Estratégias Algorítmicas 2020/21

Week 5 – Dynamic Programming

· U C ·

Universidade de Coimbra

Outline

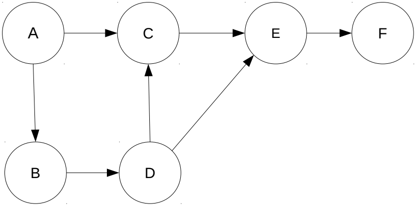
Introduction

Problem decomposition

- A problem may be decomposed in a sequence of nested sub-problems
- The original problem is solved by combining the solutions to the various sub-problems
- The choices made at the inner levels influence the choices to be made at the outer levels (in general)

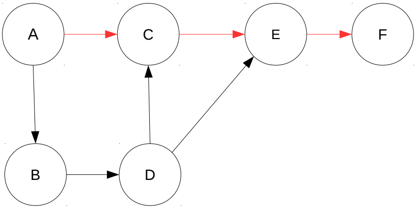
Shortest path in an acyclic directed graph

- Given an acyclic directed graph, find the shortest path between two vertices.



Shortest path in an acyclic directed graph

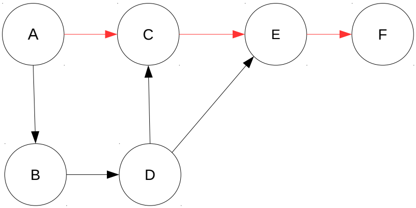
- Given an acyclic directed graph, find the shortest path between two vertices.



The shortest path from A to F is (A,C,E,F)

Shortest path in an acyclic directed graph

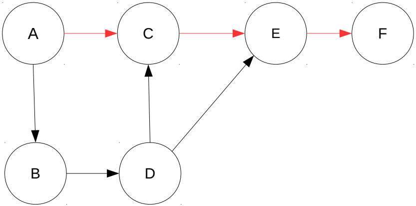
- Subproblem: Find the shortest path from source to a node v in the graph.



It has optimal substructure – the shortest path from source to target contains the shortest path for small subproblems.

Shortest path in an acyclic directed graph

- Subproblem: Find the shortest path from source to a node v in the graph.



Example: The shortest path from A to F is (A,C,E,F) $\implies$
The shortest path from A to E is (A,C,E).

Shortest path in an acyclic directed graph

Recursive algorithm to compute the value of the shortest path from node s to a target node t. The call should be $Path(t, 0)$

```
Function Path(v, len)
  if v = s then
    return len
  ℓ = ∞
  for each (i, v) ∈ A do
    ℓ = min{ℓ, Path(i, len + 1)}          {node i connects to node v}
  return ℓ
```

At each recursion, it computes the shortest path from node s to a given node v in the graph.

Shortest path in an acyclic directed graph

Top-down DP to compute the value of the shortest path from node s to a target node t. The call should be $Path(t, 0)$

Function $Path(v, len)$

  if $DP[v]$ is cached **then**
    return $DP[v]$
  if $v = s$ **then**
    return len
  $DP[v] = \infty$
  for each $(i, v) \in A$ **do**
    $DP[v] = \min\{DP[v], Path(i, len + 1)\}$ {node i connects to node v}
  return $DP[v]$

At each recursion, it computes the shortest path from node s to a given node v in the graph.

Shortest path in an acyclic directed graph

Bottom-up DP to compute the value of the shortest path from s to a target node t. The nodes are visited according to a topological ordering (more about this in some weeks). Assume that s is node 1 and t is node n.

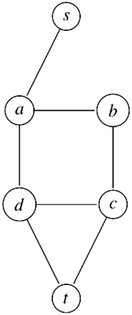
Function *Path*()

$DP[1] = 0$
for $v = 2$ **to** n **do**
  $DP[v] = \infty$
for $v = 2$ **to** n **do**
  for each $(v, i) \in A$ **do**
    $DP[i] = \min\{DP[i], DP[v] + 1)\}$ {node v connects to node i}
return $DP[n]$

Similar principle of Dijkstra Algorithm for more general graphs.

Longest simple path in an undirected graph

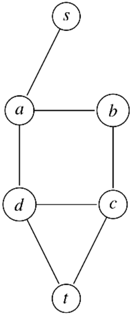
- Given an undirected graph, find the longest simple path between two vertices.



The longest simplest path from s to t is (s, a, b, c, d, t)

Longest simple path in an undirected graph

- Subproblem: Find the longest simple path from source to a node v in the graph.



It has no optimal substructure – the longest simple path from s to t does not contain the longest simple path from s to d, which is (s, a, b, c, t, d).

Problems for Dynamic Programming

- Coin changing: What is the minimum number of coins to make a change for C with n coin denominations? (assume infinite coins for each denomination)

- Subset sum problem: Is there a subset of coins that sums to C? (assume a finite set of n coins)

- Knapsack problem: I have n objects. Each object has a given weight and value. My knapsack can only carry W Kgs. Which objects should I pick that maximize the value and fit into the knapsack?

Coin Changing

What is minimum number of coins for a given change C?

Change 36 Euros with coin denominations 1, 5, 10, 20.

1. 36 - 20 = 16
2. 16 - 10 = 6
3. 6 - 5 = 1
4. 1 - 1 = 0

This is a greedy algorithm but it does not work with an arbitrary coin denominations.

Coin Changing

A counter-example for the greedy algorithm:

Change 30 Euros with coin denominations 1, 10, 25.

1. 30 - 25 $=$ 5
2. 5 - 1 $=$ 4
3. 4 - 1 $=$ 3
4. 3 - 1 $=$ 2
5. 2 - 1 $=$ 1
6. 1 - 1 $=$ 0

This totals 6 coins, but we could have used 3 coins of 10!

Coin Changing

Sub-problem

- Find the change for $C' \leq C$ with minimum number of coins using the first $i \leq n$ coin denominations.

Optimal substructure

1. If the optimal solution for the problem above contains a coin with denomination i, then by removing it, we have an optimal solution for the change without that coin. (We prove this in the following.)
2. If the optimal solution for the problem above does not contain a coin with denomination i, then we have an optimal solution for the same change for the first $i - 1$ denominations.

1. Let S be the set with the minimum number of coins to change C', taken from the first i denominations, and using a coin with denomination d_i.
2. Then, S without that coin is optimal for $C' - d_i$.

Sketch of the proof (by contradiction)

- (negate 2.) Assume that you can find a change for $C' - d_i$ with less coins using the first i denominations.
- (contradict 1.) Then, it is also possible to change C' with less coins by adding a coin with denomination d_i.

Coin Changing

Recursive approach

For denomination d_i:

1. Use denomination d_i and make the change for $C' - d_i$ with the denominations available (including d_i) or
2. Do not use denomination d_i, and make the change for C' with the remaining denominations.
3. Choose the minimum of the two.

Coin Changing

A first recursive solution:

Function $change(i, C)$

  if $C > 0$ and $i = 0$ **then** {1st base case - change $>$ 0 and}
    return ∞ {no more denominations}
  if $C = 0$ **then** {2nd base case - change is 0}
    return 0
  if $d_i > C$ **then**
    return $\underbrace{change(i-1, C)}_{\text{don't take denom. } d_i}$
  else
    return $\min(\underbrace{change(i-1, C)}_{\text{don't take denom. } d_i}, \underbrace{1 + change(i, C - d_i)}_{\text{take denom. } d_i})$

The number of recursive calls is exponential.

Can we do memoizing?

A top-down dynamic programming solution:

Function $change(i, C)$
  if $C > 0$ and $i = 0$ **then** {1st base case - change $>$ 0 and}
    return ∞ {no more denominations}
  if $C = 0$ **then** {2nd base case - change is 0}
    return 0
  if $T[i, C] > 0$ **then**
    return $T[i, C]$
  if $d_i > C$ **then**
    $T[i, C] = change(i - 1, C)$
  else
    $T[i, C] = \min(change(i - 1, C), 1 + change(i, C - d_i))$
  return $T[i, C]$

Table T stores the minimum number of coins for the first i denominations and each change C

Coin Changing

Example:

Change 12 Euros with coin denominations 1, 6, 10.

C	0	1	2	3	4	5	6	7	8	9	10	11	12
$T[0, C]$	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
$T[1, C]$	0	1	2	3	4	5	6	7	8	9	10	11	12
$T[2, C]$	0	1	2	3	4	5	1	2	3	4	5	6	2
$T[3, C]$	0	1	2	3	4	5	1	2	3	4	1	2	2

Can we do bottom-up DP? What are the base cases?

Can we order the computations?

Coin Changing

Bottom-up dynamic programming:

Function *change*(n, C)
  for $i = 0$ **to** n **do**
    $T[i, 0] = 0$
  for $j = 1$ **to** C **do**
    $T[0, j] = \infty$
  for $i = 1$ **to** n **do**
    for $j = 1$ **to** C **do**
      if $d_i > j$ **then**
        $T[i, j] = T[i-1, j]$
      else
        $T[i, j] = \min(T[i-1, j], 1 + T[i, j - d_i])$
  return $T[n, C]$

The time complexity is $O(nC)$, which is *pseudo-polynomial*.

Subset Sum

- Suppose you want to know if there exists a subset S of a set of n coins that makes the change for C (decision problem).
- This is know as the Subset Sum problem and it sounds similar to Coin Changing. Does it also have optimal substructure?

Subset Sum

Sub-problem

- Find whether it is possible to have a change for $C' \leq C$ using the first $i \leq n$ coins.
- Let S be a subset of coins, taken from the first i coins, that make change for C'.

Optimal substructure

- If S contains the i-th coin, then by removing it from S, we obtain a subset of coins for the change without that coin for the first $i - 1$ coins.
- If S does not contain the i-th coin, then it also makes the same change for the first $i - 1$ coins.

Why? (very trivial!)

Subset Sum

Recursion

- Choose the i-th coin:
  1. Either use it and solve sub-problem for $C - d_i$ with the remaining $i - 1$ coins, or
  2. Do not use it and solve sub-problem for C with the remaining $i - 1$ coins

Subset Sum

A simple recursive solution:

Function $subset(i, C)$
    if $i = 0$ and $C \neq 0$ **then** {1st base case - no more coins and}
        return false {change is not 0}
    if $C = 0$ **then** {2nd base case - change is 0}
        return true
    if $d_i > C$ **then**
        return $\underbrace{subset(i-1, C)}_{\text{don't take the } i\text{-th coin}}$
    else
        return $\underbrace{subset(i-1, C)}_{\text{don't take the } i\text{-th coin}} \vee \underbrace{subset(i-1, C-d_i)}_{\text{take the } i\text{-th coin}}$

It is an exponential approach. Can we do memoizing?

Subset Sum

A top-down dynamic programming:

Function *subset*(i, C)
  if $i = 0$ and $C \neq 0$ **then** {1st base case - no more coins and}
    return false {change is not 0}
  if $C = 0$ **then** {2nd base case - change is 0}
    return true
  if $T[i, C]$ is not empty **then**
    return $T[i, C]$
  if $d_i > C$ **then**
    $T[i, C] = subset(i - 1, C)$
  else
    $T[i, C] = subset(i - 1, C) \vee subset(i - 1, C - d_i)$
  return $T[i, C]$

Table T stores whether there is change or not with the first i coins.

Example:

Coins $= \{2, 6, 10\}$ and $C = 12$.

C	0	1	2	3	4	5	6	7	8	9	10	11	12
$T[0, C]$	T	F	F	F	F	F	F	F	F	F	F	F	F
$T[1, C]$	T	F	T	F	F	F	F	F	F	F	F	F	F
$T[2, C]$	T	F	T	F	F	F	T	F	T	F	F	F	F
$T[3, C]$	T	F	T	F	F	F	T	F	T	F	T	F	T

Can we do bottom-up DP? What are the base cases?

Can we order the computation?

Subset Sum

Bottom-up dynamic programming:

Function *subset*(n, C)

```
for i = 0 to n do                                  {1st base case}
    T[i, 0] = true
for j = 1 to C do                                  {2nd base case}
    T[0, j] = false
for i = 1 to n do
    for j = 1 to C do
        if d_i > j then
            T[i, j] = T[i − 1, j]
        else
            T[i, j] = T[i − 1, j] ∨ T[i − 1, j − d_i]
return T[n, C]
```

Also pseudo-polynomial since its time complexity is $O(nC)$.

Knapsack problem

- Knapsack problem: I have n objects. Each object i has a given weight w_i and value v_i. My knapsack can only carry W Kgs (capacity constraint). Which objects should I pick that maximize the value and fit into the knapsack?
- Does it also have optimal substructure?

Knapsack

Sub-problem

- Find the objects taken the first $i \leq n$ objects that maximize the value and satisfy the contraint $W' \leq W$.
- Let S be the optimal set of objects, taken from the first i objects, with total value v and total weight $w \leq W'$.

Optimal substructure

- If S contains the i-th object, then by removing it, we have an optimal solution with objects taken from the first $i-1$ objects that satisfies the constraint without the weight of that object. (we prove this in the following).
- If S does not contain the i-th object, then we have an optimal solution with objects taken from the first $i-1$ objects that satisfies constraint W'.

Knapsack

1. Let S be the optimal set of objects, taken from the first i objects, with total value v and total weight $w \leq W'$, and using the i-th object.

2. Then, S without that object, with total value $v - v_i$ and total weight $w - w_i$, is optimal for the first $i - 1$ objects and satisfies constraint $W' - w_i$.

Sketch of the proof (by contradiction)

- (negate 2.) Assume that there exists another set of objects, taken from the first $i - 1$ objects, with total value $v^* > v - v_i$ and total weight $w^* \leq W' - w_i$.

- (contradict 1.) Then, it also exists a set using the i-th object with total value $v^* + v_i > v$ and weight $w^* + w_i \leq W'$.

Knapsack

Recursive solution: Given the i-th object:

1. Either use it and solve sub-problem for $W - w_i$ with the remaining $i - 1$ objects, or
2. Do not use it and solve sub-problem for W with the remaining $i - 1$ objects
3. Choose the maximum value of the two.

Knapsack

A simple recursive solution:

Function $knapsack(i, W)$
  if $i = 0$ **then** {base case - no more objects}
    return 0
  if $w_i > W$ **then**
    return $\underbrace{knapsack(i-1, W)}_{\text{don't take the } i\text{-th object}}$
  else
    return $\max(\underbrace{knapsack(i-1, W)}_{\text{don't take the } i\text{-th object}}, \underbrace{v_i + knapsack(i-1, W - w_i)}_{\text{take the } i\text{-th object}})$

It is an exponential approach. Can we do memoizing?

Top-down dynamic programming:

Function *knapsack*(i, W)

  if $i = 0$ **then** {base case - no more objects}

    return 0

  if $T[i, W] \geq 0$ **then**

    return $T[i, W]$

  if $w_i > W$ **then**

    $T[i, W] = knapsack(i - 1, W)$

  else

    $T[i, W] = \max(knapsack(i - 1, W), v_i + knapsack(i - 1, W - w_i))$

  return $T[i, W]$

Table T stores the optimal value for the first i objects and constraint W.

Knapsack

Bottom-up Dynamic Programming:

Function *knapsack*(n, W)

for $j = 1$ **to** W **do** $\{$1st base case$\}$
  $T[0, j] = 0$
for $i = 0$ **to** n **do** $\{$2nd base case$\}$
  $T[i, 0] = 0$
for $i = 1$ **to** n **do**
  for $j = 1$ **to** W **do**
    if $w_i > j$ **then**
      $T[i, j] = T[i-1, j]$
    else
      $T[i, j] = \max(T[i-1, j], v_i + T[i-1, j-w_i])$
return $T[n, W]$

Also pseudo-polynomial since its time complexity is $O(nW)$.

Matrix-chain multiplication

- Given a sequence of matrices, find the most efficient way to multiply these matrices together.
- The number of operations depends of the order of multiplication.

Example: Let A_1 be a 20×40 matrix, A_2 be a 40×10 matrix and A_3 be a 10×70 matrix.

$(A_1 A_2) A_3$ has $(20 \times 40 \times 10) + (20 \times 10 \times 70) = 22000$ operations

$A_1 (A_2 A_3)$ has $(40 \times 10 \times 70) + (20 \times 40 \times 70) = 84000$ operations

The problem consists of finding the optimal parenthesisation!

Properties of matrix multiplication:

- Not commutative: $AB \neq BA$
- Associative: $(AB)C = A(BC)$

Matrix-chain multiplication

- Given n matrizes $A_1 A_2 \dots A_n$, where A_i has size $p_{i-1} p_i$, assume that the optimal parenthesisation is known. Then, it must split the product at some position k:

$$(A_1 \dots A_k)(A_{k+1} \dots A_n)$$

- Then, the parenthesisation of both $(A_1 \dots A_k)$ and $(A_{k+1} \dots A_n)$ must also be optimal: optimal substructure!

- Proof by contradiction: Assume that parenthesisation above of $(A_1 \dots, A_k)$ takes x operations, but it is still possible to find another with $x' < x$. Then, it is also possible to improve the optimal parenthesisation, which is a contradiction!

A recursive solution:

- Let $mult(i, j)$ be the minimum number of scalar multiplications needed to compute $A_i A_{i+1} \dots A_j$. (A_i has size $p_{i-1} \times p_i$)
- Base case: $mult(i, i) = 0$ for all $i = 1, 2, ..., n$
- Recursion: $mult(i, j) = mult(i, k) + mult(k + 1, j) + p_{i-1} p_k p_j$.
- As k is not known, must compute minimum over $i \leq k < j$.

A simple recursive solution:

Function $mult(i, j)$
  if $j \leq i$ **then** {base case}
    return 0
  $cost = \infty$
  for $k = i$ **to** $j - 1$ **do**
    $cost = min(cost, mult(i, k) + mult(k + 1, j) + p_{i-1} p_k p_j)$
  return $cost$

- This solution takes exponential time.

Matrix-chain multiplication

Top-down approach:

Function $mult(i, j)$
  if $j \leq i$ **then** {base case}
    return 0
  if $M[i, j] \geq 0$ **then**
    return $M[i, j]$
  $cost = \infty$
  for $k = i$ **to** $j - 1$ **do**
    $cost = \min(cost, mult(i, k) + mult(k + 1, j) + p_{i-1} p_k p_j)$
  $M[i, j] = cost$
  return $cost$

- With memoization, it takes $O(n^3)$ time.

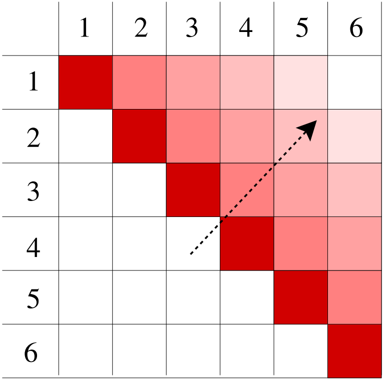
Matrix-chain multiplication

Bottom-up approach:

	1	2	3	4	5	6
1						
2						
3						
4						
5						
6						

Matrix-chain multiplication

Bottom-up approach:



Matrix-chain multiplication

Bottom-up approach:

Function $mc(n)$
  for $d = 2$ **to** n **do**
    for $i = 1$ **to** $n - d + 1$ **do**
      $j = i + d - 1$
      $M[i,j] = \infty$
      for $k = i$ **to** $j - 1$ **do**
        $M[i,j] = \min(M[i,j], M[i,k] + M[k+1,j] + p_{i-1}p_kp_j)$
  return $M[1][n]$

- Sweeps the array M diagonally
- It has $O(n^3)$ time complexity