Dynamic programming

Recherche opérationnelle
Dimitri Watel - ENSIIE

2024

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

1 Recurrence relation

Dynamic programming is a way to compute the result of a recursive algorithm. In order to solve a problem with the dynamic programming method, the first step consists in finding a recurrence relation that can be solved with a recursive algorithm.

In the rest of the course, we consider the three following example.

The Fibonacci sequence

Ensure: The i-th element of the Fibonacci sequence starting with 0 and 1.

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

The subset sum problem

Require: Two integers $i \in [\![0; n]\!]$ and $b \in [\![0; B]\!]$; and n non-negative integers $p_1, p_2, \ldots, p_n$

Ensure: Check if there exists some subset I of $[\![1; i]\!]$ such that $\sum_{j \in I} p_j = b$

function $g(i, b)$
  if $i \leq 0$ **then**
    return $(b = 0)$
  if $b - p_i \geq 0$ **then**
    return $g(i-1, b-p_i) \vee g(i-1, b)$
  else
    return $g(i-1, b)$

The shortest path in a DAG

Require: A DAG (directed acyclic graph) G with weights ω on the arcs, two nodes v and t of G

Ensure: The shortest path from v to t

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

2 Memoization

The memoization is one of the two dynamic programming techniques and 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.

Memoization is a mechanical process that can be easily executed by a computer (for example, by a compiler in a way that is transparent to the user). Below, we revisit the three previous examples using the method.

In each algorithm, we consider we have access to an empty and global association table T that accept any type of key and value.

The Fibonacci sequence

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

The subset sum problem

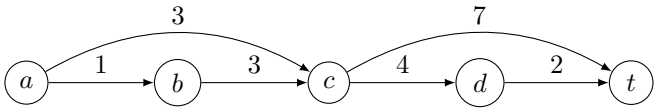
function $g_{Memo}(i, b)$
  if $T(i, b)$ is empty **then**
    if $i = 0$ **then**
      $T(i, b) \leftarrow (b = 0)$
    else if $b - p_i \geq 0$ **then**
      $T(i, b) \leftarrow g(i-1, b-p_i) \vee g(i-1, b)$
    else
      $T(i, b) \leftarrow g(i-1, b)$
  return $T(i, b)$

The shortest path in a DAG

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

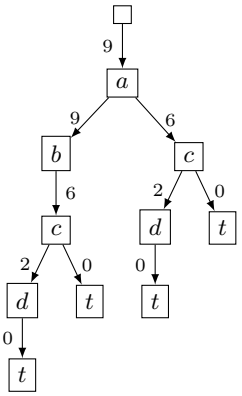
2.1 Execution example and comparison with the naive version

Let's show an execution example on the shortest path. We consider the following graph G in which we are looking for the shortest path from a to t.



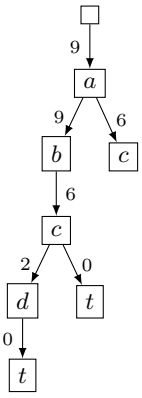
In the figures below, each tree represents an execution and each node is a recursive call. The value of the node v given as input is indicated on the node. Above a node, on its incoming edge, the returned value of the recursive call is shown

argIf we apply the naive algorithm, we will have the following execution.



We observe a small flaw in the naive algorithm; it calls the node c multiple times and executes exactly the same recursive calls twice. In an arbitrary graph, this can occur on many nodes and lead to combinatorial explosion.

If we apply the memoization method, we will have the following execution instead. When the second call to c is made, we will have already stored the result in the table T. Therefore, it is not necessary to redo the entire calculation.



2.2 Complexity calculation

We consider as elementary operations: arithmetic operations, boolean operations, conditions, the start of a loop iteration, assignments, returning a value, and calling a function.

Theorem 2.1. *Let N be the number of possible entries for the recursive calls and E be the maximum number of elementary operations within a recursive call. Then the complexity of the associated memoization algorithm is bounded by $O(NE)$.*

Example 1. We consider the subset sum example. We have two integers as input, i and b. The integer i is between 0 and n, and b is between 0 and B. Therefore, $N = (n+1)(B+1) = O(nB)$.

The maximum number of elementary operations performed by a recursive call is constant (there is no loop, the number of operations does not depend on i and b). Therefore, $E = O(1)$

Thus, the complexity of the algorithm is, according to this theorem, $O(nB)$."

Proof. When the function is called with an input for the first time, it can perform at most E elementary operations during that call. Since there are N possible inputs, the total number of elementary operations performed by the function calls when the input is seen for the first time is $O(NE)$.

During each new call to the function, the result is stored in T. Therefore, any call made with an already seen input will only execute the following operations: check if T contains the input and return the value in T. There will be no new recursive call. These calls are then executed in $O(1)$.

Thus, the total number of elementary operations performed is $O(NE)$ plus the number of recursive calls made with an already seen input. We still need to determine how many function calls there are in total to compute the complexity.

Recursive calls can only occur when the function is called with an input for the first time. Since calling the function is an elementary operation, there cannot be more than E new recursive calls. Therefore, the function cannot be called more than $O(NE)$ times.

Thus, the complexity is $O(NE + NE) = O(NE)$. □

This theorem only indicates an upper bound; we can be a bit more precise if we have information about the algorithm.

Memoization is easy to implement, but its complexity and actual computation time are difficult to assess. Additionally, it depends on the efficiency of the language in handling recursive functions.

3 Iterative method

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.

It is often considered that this version is THE method of dynamic programming. But note that memoization does the same thing in the opposite direction. It is sometimes said that memoization is a Top-Down method, while iterative dynamic programming is a Bottom-up method.

Like in the memoization part, we assume we have access to a global and empty associative table T that accepts any type of key and value. We want to store in T the results of all the necessary recursive calls for the computation. To do so, we need to reverse the recursive calls: in T, we only store the result of a call if the results it needs to compute its own result have already been stored. In other words, in the tree representing the execution of the computation, we need to go back up the nodes from the leaves to the root. We will see this through the three previous examples.

The Fibonacci sequence We seek to calculate the n-th element of the sequence. Each recursive call is $f(i)$ with i from 0 to n. We note that during the execution of $f(i)$, the recursive calls decrement the value of i. If we calculate $f(i-1)$ before $f(i)$, we get the results in reverse order. The base cases are $i = 0$ and $i = 1$. Therefore, it is sufficient to compute $f(i)$ for i varying from 0 to n.

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 $T(n)$

The subset sum problem We are trying to determine whether there exists a subset I of $[\![1; n]\!]$ such that $\sum_{j \in I} p_j \leq B$. Each recursive call is $f(i, b)$ with i from 0 to n and b ranging from 0 to B. We note that during the execution of $f(i, b)$, the recursive calls decrement the value of i. If we compute $f(i-1, b)$ for all b before $f(i, b)$ for all b, we get the results in reverse order. The base cases are $i = 0$ for all b. Thus, it is sufficient to compute, for i varying from 0 to n, the value of $f(i, b)$ for b varying from 0 to 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 $T(n, B)$

The shortest path in a DAG We want to find the shortest path from s to t in a DAG G. Each recursive call is $d(v)$ with v being a node in G. We note that, during the execution of $d(v)$, the recursive calls only invoke the successors of v. If we compute $d(w)$ for every successor w of v before $d(v)$, we get the results in reverse order. The terminal case is $d(t)$. Therefore, it is sufficient to calculate $d(v)$ for every node v by making the calls in a reversed topological ordering of the graph (such an order is a list of nodes where the predecessors of a node w appear after w, and such an order always exists in a DAG).

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 $T(S)$

In conclusion of this section, there isn't really a clear method for implementing the iterative method. Unlike memoization, which can be automated, one must have a deep understanding of the problem. This is obvious in the last example where knowledge of graph theory is necessary to produce the algorithm.

An advantage of the iterative method is that it is (generally) easier to deduce complexity by examining the algorithms produced. Furthermore, the algorithms are not recursive, so the associated programming language doesn't need to handle recursive calls efficiently.

A drawback compared to the memoization method is that the iterative approach often generates algorithms that compute (and store) many more results than necessary, unlike memoization. For example, in the subset sum problem, the iterative algorithm fills T with $B + 1$ values for each integer i. However, by looking at the recursive algorithm, we can see that memoization will not store more than one value when $i = n$, two values when $i = n - 1$, four when $i = n - 2$, and so on. This issue can sometimes be partially mitigated by noting that it is not necessary to store the entire table T, but this does not reduce the time complexity of the algorithm.

Here is, for example, a variant that works very well for the subset sum problem:

```
function f_Iter(n, B)
    for b from 0 to B do
        T(0, b) ← (b = 0)
    for i from 1 to n do
        for b from 0 to B do
            Retirer la clef (i − 2, b) de T si elle existe.
            if b ≥ p_i then
                T(i, b) ← T(i − 1, b − p_i) ∨ T(i − 1, b)
            else
                T(i, b) ← T(i − 1, b)
    return T(n, B)
```

It can be seen that, indeed, $T(i-2,b)$ is never used, so it is not necessary for the calculation.

Remark 1. It can be observed that the 3 naive algorithms presented at the beginning of the course are exponential, while the 6 dynamic programming versions are polynomial (or almost, but I will not go into details here). This is of course not a general rule; there exist problems for which there are exponential naive recursive algorithms that dynamic programming fails to improve sufficiently to make them polynomial. One might think, for example, of the traveling salesman problem.