
|
b
|
c
|
d
|
g

$S = (a, b, c, d, g)$.

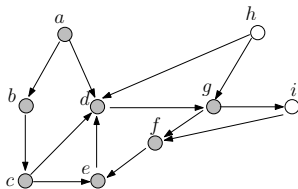Suppose we visit white out-neighbor f of g first. Push f into S

DFS tree
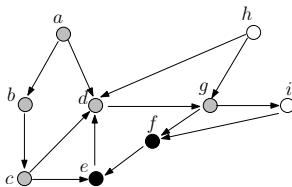
$S = (a, b, c, d, g, f)$.

After pushing e into S:
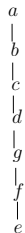

$S = (a, b, c, d, g, f, e)$.

Node e has no **white** out-neighbors. So pop it from S, and color it black.
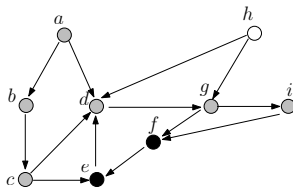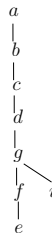Similarly, f has no **white** out-neighbors. Pop it from S, and color it black.

$S = (a, b, c, d, g)$.

(Running Example)

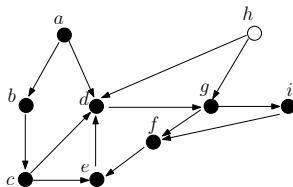Now g tops the stack again. It still has a white out-neighbor i. So, push i into S.

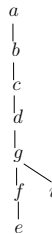$S = (a, b, c, d, g, i)$.

After popping i, g, d, c, b, a:
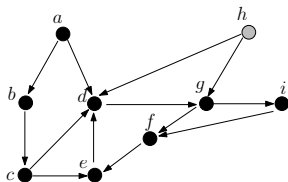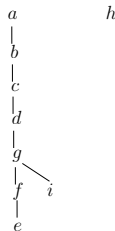

DFS tree

$S = ()$.

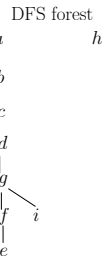Now there is still a white vertex h. So we perform another DFS starting from h.

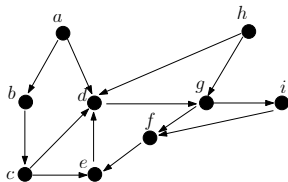
DFS forest

$S = (h)$.

Pop h. The end.

$S = ()$.

Note that we have created a DFS-forest, which consists of 2 DFS-trees.

Topological Sorting with DFS

How Can DFS Help?

Lemma

*During the execution of DFS, a node u is never colored **black** until all of its descendants (i.e., all nodes reachable from u) have already been colored **black**.*

Proof. This lemma follows directly from the algorithm description.

Corollary

During the execution of DFS on a DAG, if a node u is colored black after a node v, then u must appear before v in the topological ordering.

Proof. This follows directly from the above lemma and the definition of topological order.

Topological Sorting by DFS

The previous corollary directly yields an algorithm for topological ordering.

Input: a DAG G.

Output: a list of the nodes of G in topological order.

Algorithm:

- Run a DFS on G. During the traversal, use a counter to record the **finishing time** of each node — that is, the time at which the node is popped from the stack (i.e., when it is colored **black**).
- After the DFS completes, output the nodes in **decreasing** order of their **finishing times**.

Time Complexity: $O(|V| + |E|)$ (same as DFS).

Visual Example of DFS-based Topo Sort
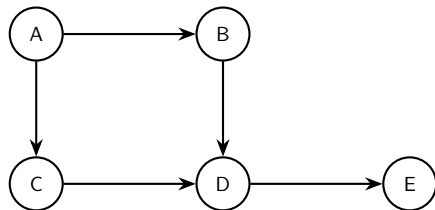
Example traversal:

- Starting node A.
- Visit A → C → D → E
- Then, **pop** E, **pop** D, **pop** C
- Backtrack and visit B → already visited D. Thus, **pop** B.
- Backtrack and **pop** A.
- Output order (reverse of finish): A, B, C, D, E
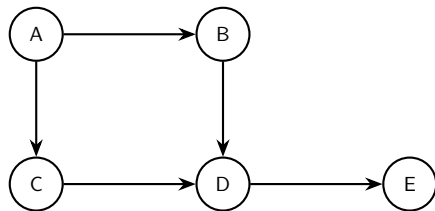
Visual Example of DFS-based Topo Sort
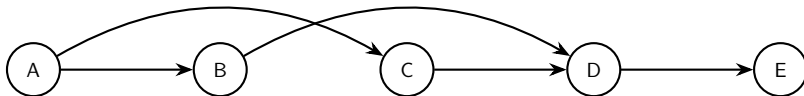

Figure: Original graph

Figure: Linearized graph

DFS vs BFS?

There aren't many problems where only DFS works but BFS does not.

- Topological sorting is one of them!

BFS Algorithm

Pseudocode for BSF:

```
1  Initialize all nodes as unvisited
2
3  Initialize an empty queue
4
5  For each node s in the graph:
6    If s is unvisited:
7      - Mark s as visited and enqueue it
8      - While the queue is not empty:
9          - Dequeue a node u
10         - For each unvisited neighbor v of u:
11             - Mark v as visited
12             - Enqueue v
```

Example: BFS Traversal
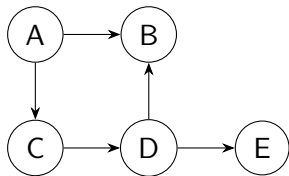
BFS starting at A:

- Dequeue A; Enqueue B and C
- Dequeue B;   /* no out-neighbor */
- Dequeue C; Enqueue D
- Dequeue D; Enqueue E   /* B has already been visited */
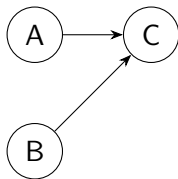- Dequeue E

Counterexample: When BFS Fails

Example DAG:

Figure: An example DAG

Valid topological orders: (A,B,C) or (B,A,C)

Invalid order: (A,C,B)   (Why? B → C exists but C appears before B)

Plain BFS May Fail

Suppose we run BFS starting from A:

- Enqueue A;
- Dequeue A; Enqueue C
- Dequeue C;
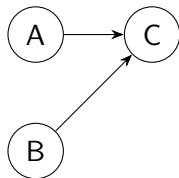- Enqueue B;   /* the queue is empty but B is not visited */
- Dequeue B

Figure: An example DAG

Why is this invalid?

- Enqueuing order? (A, C, B)   /* invalid topological order */
- Dequeuing order? (B, C, A)   /* invalid topological order */

Summary

Conclusion: DFS is more suitable than BFS for topological sorting

Caveat: Note that what we claimed is that the **plain** BFS is not suitable for topological sorting problem. But there do exist modified version of BFS (or BFS-based) algorithms that could solve this problem. One famous example is Kahn's algorithm.