Chapter 1 : Dynamic programming

ENSIIE - Operations Research Module

Dimitri Watel (dimitri.watel@ensiie.fr)

2022

Principle

Definition

Dynamic programming is a method for solving problems, it is not an algorithm but a way to build an algorithm.

Warning

The important part of the course is the method, **not the examples.** The problems on which we will apply the dynamic programming method in the course or in the tutorials are not the same as the ones you will have to solve during the exams.

In this course there will be 3 examples:

- The Fibonacci sequence
- The subset sum problem
- The shortest path problem (in a special case)

Find a recursive function

First step

Dynamic programming is a way to program a recurrence relation. In order to solve a problem with the dynamic programming method, the first step consists in finding that relation.

Fibonacci sequence

Problem definition

Given an integer n, compute $f(n)$ defined by
$f(n) = f(n-1) + f(n-2)$ if $n > 1$ and by $f(0) = 0$ and $f(1) = 1$.

The recurrence relation is given by the problem itself.

Fibonacci sequence : naïve version

function $f(i)$
    if $i \leq 1$ **then**
        return i
    return $f(i-1) + f(i-2)$

Subset sum problem

$$P = \{1, 1, 1, 5, 7, 8, 8, 9, 17\}$$

Is there a subset of P of size 40?

$$\exists ? P' \subset P | \sum_{p \in P'} p = 40$$

Subset sum problem

$$P = \{\mathbf{1}, \mathbf{1}, \mathbf{1}, \mathbf{5}, \mathbf{7}, \mathbf{8}, 8, 9, \mathbf{17}\}$$

Is there a subset of P of size 40?

$$\exists ? P' \subset P | \sum_{p \in P'} p = 40$$

Recurrence relation

$$P = \{1, 1, 1, 5, 7, 8, 8, 9, 17\}$$

Let $P = \{p_1, p_2, \dots, p_9\}$ and $B = 40$. The problem becomes:

Subset sum

$\exists ? P' \subset \{p_1, p_2, \dots, p_9\} | \sum_{p \in P'} p = B.$

More general version

Let $i \leq 9$ and $b \leq B$, compute
$f(i, b) = \exists ? P' \subset \{p_1, p_2, \dots, p_i\} | \sum_{p \in P'} p = b.$

Recurrence relation

If we assume $\exists P' \subset \{p_1, p_2, \ldots, p_i\} | \sum_{p \in P'} p = b$. In that case:

- either $p_i \in P'$ and then $\exists P" \subset \{p_1, p_2, \ldots, p_{i-1}\} | \sum_{p \in P"} p = b - p_i$
- or $p_i \notin P'$ and then $\exists P" \subset \{p_1, p_2, \ldots, p_{i-1}\} | \sum_{p \in P"} p = b$

$$f(i, b) = f(i-1, b-p_i) \vee f(i-1, b)$$

Recurrence relation

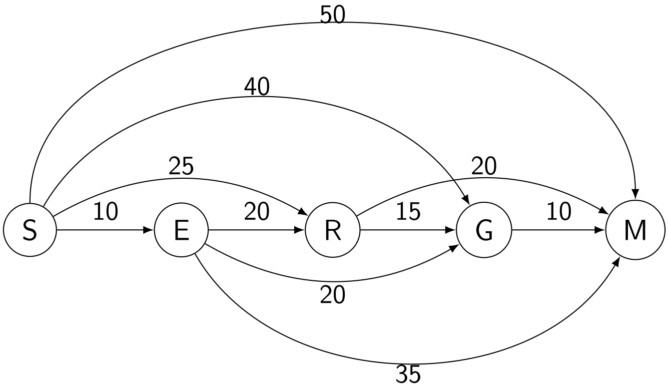
To be more exact:

$$
\begin{aligned}
f(i, b \geq 0) &= f(i-1, b-p_i) \vee f(i-1, b) \\
f(i, b < 0) &= \perp \\
f(0, b \neq 0) &= \perp \\
f(0, 0) &= \top
\end{aligned}
$$

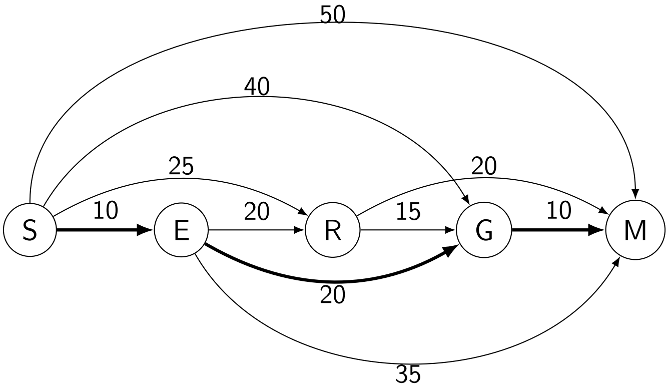
Subset sum : naïve version

function $f(i, b)$
  if $b < 0$ **then**
    return $\perp$
  if $i = 0$ **then**
    return $(b = 0)$
  return $f(i-1, b-p_i) \vee f(i-1, b)$

What is the shortest path from S to M?



What is the shortest path from S to M?



Recurrence relation

Let $d(v)$ be the weight of a shortest path from v to M. We want $d(S)$. We write $\omega(u, v)$ the weight of the arc (u, v).

More general version

Let v be a node of G, compute $d(v)$.

Recurrent relation

Let u^* be the successor of v in a shortest path P from v to M, then $d(v) = d(u^*) + \omega(v, u^*)$. If u is another successor of v, then $d(u) + \omega(v, u) \geq d(u^*) + \omega(v, u^*)$.

$$d(v) = \min_{u \in \Gamma^+(v)} (d(u) + \omega(v, u))$$
$$d(M) = 0$$

Remark : it works because the graph has no circuit!

Shortest path : naive version

function $d(v)$
    if $v = M$ **then**
        return 0
    return $\min_{u \in \Gamma^+(v)} (d(u) + \omega(v, u))$

Intractability

None of those recurrence relations should be computed naïvely: some calculations are done multiple times.

Fibonacci

$f(n) = f(n-1) + \mathbf{f(n - 2)} = \mathbf{f(n - 2)} + \mathbf{f(n - 3)} + \mathbf{f(n - 3)} + f(n-4) = \dots$

Subsetsum

Si $p_{i-1} = p_i = 1$, $f(i, B) = f(i-1, B-1) \vee f(i-1, B) =$
$f(i-2, B-2) \vee \mathbf{f(i - 2, B - 1)} \vee \mathbf{f(i - 2, B - 1)} \vee f(i-2, B)$

Shortest Path

If the arcs (u, v), (u, w) and (v, w) exist.
$d(u) = \min(d(v) + \omega(u, v), \mathbf{d(w)} + \omega(u, w), \dots) =$
$\min(\min(\mathbf{d(w)} + \omega(v, w), \dots) + \omega(u, v), \min(\dots) + \omega(u, w), \dots)$

Dynamique programming : memoization version

Definition

The memoization dynamic programming technique consists in storing the results of each recursive call so that, when the call is done a second time, the computation is not done twice.

Fibonacci sequence : Memoization

Let $T : \mathbb{N} \to \mathbb{N}$ be an array where every cell is initially empty.

function $f(i)$
  if $i \leq 1$ **then**
    return i
  return $f(i-1) + f(i-2)$

function $f_{Memo}(i)$
  if $T(i)$ is empty **then**
    if $i \leq 1$ **then**
      return i
    $T(i) \leftarrow f_{Memo}(i-1) + f_{Memo}(i-2)$
  return $T(i)$

Subset sum

Let $T : \mathbb{N}^2 \rightarrow \{\top, \bot\}$ be an array where every cell is initially empty.

function $f(i, b)$
  if $b < 0$ **then**
    return $\bot$
  if $i = 0$ **then**
    return $(b = 0)$
  return $f(i-1, b-p_i) \vee f(i-1, b)$

function $f_{Memo}(i, b)$
  if $T(i, b)$ is empty **then**
    if $b < 0$ **then**
      return $\bot$
    if $i = 0$ **then**
      return $(b = 0)$
    $T(i, b) \leftarrow f_{Memo}(i-1, b-p_i) \vee f_{Memo}(i-1, b)$
  return $T(i, b)$

Shortest path

Let $T : V \to \mathbb{N}$ be an array where every cell is initially empty.

function $d(v)$
  if $v = M$ **then**
    return 0
  return
$\min_{u \in \Gamma^+(v)} (d(u) + \omega(v, u))$

function $d_{Memo}(v)$
  if $T(v)$ is empty **then**
    if $v = M$ **then**
      return 0
    $T(v) \leftarrow$
$\min_{u \in \Gamma^+(v)} (d_{Memo}(u) + \omega(v, u))$
  return $T(v)$

Generalization

Dynamic programming : memoization version

In order to compute a recursive function $f : X \to Y$,

- create an array $T : X \to Y$
- Before doing any calculation, check if the result is not already in T
- Store every result in T before returning it.

Dynamic programming : iterative version

Definition

The iterative dynamic programming technique consists in solving first the terminal subproblems, and then in solving every subproblems by going back through the recursive calls until the problem is solved. Every intermediate result is stored.

Fibonacci sequence : Iterative version

- List of all subproblems : Compute $f(i), \forall i \leq n$
- Recursive calls : $f(i) \rightarrow f(i-1), f(i-2)$
- Terminal cases : $f(0), f(1)$

$\Rightarrow$ Compute $f(i-1)$ before $f(i)$

$\Rightarrow$ Compute $f(i)$ for i from 0 to n

Fibonacci sequence : Iterative version

Let $T : [\![0, n]\!] \rightarrow \mathbb{N}$ be an array where every cell is initially empty.

function $f(i)$
    if $i \leq 1$ **then**
        return i
    return
$f(i-1) + f(i-2)$

function $f_{Iter}(n)$
    $T(0) \leftarrow 0$
    $T(1) \leftarrow 1$
    for i from 2 to n **do**
        $T(i) \leftarrow T(i-1) + T(i-2)$
    return $T(n)$

Subset sum : Iterative version

- List of all subproblems : Compute $f(i, b), \forall i \leq n, b \leq B$
- Recursive calls : $f(i, b) \rightarrow f(i-1, b-p_i), f(i-1, b)$
- Terminal cases : $f(0, b), \forall b$

$\Rightarrow$ Compute $f(i-1, b), \forall b \leq B,$ before $f(i, b), \forall b \leq B$

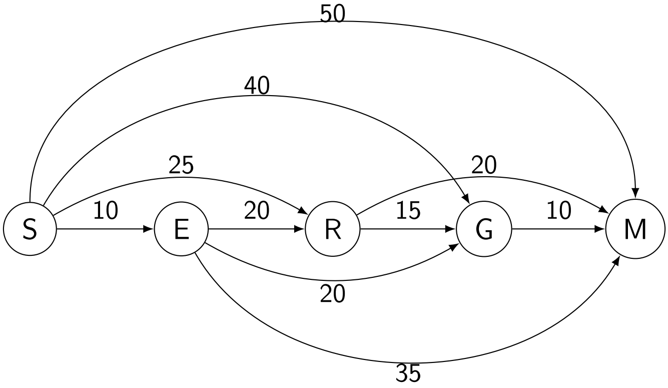
$\Rightarrow$ Compute $f(i, b)$ for i from 0 to n, for b from 0 to B

Subset sum : Iterative version

Let $T : [\![0; n]\!] \times [\![0; B]\!] \rightarrow \{\top, \bot\}$ be an array where every cell is initially empty.

function $f(i, b)$
    if $b < 0$ **then**
        return $\bot$
    if $i = 0$ **then**
        return $(b = 0)$
    return
$f(i-1, b-p_i) \vee f(i-1, b)$

function $f_{Iter}(n, B)$
    for b from 0 to B **do**
        $T(0, b) \leftarrow (b = 0)$
    for i from 1 to n **do**
        for b from 0 to B **do**
            if $b \geq p_i$ **then**
                $T(i, b) \leftarrow T(i-1, b-p_i) \vee T(i-1, b)$
            else
                $T(i, b) \leftarrow T(i-1, b)$
    return $T(n, B)$


50
40
25
20
10
20
15
10
S
E
R
G
M
20
35

Shortest path : Iterative version

- List of all subproblems

$$\text{Compute } d(v), \forall v$$

- Recursive calls

$$d(v) \rightarrow d(u), \forall u \in \Gamma^+(v)$$

- Terminal cases

$$d(M)$$

$$\Rightarrow \text{Compute } d(u), \forall u \in \Gamma^+(v), \text{ before } d(v)$$

$$\Rightarrow \text{Compute } d(u) \text{ in reversed topological order.}$$

Shortest path : Iterative version

Let $T : V \to \mathbb{N}$ be an array where every cell is initially empty.

function $d(v)$
 if $v = M$ **then**
  return 0
 return $\min_{u \in \Gamma^+(v)} (d(u) + \omega(v, u))$

function $d_{Iter}(S)$
 $T(M) = 0$
 $L \leftarrow$ Reversed topological ordering of $G \backslash M$
 for $v \in L$ **do**
  $T(v) \leftarrow \min_{u \in \Gamma^+(v)} (T(u) + \omega(v, u))$
 return $T(S)$

Generalization

Dynamic programming : iterative version

In order to compute a recursive function $f : X \to Y$,

- list all the subproblems of $f(x)$
- list all the recursive calls of f
- list all the terminal cases of f
- Compute the terminal cases and store the results in an array T
- Go back through the recursive calls until the initial problem is solve and store every intermediate result in T

Complexity : Fibonacci Memoization

1: **function** $f_{Memo}(i)$
2:     **if** $T(i)$ is empty **then**
3:         **if** $i \leq 1$ **then**
4:             **return** i
5:         $T(i) \leftarrow f_{Memo}(i-1) + f_{Memo}(i-2)$
6:     **return** $T(i)$

- Lines 3 to 5 are done at most once per $i \leq n$
- $\Rightarrow$ At most $2n$ recursive calls to f_{Memo}, twice per $i \leq n$.
- $\Rightarrow$ At most n calls with lines 3 to 5 in time $O(1)$, and at most n calls without, in time $O(1)$ too.

$\Rightarrow$ Complexity : $O(n+n) = O(n)$

Complexity : Subset sum

```
1: function f_Memo(i, b)
2:     if T(i, b) is empty then
3:         if b < 0 then
4:             return ⊥
5:         if i = 0 then
6:             return (b = 0)
7:         T(i, b) ← f_Memo(i − 1, b − p_i) ∨ f_Memo(i − 1, b)
8:     return T(i, b)
```

- Lines 3 to 7 are done at most once per $i \leq n, b \leq B$
- $\Rightarrow$ At most $2nB$ recursive calls to f_{Memo}, twice per $i \leq n, b \leq B$.
- $\Rightarrow$ At most nB calls with lines 3 to 7 in time $O(1)$, and at most nB calls without, in time $O(1)$ too.

$\Rightarrow$ Complexity : $O(nB + nB) = O(nB)$

Complexity : Subset sum

On rappelle que $G = (V, E)$, on note $n = |V|$ et $m = |E|$.

```
1: function d_Memo(v)
2:     if T(v) is empty then
3:         if v = M then
4:             return 0
5:         T(v) ← min_{u ∈ Γ+(v)} (d_Memo(u) + ω(v, u))
6:     return T(v)
```

- Lines 3 to 5 are done at most once per node $v \in V$
- $\Rightarrow$ At most $m = \sum_{v \in V} deg(v)$ recursive calls to d_{Memo}, $deg(v)$ per $v \in V$.
- $\Rightarrow$ At most n calls with lines 3 to 5 in time $O(deg(v))$, and at most $m - n$ calls without, in time $O(1)$ too.

$\Rightarrow$ Complexity : $O(\sum_{v \in V} deg(v) + m - n) = O(2m - n) = O(m)$

Iterative version

The two versions (usually) have the same complexity: on the worst case, the array T is fully filled.

Warning : special cases may occurs.

Sometimes it does not work.

Recursive combinatorial explosion

We want to transform a binary number into another using transformation rules. Each rule transform a number into a **bigger** number.

For example :

- $0 \xrightarrow{(1)} 10$
- $1 \xrightarrow{(2)} 01$
- $101 \xrightarrow{(3)} 01001$
- $010 \xrightarrow{(4)} 0111$

We can now transform numbers

$\underline{0} \xrightarrow{(1)} \underline{1}0 \xrightarrow{(2)} \underline{010} \xrightarrow{(4)} 01\underline{1}1 \xrightarrow{(2)} 0\underline{101}1 \xrightarrow{(3)} \dots$

Question : we are given two binary numbers x and y and a set $\mathcal{T}$ of m transformation rules, it is possible to transform x into y with the rules of $\mathcal{T}$?

Recursive combinatorial explosion

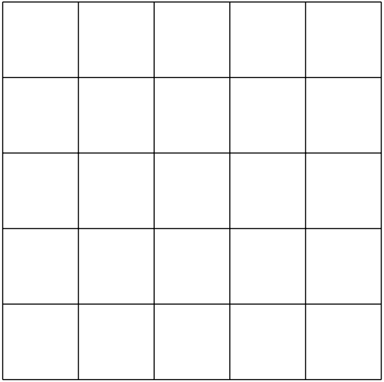
There is a dynamic programming algorithm to solve the problem.

function $f(x, y)$
  if $|x| > |y|$ **then**
    return FALSE
  else if x = y **then**
    return TRUE
  if $T(x, y)$ is empty **then**
    $T(x, y) \leftarrow$ FALSE
    for all rule $t \in \mathcal{T}$ **do**
      for all way to apply t to x **do**
        $T(x, y) \leftarrow T(x, y) \vee f(t(x), y)$
  return $T(x, y)$

Recursive combinatorial explosion

But we need to store too many information ($2^{|y|-|x|}$ cells in T).

Wrong recursion



Latin square problems

Input

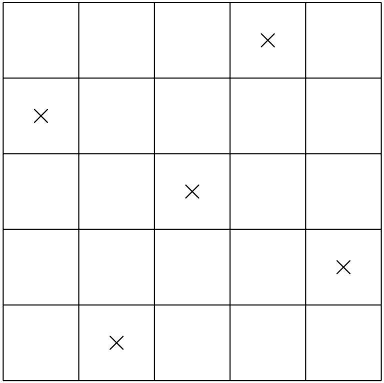
An empty grid with $n \times n$ cells

Output

n cells such that

- exactly one cell per line and column
- **maximize** the minimum distance between two chosen cells.

Wrong recursion



Latin square problems

Input

An empty grid with $n \times n$ cells

Output

n cells such that

- exactly one cell per line and column
- **maximize** the minimum distance between two chosen cells.

Wrong recursion

No obvious way to generate a recursive function.

In the syllabus, to be seen in tutorials

- How to find the solution in addition to the value of the solution? (for instance, how to find a shortest path in addition to the cost of the shortest path?; how to find the subset instead of only proving it exists, . . .)
- The Roy-Floyd-Warshall algorithm : find all the pair of shortest paths in a graph.