Dynamic Programming

CMPUT 275 - Winter 2018

University of Alberta

Outline

- Recursion and Memoization
  - Can we make brute force run in polynomial time?
- Top-Down vs. Bottom-Up Approaches
- DP Examples
  - Fibonacci Sequence Problem
  - Shortest Paths Problem
  - Text Justification Problem
  - Knapsack Problem

What is Dynamic Programming (DP)?

What is Dynamic Programming (DP)?

A general powerful algorithm design technique for solving optimization problems and other problems that can be solved using recursion

A carefully executed exhaustive search (brute force search) algorithm which can be polynomial time

A family of algorithms based on recursion and memoization (to avoid recomputing subproblems)

What is Dynamic Programming (DP)?

A general powerful algorithm design technique for solving optimization problems and other problems that can be solved using recursion

A carefully executed exhaustive search (brute force search) algorithm which can be polynomial time

A family of algorithms based on recursion and memoization (to avoid recomputing subproblems)

Basic Idea
(a) Divide a problem into subproblems,
(b) Solve these subproblems,
(c) Reuse their solutions to solve the original problem

Examples: Fibonacci Numbers

Fibonacci Numbers: 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Each number is the sum of the previous two, except the first two which we just state explicitly. A compact way to state this is

$$F_n = \begin{cases} 1, & \text{if } n \leq 2\,; \\ F_{n-1} + F_{n-2}, & \text{otherwise}\,. \end{cases}$$

Goal: Compute F_n

Naive Recursive Algorithm

```python
def fib(n):
    if n <= 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Naive Recursive Algorithm

```python
def fib(n):
    if n <= 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Observe that *fib*(n) is computed repeatedly for many values n.

$$F_n$$

$$F_{n-1} \qquad\qquad F_{n-2}$$

$$\mathbf{F_{n-2}} \qquad F_{n-3} \qquad\qquad \mathbf{F_{n-3}} \qquad F_{n-4}$$

Running Time Analysis

Let T(n) be the time that it takes to compute fib(n)

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + O(1) \\ &\geq 2T(n-2) \\ &= O(2^{n/2}) \end{aligned}$$

Takes exponential time, about ϕ^n where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.

Memoized (Top-Down) DP Algorithm

Improve the running time by using a dictionary (memo) to store the Fibonacci numbers as we compute them so we only have to use the recurrence once for each value of *n*.

```
def fib(n, memo):
    if memo is None:
        memo = dict()
    if n in memo:
        return memo[n]        // reuse
    if n <= 2:
        f = 1
    else:
        f = fib(n-1, memo) + fib(n-2, memo)
    memo[n] = f               // memoize
    return f
```

You can apply the same procedure to any recursive algorithm

Running Time Analysis

Analyzing the running time is fairly straightforward. Usually the answer is the number of possible distinct subproblems times the time it takes to compute each subproblem. The only caveat is to treat recursive calls as constant time operations to avoid double counting.

Running Time Analysis

Analyzing the running time is fairly straightforward. Usually the answer is the number of possible distinct subproblems times the time it takes to compute each subproblem. The only caveat is to treat recursive calls as constant time operations to avoid double counting.

For example, with the Fibonacci numbers one could say the running time is $O(n)$ because there are n values computed and each call takes constant time. This is not quite accurate because the numbers grow so quickly that it is not fair to say addition takes constant time. The truth is that it really takes $O(n^2)$ time due to the cost of addition.

Bottom-Up DP Algorithm

```python
def DP():
    fib = [None] * n
    for k in range(1, n+1):
        if k <= 2: f = 1
        else: f = fib[k-1] + fib[k-2]
        fib[k] = f
    return fib[n]
```

It is a bit more efficient since we got rid of recursive calls (still $O(n)$ though).

We do not need to store the whole list, we just need the last two numbers. So it can be implemented using constant memory!

Example: Shortest Paths (Fixed Source)

Input: A DAG denoted $G(V, E)$, weight of edge (s, s') is $w(s, s')$
Goal: Compute shortest path from s to $v \;\; \forall v \in V$

Example: Shortest Paths (Fixed Source)

Input: A DAG denoted $G(V, E)$, weight of edge (s, s') is $w(s, s')$
Goal: Compute shortest path from s to $v \;\; \forall v \in V$
Subproblem: $delta(s, s')$: weight of the shortest path from s to s'

Example: Shortest Paths (Fixed Source)

Input: A DAG denoted $G(V, E)$, weight of edge (s, s') is $w(s, s')$
Goal: Compute shortest path from s to $v \ \ \forall v \in V$
Subproblem: $delta(s, s')$: weight of the shortest path from s to s'
Guess: the node directly connected to v on this shortest path
Try all nodes connected to v and remember the best one

$$s - - > \cdots - - > u - - > v$$

Example: Shortest Paths (Fixed Source)

Input: A DAG denoted $G(V, E)$, weight of edge (s, s') is $w(s, s')$
Goal: Compute shortest path from s to $v \;\; \forall v \in V$
Subproblem: $delta(s, s')$: weight of the shortest path from s to s'
Guess: the node directly connected to v on this shortest path
Try all nodes connected to v and remember the best one

$$s - - > \cdots - - > u - - > v$$

Recursive Relation:

$$delta(s, v) = \begin{cases} 0, & \text{if} \quad s = v \\ \min_{(u,v) \in E} \Big(delta(s, u) + w(u, v) \Big), & \text{otherwise}\,. \end{cases}$$

Example: Shortest Paths (Fixed Source)

Input: A DAG denoted $G(V, E)$, weight of edge (s, s') is $w(s, s')$
Goal: Compute shortest path from s to $v \ \ \forall v \in V$
Subproblem: $delta(s, s')$: weight of the shortest path from s to s'
Guess: the node directly connected to v on this shortest path
Try all nodes connected to v and remember the best one

$$s - - > \cdots - - > u - - > v$$

Recursive Relation:

$$delta(s, v) = \begin{cases} 0, & \text{if} \quad s = v \\ \min_{(u,v) \in E} \Big(delta(s, u) + w(u, v) \Big), & \text{otherwise}\,. \end{cases}$$

s is the same, so we just need to use v as the key in the dictionary

$$memo[v] = delta(s, v)$$

Running Time Analysis

Running time of subproblem delta(s, v) is indegree(v) + 1

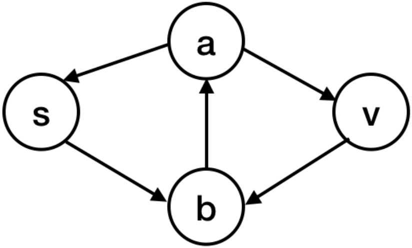
Total time:

$$\sum_{v \in V} indegree(v) + 1 = O(|E| + |V|)$$

Without memoization, this would be an exponential time algorithm

General Shortest Path Algorithm (Fixed Source)

If G has a cycle, we might get stuck in an infinite loop because a subproblem of $delta(s, u)$ can be $delta(s, v)$



$$delta(s, v) --> delta(s, a) --> delta(s, a) --> delta(s, v)$$

General Shortest Path Algorithm (Fixed Source)

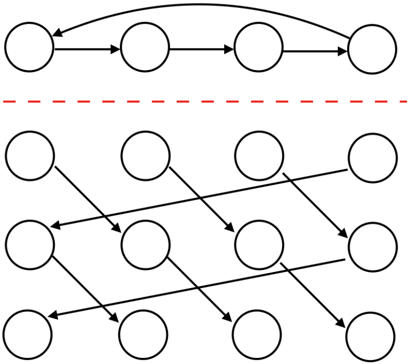
Basic Idea

- Create k copies of the graph with cycles
- All edges go from one layer to the next layer, making the graph acyclic

General Shortest Path Algorithm (Fixed Source)

Basic Idea

- Create k copies of the graph with cycles
- All edges go from one layer to the next layer, making the graph acyclic



General Shortest Path Algorithm (Fixed Source)

Subproblem: $delta_k(s, v)$: weight of shortest path from s to v that uses at most k edges
Number of subproblems : $|V|^2$ (as $0 \leq k < |V|$)

General Shortest Path Algorithm (Fixed Source)

Subproblem: $delta_k(s, v)$: weight of shortest path from s to v that uses at most k edges
Number of subproblems : $|V|^2$ (as $0 \leq k < |V|$)
Recursive Relation:

$$delta_k(s, v) = min_{(u,v) \in E}(delta_{k-1}(s, u) + w(u, w))$$

General Shortest Path Algorithm (Fixed Source)

Subproblem: $delta_k(s, v)$: weight of shortest path from s to v that uses at most k edges
Number of subproblems : $|V|^2$ (as $0 \leq k < |V|$)
Recursive Relation:

$$delta_k(s, v) = min_{(u,v) \in E}(delta_{k-1}(s, u) + w(u, w))$$

Original Problem: $delta_{|V|-1}(s, v)$

General Shortest Path Algorithm (Fixed Source)

Subproblem: $delta_k(s, v)$: weight of shortest path from s to v that uses at most k edges
Number of subproblems : $|V|^2$ (as $0 \leq k < |V|$)
Recursive Relation:

$$delta_k(s, v) = min_{(u,v) \in E}(delta_{k-1}(s, u) + w(u, w))$$

Original Problem: $delta_{|V|-1}(s, v)$
Running time of subproblem delta(s, v) is indegree(v)
Total running time

$$\sum_{v \in V} indegree(v) = O(|V| \cdot |E|)$$

5 Steps to Solve Dynamic Programming Problems

1. Define subproblems
   1.1 Determine their count
2. Guess part of the solution
   2.1 Determine the number of choices
3. Related subproblem solutions
4. Recurse and memoize (top-down approach) or build a table (bottom-up approach)
   4.1 Compute the time per subproblem
   4.2 Check if the subproblem recurrence is acyclic
   4.3 Compute the total time
5. Solve the original problem by combining the solutions to subproblems; this might take some extra time

Example: Text Justification

Input: A sequence of n words with different lengths
Goal: Split input text into “good” lines with as small gaps as possible

Example: Text Justification

Input: A sequence of n words with different lengths
Goal: Split input text into "good" lines with as small gaps as possible
Define a "badness" measure: how bad it is to create a line using words[i:j]

$$badness = \begin{cases} (pagewidth - totalwidth)^3, & \text{if words[i:j] fit in a line}\,; \\ \infty, & \text{otherwise}\,. \end{cases}$$

Why cubed? This is the latex rule!

Example: Text Justification – Cont'd

Goal: Minimize the sum of badness measures of the lines

Subproblem: Just(i) the sum of badness measures of the remaining lines given words[i:]

Guess: word starting of the second line
Number of choices: at most $n - i$ which is O(n)

Recurrence Relation: Just(n): return 0
Just(i): return $\min_{j \text{ in } range(i+1,n+1)}[Just(j) + badness(i,j)]$

Original Problem: Just(0)

Running Time Analysis

Time per subproblem is O(n)
The subproblem recurrence is acyclic
Total time $O(n^2)$

Running Time Analysis

Time per subproblem is O(n)
The subproblem recurrence is acyclic
Total time $O(n^2)$

Brute force algorithm:
Check whether each word can at the beginning of a line
Running time: 2^n if we have n words

Reconstructing the solution

To combine solutions to subproblems we must remember which guess was the best (the argument in optimization)

So we store the best guess for each subproblem.

$$parent[i] = argmin(Just(j) + badness(i, j))$$

Thus, the lines start with the following words:

$$0, parent[0], parent[parent[0]], parent[parent[parent[0]]], \cdots$$

Example: Knapsack (How to pack?)

We are given a collection of n items. Item i has a value $v_i \geq 0$ and a weight $w_i \geq 0$. We want to pack or knapsack so that the value of items packed is the largest possible while the total weight of packed items does not exceed a maximum value, or capacity $C \geq 0$. Formally, the goal is to find a subset of items $S \subset \{1, \ldots, n\}$ with maximum possible value such that the total weight of S does not exceed C.

For a given subset S, let $v(S) = \sum_{j \in S} v_j$ be their total value and let $w(S) = \sum_{j \in S} w_j$ be their total weight. With this we can write that the goal is to calculate $v^*(n, C)$ where for any $1 \leq k \leq n$, $c \geq 0$ we define

$$v^*(k, c) \doteq \max\{v(S) \,|\, S \subset \{1, \ldots, k\}, w(S) \leq c\}\,.$$

Knapsack – A Dynamic Programming Approach

Goal: Maximize the sum of values for a subset of items with size $\leq C$

Knapsack – A Dynamic Programming Approach

Goal: Maximize the sum of values for a subset of items with size $\leq C$

Subproblem: Knapsack(i, c)
Number of subproblems: O(n.C)
Guess: Include the ith item or not (two choices)

Knapsack – A Dynamic Programming Approach

Goal: Maximize the sum of values for a subset of items with size $\leq C$

Subproblem: Knapsack(i, c)
Number of subproblems: O(n.C)
Guess: Include the ith item or not (two choices)

Recurrence Relation:

$$Knapsack(i, c) = \max(Knapsack(i+1, c-w_i)+v_i, Knapsack(i+1, c))$$

Running time per subproblem: O(1)
Total running time: O(n.C)
Without memoization, the solution is not better than brute-force enumeration of all possibilities which is an exponential algorithm

Original Problem: Knapsack(0, C)

Summary

The main difficulty with solving a problem via dynamic programming is finding the right recurrence (i.e., designing the subproblems).