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:

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

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

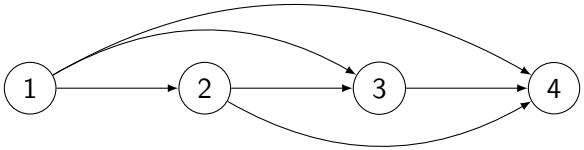


Figure: Exemplary Dependency Graph ($n = 4$)

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

For Longest Increasing Subsequence

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

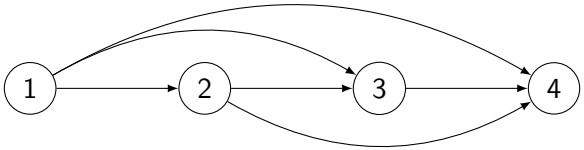


Figure: Exemplary Dependency Graph ($n = 4$)

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

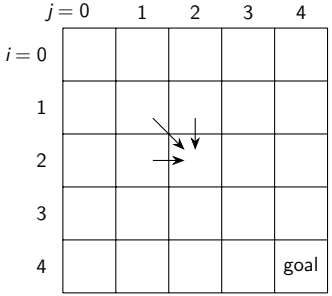


Figure: Exemplary 2-D Table of Subproblems ($n = 4$)

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

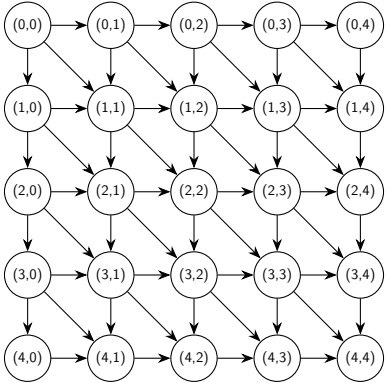
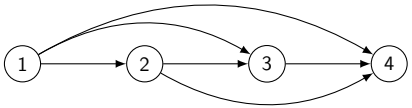


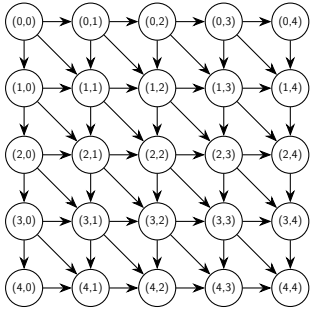
Figure: Exemplary Dependency Graph ($n = 4$)

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.

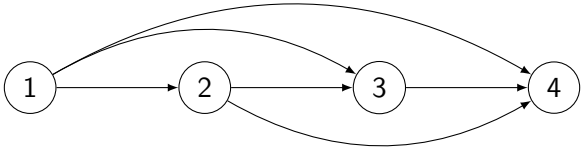
Moreover, the **nice order** we keep talking about can be translated as a way to arrange the nodes of a DAG into a sequence such that:

- 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

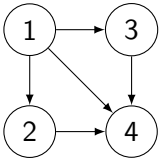


It is clear to see that the topological order for this graph is just

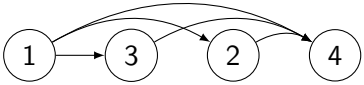
(node 1, node 2, node 3, node 4)

It's less clear what is the topological order for the dependency graph of the **edit distance problem**.

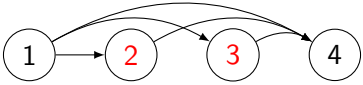
Let's start with a simpler example with a 2-by-2 grid graph:



(a) Original graph



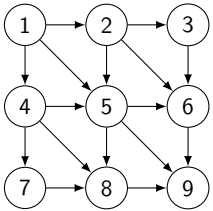
(b) Linearized (i)



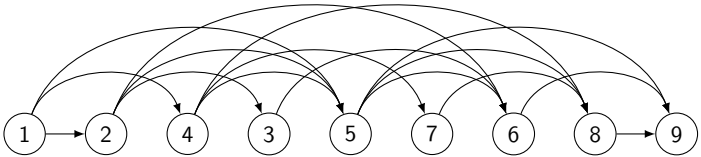
(c) Linearized (ii)

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

[1] This part is primarily based on materials prepared by Prof. Yufei Tao (please refer to Prof. Tao's version from 2024 Fall for the original content). Some modifications have been made to better align with this year's teaching progress, incorporating student feedback, in-class interactions, and my own teaching style and research perspective.

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 two properties of DFS, which will help us perform topological sorting.

In a future lecture, we will introduce a **white path theorem**[2] for DFS, which will help us find the so-called **strongly connected components**.

[2] This term was used in our textbook [CRLS].

The DFS Algorithm (1/2)

Algorithm description:

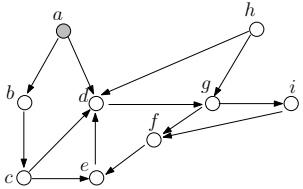
- Let $G = (V, E)$ be a directed simple[3] graph.
- In the beginning, color all vertices in the graph **white**.
- Create an empty tree T. /* this will be called a "DFS tree" */
- Create a stack S, and then:
  - Pick an arbitrary vertex v
  - Push v into S, and color it **gray** /* gray means "in the stack" */
  - Make v the root of T

(to be continued ...)

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

Example

Suppose that we start from a.



DFS tree

$S = (a)$.

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" */

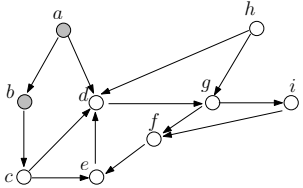
If there are still white vertices, repeat the above by restarting from an arbitrary white vertex v', creating a new DFS-tree rooted at v'.

(end of description)

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

Running Example

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

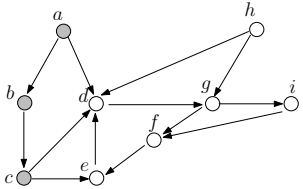


DFS tree

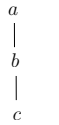
$S = (a, b)$.

Running Example

After pushing c into S:



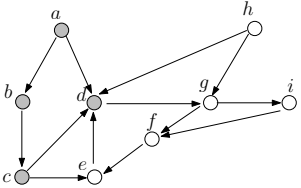
DFS tree

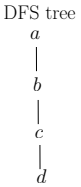


$S = (a, b, c)$.

Running Example

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

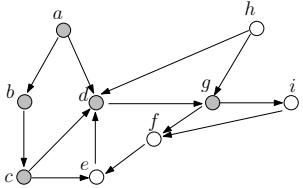




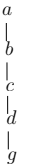
$S = (a, b, c, d)$.

Running Example

After pushing g into S:



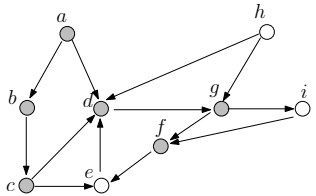
DFS tree



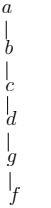
$S = (a, b, c, d, g)$.

Running Example

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



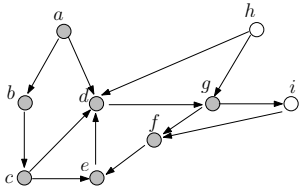
DFS tree



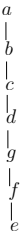
$S = (a, b, c, d, g, f)$.

Running Example

After pushing e into S:



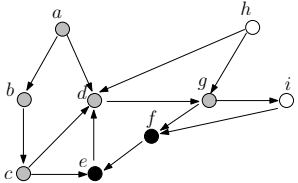
DFS tree

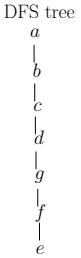


$S = (a, b, c, d, g, f, e)$.

Running Example

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.

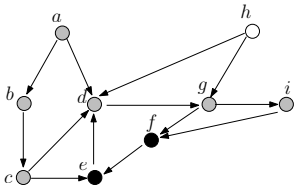


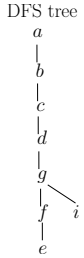


$S = (a, b, c, d, g)$.

Running Example

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

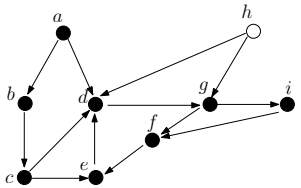


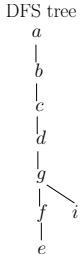


$S = (a, b, c, d, g, i)$.

Running Example

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

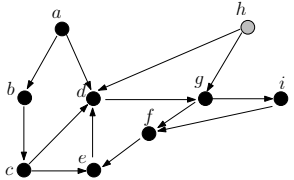


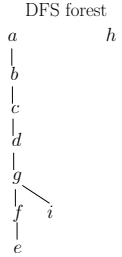


$S = ()$.

Running Example

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

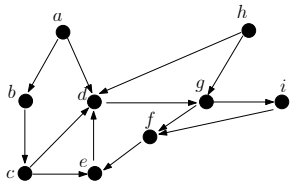


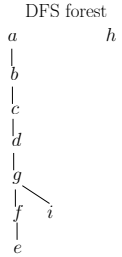


$S = (h)$.

Running Example

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 (1)

If a DFS execution on a DAG $G = (V, E)$ output k DFS trees in order: $(T_1, T_2, \ldots, T_k)$, then there is no (directed) edge (in the original edge set E) from any node of an earlier tree to any node of a later tree.

Proof. If there is a node u in an earlier tree T_i that has an out-neighbor v, then u must have already been included in

- the current DFS tree T_i; or
- some earlier tree T_m with $m < i$ (Think: can you come up with an example for this case?)

This is because the only reason that v is not added in T_i before we mark u as being done while building T_i, is that v is already done (i.e., marked black), which means that it must appear in an earlier tree T_m with $m < i$.

Moreover, observe that within each tree T_i, the nodes are already in topological order, in the following sense:

Lemma (2)

If a DFS execution on a DAG $G = (V, E)$ output k DFS trees in order: $(T_1, T_2, \ldots, T_k)$, then for each T_i, there is no (directed) edge (in the original edge set E) from a lower-level node to any upper-level node.

Proof. Otherwise, it contradicts the condition that G is acyclic.

Therefore, if we only want to produce a topological ordering of the nodes in T_i, we can simply sort them in ascending order of their levels. (The nodes within each level can appear in any order.)

Topological Sorting by DFS: Method 1

Lemma (1) and Lemma (2) inspires the following algorithm for topological sorting:

- First, run the DFS to build the DFS trees $T_1, \ldots, T_k$.
- Output the nodes in the following order:
  - At the tree level, nodes are output in descending order of their tree labels:

$$(\text{nodes in } T_k), (\text{nodes in } T_{k-1}), \ldots, (\text{nodes in } T_1)$$

  - Within each tree, (nodes in T_i) are output in ascending order of their levels. (The nodes within each level can appear in any order.)

(Think: it is easy to modify the DFS algorithm to output nodes in the described order.)

Topological Sorting by DFS: Method 2

There is another (only slightly different) DFS-based 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).

Proof of Correctness for Method 2

First, notice that Method 2 also (i.e., same as Method 1) output nodes of trees in descending order of their tree labels:

$$(\text{nodes in } T_k), (\text{nodes in } T_{k-1}), \ldots, (\text{nodes in } T_1),$$

because that nodes in later trees turn black later, and that Method 2 output nodes in the reversed order as they turn black.

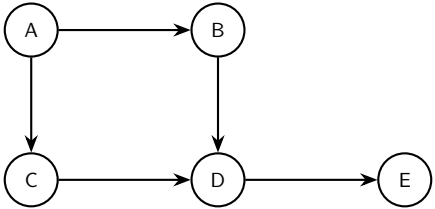
Thus, it suffices to prove that for each T_i, higher-level nodes must turn black earlier than lower-level nodes. Formally:

Lemma (3)

Consider an execution of DFS that output trees $(T_1, \ldots, T_k)$. During this execution, in any T_i, a node u was colored black only after all of its descendants in T_i have been colored black.

The proof is simply and left as an exercise.

Visual Example of DFS-based Topo Sort



Example traversal:

- Starting node A.
- Visit A $\rightarrow$ C $\rightarrow$ D $\rightarrow$ E
- Then, **pop** E, **pop** D, **pop** C
- Backtrack and visit B $\rightarrow$ 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

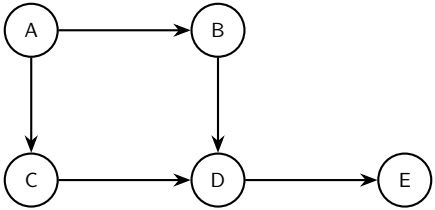


Figure: Original graph

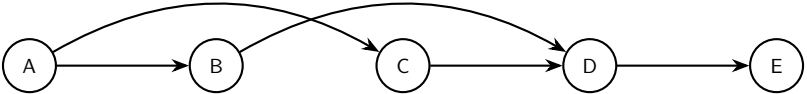


Figure: Linearized graph

DFS vs BFS?

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

- Topological sorting is one of them!

BFS Algorithm

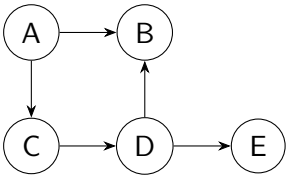
Pseudocode for BSF:

```
Initialize all nodes as unvisited

Initialize an empty queue

For each node s in the graph:
  If s is unvisited:
    - Mark s as visited and enqueue it
    - While the queue is not empty:
        - Dequeue a node u
        - For each unvisited neighbor v of u:
            - Mark v as visited
            - 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:

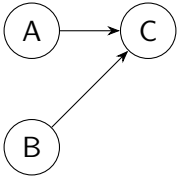


Figure: An example DAG

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

Invalid order: (A,C,B) (Why? B $\rightarrow$ 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

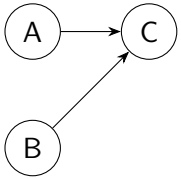


Figure: An example DAG

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.