Algorithms, Design & Analysis

Lecture 22: Longest Path Problem

Afrazia Umer Farooq & Qudsia Khan

Information Technology University

April 22, 2025

About Your Fellows

- Hello guys! **Afrazia** and **Qudsia Khan** here.
- We are Associate Students at ITU.

Overview

- Recap: Shortest Path Algorithms
- Longest Path Problem Statement
- NP-Hardness and Complexity
- Special Case: DAGs Solution
- Example Walkthrough

Recap: Shortest Path Algorithms

- **Dijkstra's**: Non-negative weights
- **Bellman-Ford**: Handles negative weights
- **Floyd-Warshall**: All-pairs shortest paths
- **DFS**: For unweighted graphs

Longest Path Problem

(Finding the Longest Path in a Directed Acyclic Graph (DAG))

Longest Path Problem Statement

Definition

Find a path in graph $G = (V, E, W)$ with maximum sum of edge weights

- Weights are non-negative
- Works for both directed/undirected graphs
- Start and end nodes can be arbitrary
- Only simple paths (no repeated nodes)

Characteristics

Key Characteristics:

- Path: A sequence of vertices with directed edges between them.
- Longest Path: Path with maximum total weight.
- DAGs are efficiently solvable due to absence of cycles.
- No vertex will be used more than once i.e no repeated vertex.

Difference between normal graphs and DAGs:

- **For general graphs:** Longest path problem is NP-hard because it does not have optimal structure property.
- Time complexity in general graphs is $\mathbf{N^p}$
  - **NP** (Non-deterministic Polynomial time): Class of problems where a solution can be **verified** in polynomial time.
  - **NP-Hard** problems are **at least as hard as the hardest NP problems**.
  - **No known fast (polynomial-time) algorithm** exists to solve NP-Hard problems in all cases.
- **For directed acyclic graphs (DAG):** Longest path problem has **linear time complexity**.

Solving Strategies

Solving Strategies:

There are two main approaches to solve the Longest Path Problem:

- **Topological Sorting Algorithm**
- **Dynamic Programming**

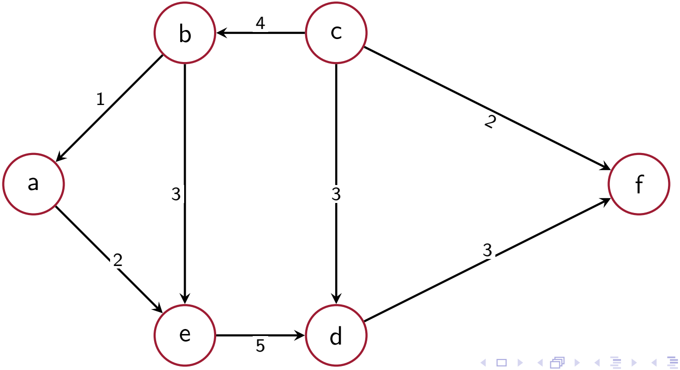
Topological Sorting Algorithm

Topological Sorting Algorithm

- **Topological sort:**
- Since DAG has no cycles, topological sort is possible.
- Topological order ensures that for every edge $(u \rightarrow v)$, u appears before v.

- **Example:**

  Let us consider the following graph:



Topological Sorting Algorithm

Distance from Source Node 'c'

Node	Distance
a	$-\infty$
b	$-\infty$
c	0
d	$-\infty$
e	$-\infty$
f	$-\infty$

- Source node c initialized to 0
- All other nodes set to $-\infty$ (unreachable)
- Will update distances during relaxation

Step 1: Start at Node c

Cumulative Weight at Node c:

- From node **c**, there is an edge to node **b** with weight 4.
- Cumulative weight: 4

Updated DP table:

$$\text{dp}[c] = 0, \quad \text{dp}[b] = 4, \quad \text{dp}[a] = -\infty, \quad \text{dp}[e] = -\infty, \quad \text{dp}[d] = -\infty, \quad \text{dp}[f] = -\infty$$

Step 2: Process Node b

Cumulative Weight at Node b:

- From node **b**, there is an edge to node **a** with weight 1.
- Cumulative weight: $4 + 1 = 5$

Updated DP table:

$$\text{dp}[c] = 0, \quad \text{dp}[b] = 4, \quad \text{dp}[a] = 5, \quad \text{dp}[e] = -\infty, \quad \text{dp}[d] = -\infty, \quad \text{dp}[f] = -\infty$$

Step 3: Process Node a

Cumulative Weight at Node a:

- From node **a**, there is an edge to node **e** with weight 2.
- Cumulative weight: $4 + 1 + 2 = 7$

Updated DP table:

$$\text{dp}[c] = 0, \quad \text{dp}[b] = 4, \quad \text{dp}[a] = 5, \quad \text{dp}[e] = 7, \quad \text{dp}[d] = -\infty, \quad \text{dp}[f] = -\infty$$

Step 4: Process Node e

Cumulative Weight at Node e:

- From node **e**, there is an edge to node **d** with weight 5.
- Cumulative weight: $4 + 1 + 2 + 5 = 12$

Updated DP table:

$$\text{dp}[c] = 0, \quad \text{dp}[b] = 4, \quad \text{dp}[a] = 5, \quad \text{dp}[e] = 7, \quad \text{dp}[d] = 12, \quad \text{dp}[f] = -\infty$$

Step 5: Process Node d

Cumulative Weight at Node d:

- From node **d**, there is an edge to node **f** with weight 3.
- Cumulative weight: $4 + 1 + 2 + 5 + 3 = 15$

Updated DP table:

$$\text{dp}[c] = 0, \quad \text{dp}[b] = 4, \quad \text{dp}[a] = 5, \quad \text{dp}[e] = 7, \quad \text{dp}[d] = 12, \quad \text{dp}[f] = 15$$

Step 6: Process Node f

Cumulative Weight at Node f:

- Node **f** has no outgoing edges.
- Cumulative weight: 15 (final weight)

Final DP table:

$$\text{dp}[c] = 0, \quad \text{dp}[b] = 4, \quad \text{dp}[a] = 5, \quad \text{dp}[e] = 7, \quad \text{dp}[d] = 12, \quad \text{dp}[f] = 15$$

Longest Path

Path Weights :

- $c \rightarrow b = 4$ $\quad (4)$
- $b \rightarrow a = 5$ $\quad (4+1)$
- $a \rightarrow e = 7$ $\quad (4+1+2)$
- $e \rightarrow d = 12$ $\quad (4+1+2+5)$
- $d \rightarrow f = 15$ $\quad (4+1+2+5+3)$

Longest Path:

$$\textbf{Total:} \quad 15 \text{ in } O(|V| + |E|)$$
$$c \rightarrow b \rightarrow a \rightarrow e \rightarrow d \rightarrow f$$

- Path shown with cumulative weights at each step
- Final path length matches your calculation
- **Time complexity** is $\boldsymbol{O(|V| + |E|)}$

Longest Path Problem

Dynamic Programing

Longest Path in DAG —DP (Step 1)

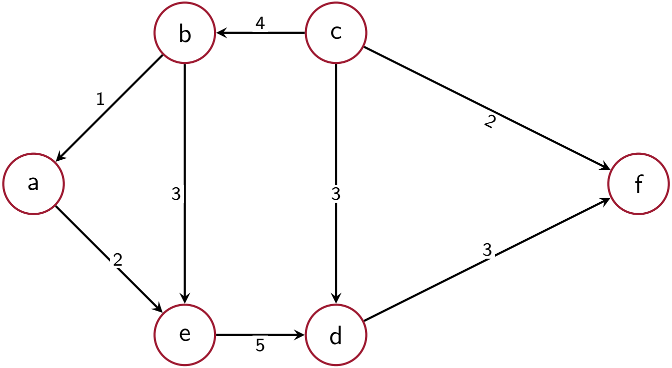
- **Goal:** Find the longest path in a Directed Acyclic Graph (DAG).
- **Step 1: Initialize**
  - For each node, create `lp[node]` $= 0$.
  - If a node has no outgoing edges, its longest path remains 0
- **Step 2: Maintain Incoming Edge Count**
  - Count how many incoming edges each node has.
  - This helps us know when a node is ready to be processed.
- **Step 3: Start Queue**
  - Add all nodes with no outgoing edges into the processing queue.

Longest Path in DAG — DP (Step 2)

- **Step 4: Process Nodes from Queue**
  - While the queue is not empty:
  - Take a node `u` from the queue.
  - For every node `v` that goes into `u` (i.e., there is an edge `v → u`):
    - Update: `lp[v] = max(lp[v], lp[u] + weight(v,u))`
    - Decrease incoming edge count of `v`.
    - If `v` now has no more outgoing edges, add it to the queue.
- **Step 5: Final Answer**
  - After the queue is empty, the longest path in the DAG is the maximum value in `lp[]` array.
- **Time Complexity:** $\mathcal{O}(V + E)$

Problem Overview

- **Goal:** Find the longest path in a Directed Acyclic Graph (DAG).



Initialization (Destination Node)

Goal: Find longest path that ends at node `f`

Initialization:

- $\text{lp}[f] = 0$ (start from destination)
- All other lp values $= -\infty$
- Initial queue = `[f]`

Processing Node `f`

Queue before: `[f]`

Pop: `f`

Incoming edges:

- $d \to f$ (weight 3)
- $c \to f$ (weight 2)

Updates:

- $\text{lp}[d] = 0 + 3 = 3$
- $\text{lp}[c] = 0 + 2 = 2$

Queue after: `[d, c]`

Processing Node `d`

Queue before: `[d, c]`

Pop: `d`

Incoming edges:

- $e \to d$ (weight 5)
- $c \to d$ (weight 3)

Updates:

- $\text{lp}[e] = 3 + 5 = 8$
- $\text{lp}[c] = \max(2, 3 + 3) = 6$

Queue after: `[c, e]`

Processing Node c

Queue before: `[c, e]`

Pop: `c`

Incoming edges: None

No updates

Queue after: `[e]`

Processing Node e

Queue before: `[e]`

Pop: `e`

Incoming edges:

- $a \to e$ (weight 2)
- $b \to e$ (weight 3)

Updates:

- $\text{lp}[a] = 8 + 2 = 10$
- $\text{lp}[b] = 8 + 3 = 11$

Queue after: `[a, b]`

Processing Node a

Queue before: `[a, b]`

Pop: `a`

Incoming edge: $b \rightarrow a$ (weight 1)

Update: $\text{lp}[b] = \max(11, 10 + 1) = 11$ (no change)

Queue after: `[b]`

Processing Node b

Queue before: [b]

Pop: b

Incoming edge: $c \to b$ (weight 4)

Update: $\text{lp}[c] = \max(6, 11 + 4) = 15$

Queue after: [c]

Processing Node c (again)

Queue before: [c]

Pop: c

Incoming edges: None

No updates

Queue after: [] (empty)

Final lp Values

- $\text{lp}[a] = 10$
- $\text{lp}[b] = 11$
- $\text{lp}[c] = \mathbf{15}$
- $\text{lp}[d] = 3$
- $\text{lp}[e] = 8$
- $\text{lp}[f] = 0$

Final Answer

- **Longest path ending at `f`: 15**
- **Path:** `c → b → a → e → d → f`

Final Answer

- **Longest path ending at `f`: 15**
- **Path:** `c → b → a → e → d → f`

Time Complexity

- **Time Complexity of the Longest Path Algorithm (DP):**
  - **Reverse Traversal:** Each node's incoming edges are checked once, so total edge processing is $O(E)$.
  - **Distance Updates:** Each node is processed once and updated based on its incoming neighbors.
  - **Queue Operations:** Each node and edge can contribute at most once to the queue, maintaining $O(V + E)$ complexity.
  - **Overall Time Complexity:** The dynamic programming approach still maintains:

$$O(V + E)$$

Conclusion

- **Recap:**
  - We used Dynamic Programming starting from node `f` and propagated backward through incoming edges.
  - The longest path ending at `f` is 15, and the correct path is `c` $\rightarrow$ `b` $\rightarrow$ `a` $\rightarrow$ `e` $\rightarrow$ `d` $\rightarrow$ `f`.
- **Key Takeaways:**
  - DP can be used when the endpoint (like `f`) is known and incoming edges are easy to trace.
  - The queue simulates the flow of longest distance calculations backward through the graph.
  - The time complexity remains $O(V + E)$, making this approach efficient for DAGs.

Relationship between Dynamic Programming questions and DAG

Why is any DP problem a DAG?

- Each subproblem depends only on smaller subproblems.
- The dependencies move in one direction — from smaller to larger.
- We can draw this as a graph:
  - Nodes = subproblems
  - Edges = dependencies
- There are no cycles — no subproblem depends on itself.

Therefore:

The structure of subproblems and their dependencies forms a **Directed Acyclic Graph (DAG)**.

Fibonacci as a DAG

Fibonacci recurrence:

$$F(n) = F(n-1) + F(n-2)$$

- To compute $F(5)$, we must compute $F(4)$ and $F(3)$.
- To compute $F(4)$, we need $F(3)$ and $F(2)$.
- To compute $F(3)$, we need $F(2)$ and $F(1)$.
- This dependency continues downward until base cases.

Insight

These recursive calls form a DAG — every node depends on smaller nodes, and there are no cycles.

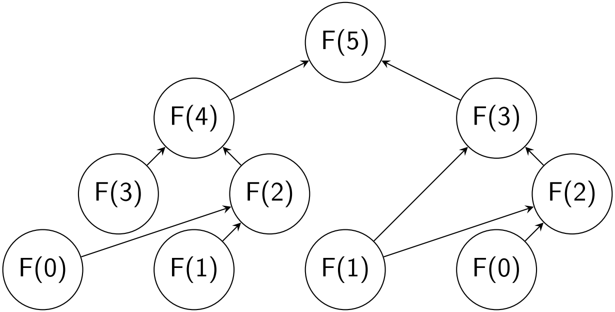
Problem Statement

- Compute $F(5)$ using reverse dynamic programming (bottom-up).
- Fibonacci is defined as:

$$F(0) = 0, \quad F(1) = 1, \quad F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2$$

- We'll process in order: $F(2), F(3), F(4), F(5)$, tracking computation and queue at each step.

Initialization



Initialization:

$$F(0) = 0, \quad F(1) = 1$$

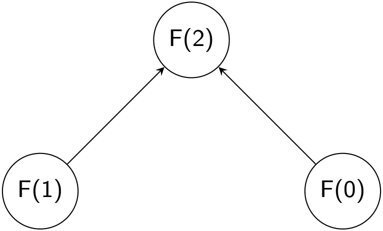
Queue: `[2, 3, 4, 5]`

Processing Node: F(2)

Queue before: `[2, 3, 4, 5]`
Pop: F(2)
Dependencies: $F(1) = 1$, $F(0) = 0$

$$F(2) = F(1) + F(0) = 1 + 0 = 1$$

Queue after: `[3, 4, 5]`

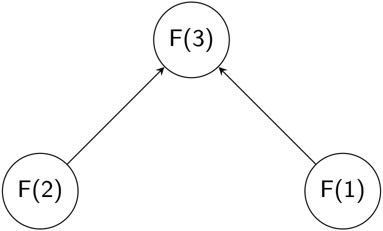


Processing Node: F(3)

Queue before: `[3, 4, 5]`
Pop: F(3)
Dependencies: F(2) = 1, F(1) = 1

$$F(3) = F(2) + F(1) = 1 + 1 = 2$$

Queue after: `[4, 5]`

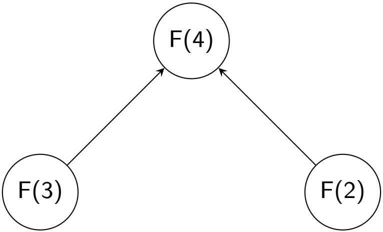


Processing Node: F(4)

Queue before: `[4, 5]`
Pop: F(4)
Dependencies: F(3) = 2, F(2) = 1

$$F(4) = F(3) + F(2) = 2 + 1 = 3$$

Queue after: `[5]`

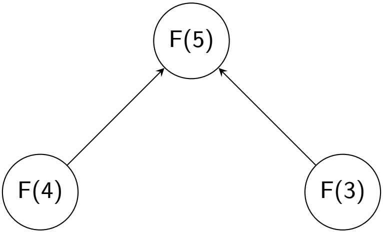


Processing Node: F(5)

Queue before: [5]
Pop: F(5)
Dependencies: F(4) = 3, F(3) = 2

$$F(5) = F(4) + F(3) = 3 + 2 = \boxed{5}$$

Queue after: [ ]



Final Fibonacci Table

F(i)	Value
F(0)	0
F(1)	1
F(2)	1
F(3)	2
F(4)	3
F(5)	**5**

Conclusion

- Each Fibonacci value was treated as a node in a DAG.
- The dependencies formed edges, and the problem was solved in bottom-up order using DP.
- This method avoids recomputation and mirrors DAG traversal with queue.
- Final result:

$$\boxed{F(5) = 5}$$

Practice Questions

Rod Cutting Problem

Problem:

Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n, determine the maximum value obtainable by cutting up the rod and selling the pieces.

Input: Integer n, and array $price[]$ of size n
Output: Maximum total price from cutting and selling the rod

Recurrence Relation:

$$DP[i] = \max_{1 \leq j \leq i} (price[j-1] + DP[i-j])$$

Painting Fence Problem

Problem:

Given n posts and k colors, find the number of ways to paint the posts such that no more than two adjacent posts have the same color.

Input: Integers n and k
Output: Number of valid ways to paint

Recurrence Relation:

$$DP[i] = (k-1) \times (DP[i-1] + DP[i-2])$$

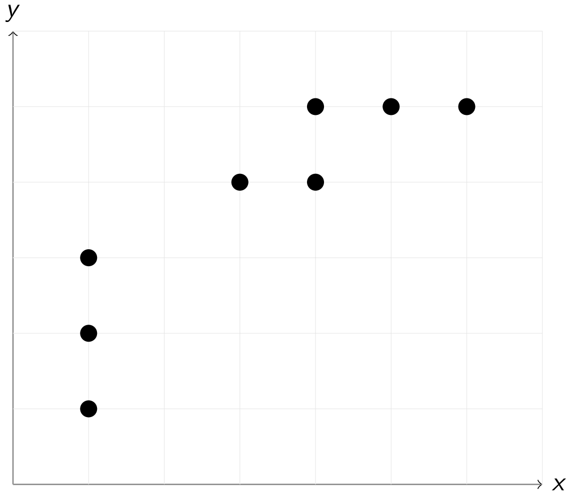
Max Coin Collection Grid Game

Game Summary

- You are given a grid with hidden coins placed on certain cells.
- There are two extractors:
  - Extractor — starts at bottom-left and moves **up**.
  - Extractor — starts at bottom-left and moves **right**.
- Allowed moves: move either extractor one step in its direction.
- When an extractor moves to a new row or column containing coins:
  - One coin is collected.
  - All other coins in that row or column are lost.
- The game ends when either extractor reaches the top or right edge.
- **Goal:** Plan your extractor moves to collect the **maximum number of coins**.

Initial Coin Grid

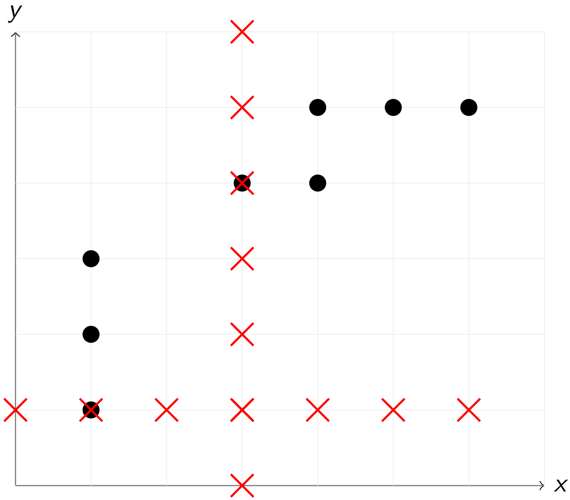
Start at (0,0). Coins (•), blocked cells (X):



No coins collected yet.

First Move

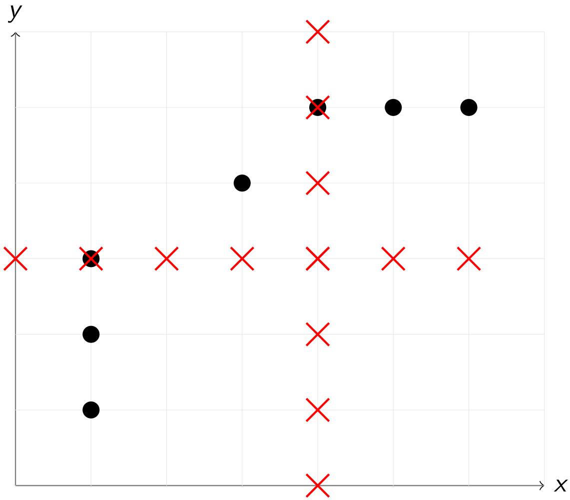
Pick coin at (1,3). Block row 1 and column 3.



Coins collected: (1,3)

Second Move

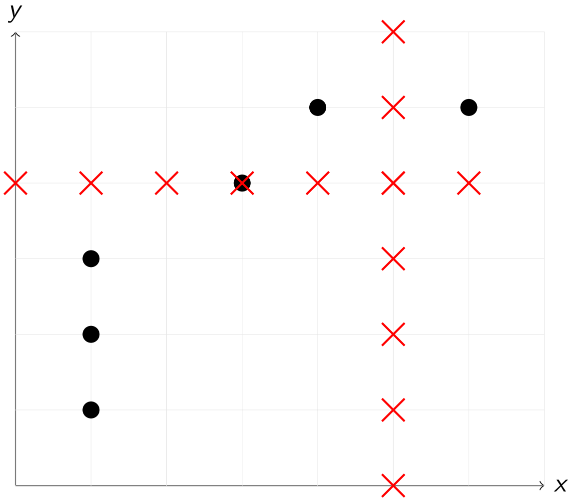
Pick coin at (3,4). Block row 3 and column 4.



Coins collected: (1,3), (3,4)

Third Move

Pick coin at (4,5). Block row 4 and column 5.



Coins collected: (1,3), (3,4), (4,5)

Game Summary

- Total coins collected: **3**
- Other coins blocked due to row/column restrictions.
- You reached a dead-end – further coins not accessible.

Q: When does a greedy approach fail in this game?