DYNAMIC PROGRAMMING

Gustavo Carvalho
(ghpc@cin.ufpe.br)

Universidade Federal de Pernambuco
Centro de Informática, 50740-560, Brazil

Agenda

1 Dynamic programming

2 Approximation algorithms

3 Bibliography

Dynamic programming

Technique for solving problems with overlapping subproblems

Fibonacci numbers

- $F(n) = F(n-1) + F(n-2)$, for $n > 1$
- $F(0) = 0$ e $F(1) = 1$

Assuming that $n \geq 0$:

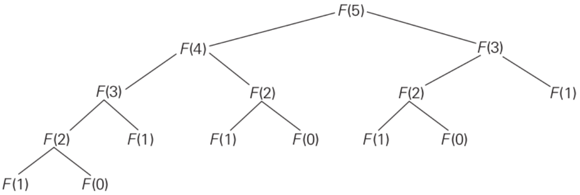
Algorithm: int Fib(n)

if $n \leq 1$ **then return** n;
else return $Fib(n-1) + Fib(n-2)$;

Centro de Informática UFPE

Universidade Federal de Pernambuco

Dynamic programming[1]



Better approach (assuming that $n \geq 0$):

Algorithm: int Fib2(n)

$F[0], F[1] \leftarrow 0, 1;$
for $i \leftarrow 2$ **to** n **do**
  $F[i] \leftarrow F[i-1] + F[i-2];$
return $F[n];$

[1] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Dynamic programming

This technique can be seen as a variation of space-for-time trade-off

In some situations, it is even possible to avoid the extra space

- Fibonacci: store only the last two elements

Approaches for implementing dynamic programming

- Bottom-up: solves all subproblems
- Top-down: solves all required subproblems

Example: coin-row problem

There is a row of n coins whose values are some positive integers $c_1, c_2, ..., c_n$ not necessarily distinct. Goal: pick up the maximum amount of money, but not picking up two adjacent coins

Recurrence relation:

- $F(n) = max\{c_n + F(n-2), F(n-1)\}$, for $n > 1$
- $F(0) = 0$ e $F(1) = c_1$

Algorithm: int CoinRow(C[1..n])

```
F[0], F[1] ← 0, C[1];
for i ← 2 to n do
    F[i] ← max(C[i] + F[i − 2], F[i − 1]);
return F[n];
```

Centro de Informática UFPE

Universidade Federal de Pernambuco

Example: coin-row problem[2]

Considering: 5, 1, 2, 10, 6, and 2

$F[0] = 0$, $F[1] = c_1 = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5					

$F[2] = \max\{1 + 0, 5\} = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5				

$F[3] = \max\{2 + 5, 5\} = 7$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7			

$F[4] = \max\{10 + 5, 7\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15		

$F[5] = \max\{6 + 7, 15\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	

$F[6] = \max\{2 + 15, 15\} = 17$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	**17**

[2] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Example: change-making problem

Give change for amount n using the minimum number of coins $d_1 < d_2 < \dots < d_m$ where $d_1 = 1$ (assuming unlimited coins)

Recurrence relation:

- $F(n) = min_{j:n \geq d_j}\{F(n - d_j)\} + 1$, para $n > 0$
- $F(0) = 0$

Algorithm: int ChangeMaking(D[1..m], n)

```
F[0] ← 0;
for i ← 1 to n do
    temp, j ← ∞, 1;
    while j ≤ m ∧ i ≥ D[j] do
        temp ← min(F[i − D[j]], temp);
        j ← j + 1;
    F[i] ← temp + 1;
return F[n];
```

Example: change-making problem[3]

Considering: $n = 6$ and the following denominations $1, 3$ e 4

$F[0] = 0$

n	0	1	2	3	4	5	6
F	0						

$F[1] = \min\{F[1-1]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1					

$F[2] = \min\{F[2-1]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2				

$F[3] = \min\{F[3-1], F[3-3]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1			

$F[4] = \min\{F[4-1], F[4-3], F[4-4]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1	1		

$F[5] = \min\{F[5-1], F[5-3], F[5-4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	

$F[6] = \min\{F[6-1], F[6-3], F[6-4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	**2**

Centro de Informática UFPE

Universidade Federal de Pernambuco

[3] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Example: knapsack (0-1)[4]

Given n items of weight w_i and value v_i, where $i = 1 \cdots n$, and a capacity W, find the most valuable subset of items that fits into the knapsack's capacity

Recurrence relation:

- $F(i, j) = max\{F(i-1, j), v_i + F(i-1, j-w_i)\}$, if $j - w_i \geq 0$
- $F(i, j) = F(i-1, j)$, if $j - w_i < 0$
- $F(0, j) = 0$, for $j \geq 0$, $F(i, 0) = 0$, for $i \geq 0$

		0	$j-w_i$	j	W
	0	0	0	0	0
	$i-1$	0	$F(i-1, j-w_i)$	$F(i-1, j)$	
w_i, v_i	i	0		$F(i, j)$	
	n	0			goal

[4] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Example: knapsack (0-1) – bottom-up[5]

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3,\ v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2,\ v_4 = 15$	4	0	10	15	25	30	**37**

Algorithm: int Knapsack(n,W,w[1..n],v[1..n],F[0..n,0..W])

for $i \leftarrow 0$ **to** n **do**
  for $j \leftarrow 0$ **to** W **do**
    if $i = 0 \lor j = 0$ **then** $F[i][j] \leftarrow 0$;
    else if $w[i] \leq j$ **then**
      $F[i][j] \leftarrow max(F[i-1][j], v[i] + F[i-1][j-w[i]])$;
    else $F[i][j] \leftarrow F[i-1][j]$;

return $F[n][W]$;

[5] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Example: knapsack (0-1) – top-down | 0,j/i,0=0, i,j=-1[6]

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	—	12	22	—	22
$w_3 = 3,\ v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2,\ v_4 = 15$	4	0	—	—	—	—	**37**

Algorithm: int MFKnapsack(i,j,w[1..n],v[1..n],F[0..n,0..W])

if $F[i,j] < 0$ **then**
  if $j < w[i]$ **then** $value \leftarrow MFKnapsack(i-1, j, w, v, F)$;
  else
    $value \leftarrow max(MFKnapsack(i-1, j, w, v, F),$
      $v[i] + MFKnapsack(i-1, j-w[i], w, v, F));$
  $F[i,j] \leftarrow value;$
return $F[i,j];$

[6] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Agenda

1 Dynamic programming

2 Approximation algorithms

3 Bibliography

Centro de Informática UFPE

Universidade Federal de Pernambuco

Approximation algorithms

Sometimes it is even harder to find a solution

- NP-Complete $\subset$ NP-hard

Alternative: “solve” the problem using approximation algorithms

- The solution does not need to be optimal, but only acceptable

Examples: greed approximation for TSP

- Nearest-neighbour algorithm
- MST-based algorithm

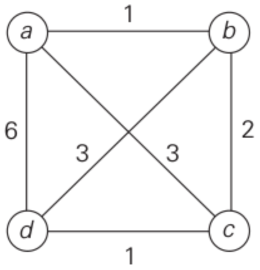
TSP: nearest-neighbour algorithm[7]

Initial node: a

- $s_a = a - b - c - d - a$,
  cost = 10

Optimal solution

- $s^* = a - b - d - c - a$,
  cost = 8
- Accuracy ration:
  $r(s_a) = \frac{10}{8} = 1,25$



Changing the weight of (a, d) to w:

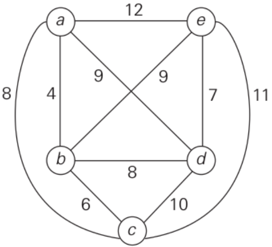
- $r(s_a) = \frac{4+w}{8}$

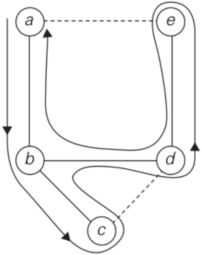
[7] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

TSP: MST-based algorithm[8]

In general terms:

1. Construct an MST
2. Perform a walk around the MST
   (recording the order of nodes visited)
3. Eliminate all repeated occurrences except the initial node
   (making shortcuts in a walk)





[8] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

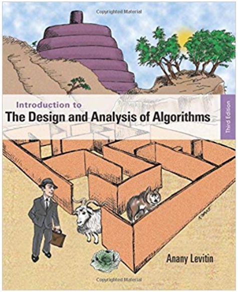
Agenda

1 Dynamic programming

2 Approximation algorithms

3 Bibliography

Bibliography



Chapter 9 (pp. 283–296)
Chapter 12 (pp. 441–457)
Anany Levitin.
Introduction to the Design and Analysis of Algorithms.
3rd edition. Pearson. 2011.

Chapter 16 (pp. 509–513)
Clifford Shaffer.
Data Structures and Algorithm Analysis. Dover, 2013.

Centro de Informática UFPE
Universidade Federal de Pernambuco

DYNAMIC PROGRAMMING

Gustavo Carvalho
(ghpc@cin.ufpe.br)

Universidade Federal de Pernambuco
Centro de Informática, 50740-560, Brazil