CSCI 470: Dynamic Programming

Vijay Chaudhary

November 13, 2023

Department of Electrical Engineering and Computer Science
Howard University

Overview

1. Bellman-Ford algorithm

2. Dynamic Programming

Bellman-Ford algorithm

Bellman-Ford algorithm

BELLMAN-FORD(G, w, s)

```
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3        for each edge (u, v) ∈ G.E
4              RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6        if v.d > u.d + w(u, v)
7              return FALSE
8  return TRUE
```

Lemma 24.2

Lemma 24.2

Let $G = (V, E)$ be a weighted, directed graph with source s and weight function $w : E \to \mathbb{R}$, and assume that G contains no negative-weight cycles that are reachable from s. Then, after the $|V| - 1$ iterations of the **for** loop of lines 2-4 of BELLMAN-FORD, we have $v.d = \delta(s, v)$ for all vertices v that are reachable from s.

Lemma 24.2: Proof

Proof We prove the lemma by appealing to the path-relaxation property.

Consider any vertex v that is reachable from s, and let $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = s$ and $v_k = v$, be any shortest path from s to v. Because shortest paths are simple, p has at most $|V| - 1$ edges, and so $k \leq |V| - 1$. Each of the $|V| - 1$ iterations of the **for** loop of lines 2-4 relaxes all $|E|$ edges. Among the edges relaxed in the i-th iteration, for $i = 1, 2, \ldots, k$, is (v_{i-1}, v_i). By the path-relaxation property, therefore, $v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$. □

Corollary 24.3

Corollary 24.3

Let $G = (V, E)$ be a weighted, directed graph with source vertex s and weight function $w : E \to \mathbb{R}$, and assume that G contains no negative-weight cycles that are reachable from s. Then, for each vertex $v \in V$, there is a path from s to v **if and only if** BELLMAN-FORD terminates with $v.d < \infty$ when it is run on G.

Proof is left as an exercise for HW 5.

Correctness of Bellman-Ford algorithm

Let BELLMAN-FORD be run on a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \to \mathbb{R}$. If G contains no negative-weight cycles that are reachable from s, then the algorithm returns TRUE, we have $v.d = \delta(s, v)$ for all vertices $v \in V$, and the predecessor subgraph G_π is a shortest-paths tree rooted at s. If G does contain a negative-weight cycles reachable from s, then the algorithm returns FALSE.

Proof: Case 1

Case 1:
Suppose that graph G contains no negative-weight cycles that are reachable from the source s. We first prove the claim that at termination, $v.d = \delta(s, v)$ for all vertices $v \in V$.

- If vertex v is reachable from s, then Lemma 24.2 proves this claim.
- If v is not reachable from s, then the claim follows from the no-path property. Thus, the claim is proven.

Proof: Case 1

- The predecessor-subgraph property, along with the claim, implies that G_π is a shortest-paths tree.
- Now we use the claim to show that BELLMAN-FORD returns TRUE. At termination, we have for all the edges $(u, v) \in E$,

$$\begin{aligned} v.d &= \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) \\ &= u.d + w(u, v), \end{aligned}$$

  and so none of the tests in line 6 causes BELLMAN-FORD to return FALSE. Therefore, it returns TRUE.

Proof: Case 2

Case 2:
Now, suppose that graph G contains a negative-weight cycle that is reachable from the source s; let this cycle be $c = \langle v_0, v_1, ..., v_k \rangle$, where $v_0 = v_k$. Then,

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0. \tag{1}$$

Proof: Case 2

- Assume for the purpose of contradiction that the Bellman-Ford algorithm returns TRUE.
- Thus, $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$. Summing the inequalities around cycle c gives us

$$\begin{aligned}\sum_{i=1}^{k} v_i.d &\leq \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)\end{aligned}$$

Proof: Case 2

- Since $v_0 = v_k$, each vertex in c appears exactly once in each of the summations $\sum_{i=1}^{k} v_i.d$ and $\sum_{i=1}^{k} v_{i-1}.d$, and so

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d.$$

- Moreover, by Corollary 24.3, $v_i.d$ is finite for $i = 1, 2, \ldots, k$. Thus,

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i),$$

  which contradicts inequality (4).

Proof

We conclude that the BELLMAN-FORD algorithm returns TRUE if graph G contains no negative-weight cycles reachable from the source, and FALSE otherwise. □

Dynamic Programming

Dynamic Programming: Fibonacci

$F_0 = 0, F_1 = 1$

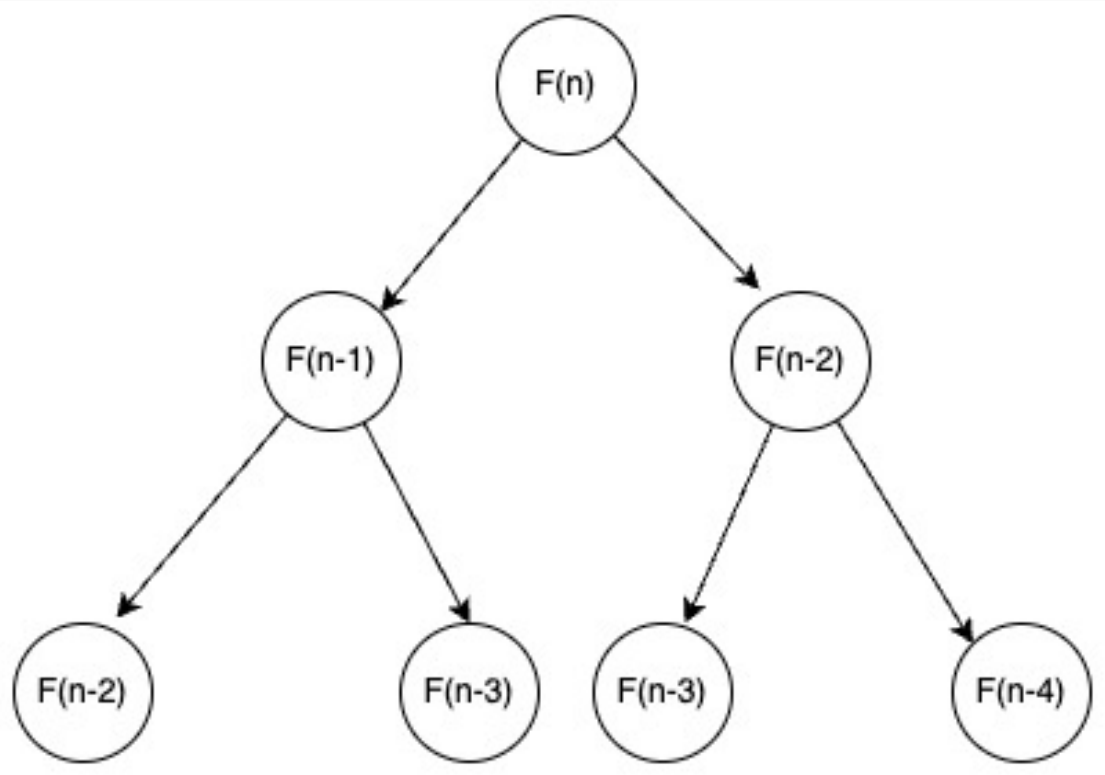
$F_n = F_{n-1} + F_{n-2}$ for $n > 1$

- $F_2 = F_1 + F_0 = 0 + 1 = 1$
- $F_3 = F_2 + F_1 = 1 + 1 = 2$
- $F_4 = F_3 + F_2 = 2 + 1 = 3$
- $F_5 = F_4 + F_3 = 3 + 2 = 5$
- $F_6 = F_5 + F_4 = 5 + 3 = 8$

Recursive Top-Down Implementation

```
TOP-DOWN-FIB(n)
1  if n == 0 or n == 1
2        return n
3  return TOP-DOWN-FIB(n − 1) + TOP-DOWN-FIB(n − 2)
```

Issues with this implementation



- Repeated calculations
- For instance, to calculate F_3, we need F_2 and F_1. However, the recursion will calculate F_2 twice.

Sub-problems

- Is there a way we can avoid computing repeated calculations?

Sub-problems

- Is there a way we can avoid computing repeated calculations?
- Can we keep a *memo* of results of sub-problems?

Top-Down with Memoization

- In this approach, we write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array or hash table).

Top-Down with Memoization

- In this approach, we write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array or hash table).
- The procedure now first checks to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level; if not, the procedure computes the value in the usual manner.

Top-Down with Memoization

- In this approach, we write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array or hash table).
- The procedure now first checks to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level; if not, the procedure computes the value in the usual manner.
- We say that the recursive procedure has been memoized; it "remembers" what results it has computed previously.

Top-Down with Memoization: Fib

```
MEMOIZED-FIB(memo, n)
1  if n == 0 or n == 1
2        return n
3  if n ∈ memo
4        return memo[n]
5  res = MEMOIZED-FIB(memo, n − 1) + MEMOIZED-FIB(memo, n − 2)
6  memo[n] = res
7  return res
```

Bottom-up Approach

- This approach typically depends on some natural notion of the "size" of a subproblem, such that solving any particular subproblem depends only on solving "smaller" subproblems.

Bottom-up Approach

- This approach typically depends on some natural notion of the "size" of a subproblem, such that solving any particular subproblem depends only on solving "smaller" subproblems.
- We sort the subproblems by size and solve them in size order, smallest first.

Bottom-up Approach

- This approach typically depends on some natural notion of the "size" of a subproblem, such that solving any particular subproblem depends only on solving "smaller" subproblems.
- We sort the subproblems by size and solve them in size order, smallest first.
- When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions.

Bottom-up Approach

- This approach typically depends on some natural notion of the "size" of a subproblem, such that solving any particular subproblem depends only on solving "smaller" subproblems.
- We sort the subproblems by size and solve them in size order, smallest first.
- When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions.
- We solve each subproblem only once, and when we first see it, we have already solved all of its prerequisite subproblems.

Bottom-up: Fib

BOTTOM-UP-FIB(n)

```
1  let r[0, ..., n] be a new array
2  r[0] = 0 and r[1] = 1
3  if n == 0 or n == 1
4        return r[n]
5  for i = 2 to n
6        r[i] = r[i − 1] + r[i − 2]
7  return r[n]
```

Classwork: House Robber

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and it will automatically contact the police if two adjacent houses were broken into on the same night.
Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

Classwork: House Robber II

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. **All houses at this place are arranged in a circle.** That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have a security system connected, and it will automatically contact the police if two adjacent houses were broken into on the same night.
Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.