Dynamic Programming

CMPUT 275 - Winter 2018

University of Alberta

Outline

- Memoize Decorator
- Dynamic Programming Examples
    - Knapsack
    - Make Change
    - Maximum Contiguous Sum
    - Edit Distance

Memoizing a Function

How to write a function which could be used to memoize any function that is called recursively?

Memoizing a Function

How to write a function which could be used to memoize any function that is called recursively?

Basic Idea: write a wrapper function which stores (in a dictionary) the result of calling this function with specific input parameters

It can be done using function decorators in Python

Memoizing a Function

How to write a function which could be used to memoize any function that is called recursively?

Basic Idea: write a wrapper function which stores (in a dictionary) the result of calling this function with specific input parameters

It can be done using function decorators in Python

@functools.lru_cache(max_size=1024) is a decorator function that already exists and can be used to wrap a function with a cache which stores the max_size most recent function call values

5 Steps to Solve Dynamic Programming Problems

1. Define subproblems
2. Guess part of the solution
3. Related subproblem solutions
4. Recurse and memoize (top-down approach) or create a table (bottom-up approach)
5. Solve the original problem by combining solutions of the subproblems

5 Steps to Solve Dynamic Programming Problems

1. Define subproblems
2. Guess part of the solution
3. Related subproblem solutions
4. Recurse and memoize (top-down approach) or create a table (bottom-up approach)
5. Solve the original problem by combining solutions of the subproblems

Running Time Analysis Template
Running time = (# of distinct subproblems we need to solve) $\times$ (computation required to solve each subproblem) + (computation required to combine solutions to solve the original problem)

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 a knapsack so that the value of items packed is the largest possible while the total weight of packed items does not exceed a maximum capacity $C \geq 0$ (items are indivisible). 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 $w(S) = \sum_{j \in S} w_j$ be their total weight. 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 total size $\leq C$

Knapsack – A Dynamic Programming Approach

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

Subproblem: $\mathit{Knapsack}(i, c)$: the maximum total value when packing a knapsack of size c considering items $\{1, \ldots, i\}$
Number of subproblems: $O(n.C)$

Guess: Include the ith item in the knapsack or not

Knapsack – A Dynamic Programming Approach

Recurrence Relation:

$$Knapsack(i, c) = \begin{cases} 0, & \text{if } i = 0, \\ Knapsack(i-1, c), & \text{if } w_i > c, \\ \max(Knapsack(i-1, c-w_i) + v_i, \\ \qquad Knapsack(i-1, c)) & \text{otherwise.} \end{cases}$$

Running Time Analysis:
Number of subproblems: $O(n.C)$
Running time per subproblem: $O(1)$
Total running time: $O(n.C)$

Original Problem: $Knapsack(n, C)$

Top-Down Approach

```
def knapsack(k, capacity, memo=None):
    if memo is None:
        memo = {}
    if (k,capacity) in memo:
        return memo[(k,capacity)]
    if k==0:
        ret = 0 if sizes[0] > capacity else values[0]
        memo[(k,capacity)] = ret
        return ret
    max_value = knapsack(k-1, capacity, memo)
    if sizes[k] <= capacity:
        packing_value = values[k] + knapsack(k-1, capacity-sizes[k], memo)
        max_value = max(max_value, packing_value)
    memo[(k,capacity)] = max_value

    return max_value
```

Bottom-Up Approach

```
def bottom_up_knapsack(k, capacity):
    total = [[0 for x in range(capacity+1)] for x in range(k+1)]
    for i in range(n+1):
        for w in range(capacity+1):
            if i==0 or w==0:
                total[i][w] = 0
            elif sizes[i-1] <= w:
                total[i][w] = max(vals[i-1] + total[i-1][w-sizes[i-1]],  total[i-1][w])
            else:
                total[i][w] = total[i-1][w]

    return total[k][capacity]
```

Knapsack – Alternative Solution

It may happen that in some problem the total value of all items $V = \sum_{i=1}^{n} v_i$ is significantly smaller than the size of the knapsack C. When this happens, it is worthwhile to consider an alternative solution which runs in $O(n.V)$ and is therefore more efficient than the previous solution which runs in $O(n.C)$.

For a given subset of items S, let $0 \leq v(S) \leq V$ be their total value and $w(S)$ be their total weight. The goal is to calculate $w^*(k, v)$, which is defined as the minimum possible weight of using only the first k items that has a value that is at least v.

$$w^*(k, v) \doteq \min\{w(S) \,|\, S \subset \{1, \ldots, k\}, v(S) \geq v\}\,.$$

Knapsack – Alternative Solution

From the definition of w^* it follows that $w^*(k, v)$ is an increasing function of v: if we require items with a larger total value to be packed, the smallest size that can be achieved increases.

Since $w^*(n, \cdot)$ is increasing, the largest value of $0 \leq v \leq V$ such that $w^*(n, v) \leq C$ can be found by a form of binary search in $O(log(V))$ time.

Knapsack – Alternative Solution

Goal: Minimize the sum of weights for a subset of items with total value $\geq v$

Knapsack – Alternative Solution

Goal: Minimize the sum of weights for a subset of items with total value $\geq v$

Subproblem: $Knapsack2(i, v)$: the minimum total weight when packing a knapsack of total value $\geq v$ considering items $\{1, \ldots, i\}$
Number of subproblems: O(n.V)

Guess: Include the *i*th item in the knapsack or not

Knapsack – Alternative Solution

Recurrence Relation:

$$
Knapsack2(i,v) = \begin{cases} +\infty, & \text{if } i = 0, \\ \max(Knapsack2(i-1, \max(v-v_i, 0)) + w_i, & \text{otherwise.} \\ \qquad Knapsack2(i-1, v)), & \end{cases}
$$

Running Time Analysis: Running time per subproblem: $O(1)$
Total running time for computing $Knapsack2(i, v)$: $O(n.V)$

Original Problem: $Knapsack2(n, \cdot)$

After these values are calculated, the maximum v^* which can be packed in a knapsack of size C can be found with binary search in $O(log(V))$ time.
Hence, total running time of this algorithm would be $O(n.V)$.

Example: Making Change

Given a list of coin denominations and a desired amount of change, what is the fewest number of coins required to make this change?

Example: Making Change

Given a list of coin denominations and a desired amount of change, what is the fewest number of coins required to make this change?

Is there a greedy algorithm to solve this problem?

Example: Making Change

Given a list of coin denominations and a desired amount of change, what is the fewest number of coins required to make this change?

Is there a greedy algorithm to solve this problem?

In the Canadian system with denominations [5, 10, 25, 100, 200], we simply hand back the largest coin that does not exceed the value owed until the change is made. This does indeed find the fewest number of coins!

Example: Making Change

Given a list of coin denominations and a desired amount of change, what is the fewest number of coins required to make this change?

Is there a greedy algorithm to solve this problem?

In the Canadian system with denominations [5, 10, 25, 100, 200], we simply hand back the largest coin that does not exceed the value owed until the change is made. This does indeed find the fewest number of coins!

What if the denominations are more “exotic”? For example, when the denominations are [1, 8, 12] and we want to make 16 cents change

Making Change – Top-Down Approach

Input: the set of denominations $\mathcal{C} = \{i\}_{1,\cdots,N}$ each is worth v_i, and the desired amount of change V

Goal: Minimize the number of coins required so that $\sum_i v_i = V$

Making Change – Top-Down Approach

Input: the set of denominations $\mathcal{C} = \{i\}_{1,\cdots,N}$ each is worth v_i, and the desired amount of change V

Goal: Minimize the number of coins required so that $\sum_i v_i = V$

Subproblem: Change(s) the fewest number of coins required to make the amount of change s

Number of subproblems: $O(M)$ where $M \doteq \frac{V}{\min_i v_i}$

Making Change – Top-Down Approach

Guess: the value of the last coin used to make this change
Number of choices: #denominations: N

Making Change – Top-Down Approach

Guess: the value of the last coin used to make this change
Number of choices: #denominations: N

Recursive Relation:

$$Change(s) = \begin{cases} 0, & \text{if } \; s = 0 \\ +\infty, & \text{if } \; s < 0 \\ \min_{i \in C} \Big(Change(s - v_i) + 1 \Big), & \text{otherwise} \, . \end{cases}$$

Running Time Analysis: Time per subproblem: $O(N)$
Total running time: $O(M \cdot N)$

Original Problem: Change(V)

Making Change – Top-Down Approach

Guess: the value of the last coin used to make this change
Number of choices: #denominations: N

Recursive Relation:

$$Change(s) = \begin{cases} 0, & \text{if } s = 0 \\ +\infty, & \text{if } s < 0 \\ \min_{i \in C} \Big(Change(s - v_i) + 1 \Big), & \text{otherwise}\,. \end{cases}$$

Running Time Analysis: Time per subproblem: $O(N)$
Total running time: $O(M \cdot N)$

Original Problem: Change(V)

How to reconstruct the optimal solution?

Top-Down Approach

```python
def change(value, memo=None):
    if memo is None:
        memo = {}
    if value < 0:
        return inf
    if not value in memo:
        if value == 0:
            memo[value] = 0
        else:
            memo[value] = min([change(value - d[i], memo)
                               for i in range(len(d))]) + 1

    return memo[value]
```

Bottom-Up Approach

Think about the values you need to calculate and create an ordering of the subproblems that allows you to build the table of optimal values.

Bottom-Up Approach

Think about the values you need to calculate and create an ordering of the subproblems that allows you to build the table of optimal values.

```
def bottom_up_change(value):
    table = [0 for amt in range(value+1)]

    for i in range(1, value+1):
        table[i] = inf
        for j in range(len(d)):
            if d[j] <= i and 1 + table[i - d[j]] < table[i]:
                table[i] = 1 + table[i - d[j]]

    return table[value]
```

Examples: Optimal Parenthesization

Solve the problem defined in the worksheet!

Summary

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

The top-down approach may be simpler in that we do not have to worry about the order in which we iterate through the subproblems, but we quickly run into problems with recursion depth. The bottom-up approach avoids that altogether.