STUDIA UNIV. BABEŞ-BOLYAI, INFORMATICA, Volume XLIII, Number 1, 1998

# THE DYNAMIC PROGRAMMING METHOD: A NEW APPROACH

HORIA GEORGESCU AND CLARA IONESCU

**Abstract.** In this article, the authors propose a new approach to the Dynamic Programming method.

The classical description of the method is rather vague (including notions as "state" and "decision", that have to be defined for each problem) and is based mainly on presenting some (very few) examples.

Firstly, a so-called DP-problem is presented, as a general framework for the problems for which the method can be applied. Secondly, the programmer has to identify a so-called DP-tree in the graph defined by the DP-problem. Thirdly, a postorder traversal of this DP-tree has to be done in order to get the value attached to the root, which is the value requested by each concrete problem.

A parallel analysis of the two approaches is performed. The authors claim that this approach is more precise, more general and easier to understand and use.

## 1. Introduction

Richard Bellman [1] proposed in 1957 the Dynamic Programming Method as a technique to solve sequential decision problems. This method is classically described in the following way:

1) the "states" of the problem are determined obviously, the significance of a state being different from problem to problem;

2) in order to solve the problem, successive decisions had to be taken in order to pass from one state to another. A cost, representing the "effort" to reach the final state (corresponding to the problem considered) from the initial state (the cost of which is known from the beginning), is associated to each state;

3) the aim is to take those decisions (to run through that itinerary of successive states) which start from the initial state and allow to reach the final state, namely by associating to this last one the possible optimum value (minimum or maximum). The meaning is that no such other route of states will reach the final state with a "better" value attached to it;

We can notice that, from the beginning, it is pointed out that the Dynamic Programming Method is considered to be applicable only to optimization problems.

Richard Bellman suggests that the optimum decision string should be searched for on the basis of the *optimality principle*, which has the following enunciation: "An optimal strategy has the following property: whatever the initial state and the initial

---

*1991 Mathematics Subject Classification.* 68T20, 68Q25.

decision, the strategy obtained from the initial one by removing this initial decision is optimal if applied to the state resulting from the first decision" (see [1]).

The above formulation of the method was copied almost identically in all subsequent approaches of this method; the slight generalizations did not succeed in throwing light upon the mysterious "states" and "decisions" that "are to be chosen depending on the concrete problem".

## 2. The *DP*-problem

*Let* $A$ *and* $B$ *be two sets of elements. For each element* $x\in A$ *a significance, that depends on the concrete problem to solve, is given. The significance is materialized through a value* $v_x\in B$; *initially, only the values* $v_x$ *of the elements* $x$ *from a given subset* $X\subset A$ are known.

*For each element* $x\in A$ *both a subset* $A_x\subset A$ *and a function* $f_x$ *are known; the function* $f_x$ *depends on the values attached to the elements from* $A_x$ *and on the input data. The value attached to the element* $x$ *can be computed (using the function* $f_x$*) only if the values attached to every element from* $A_x$ *are known.*

*The subset* $X$ *is* $X=\{x\in A \mid A_x=\emptyset\}$, *i.e. the set of elements* $x$ *from* $A$ *for which the value* $v_x$ *is given from the beginning (does not depend on the values associated with other elements). For each such an element the function* $f_x$ *has no arguments, so that it is a certain constant* $v_x$. *We will presume* $X\neq\emptyset$.

*An element* $z\in A$ *is given too.*

*The task is to compute, if possible, the value* $v_z$.

***Example 1.*** Let be $A=\{1,2,\ldots,12\}$, $B=N$, $X=\{1,2,6,7,8,9\}$ and $A_3=\{1,2\}$, $A_4=\{1,2,3\}$; $A_5=\{1,4\}$, $A_{10}=\{7,8\}$, $A_{11}=\{8,9\}$, $A_{12}=\{10,11\}$. The value 1 is attached to every vertex from $X$. For all the other vertices $x\in A\setminus X$, the attached function is the sum of its arguments. Therefore, the value attached to every vertex $x$ will be the sum of the values that are attached to the elements from $A_x$. Finally, let $z=5$.

It can be easily seen that the values that will be attached to the 12 vertices are: $(1,1,2,4,5,1,1,1,1,2,2,4)$. The solution of the problem is $v_z=v_5=5$.

***Example 2.*** Let be $A=\{1,2,\ldots,8\}$, $B=N$, $X=\{1,2,6,7\}$ and $A_3=\{1,2,4\}$, $A_4=\{2,5\}$, $A_5=\{3\}$, $A_8=\{6,7\}$. The value 1 is attached to every vertex from $X$. For every other vertex $x\in A\setminus X$, the attached function is the sum of its arguments, i.e. the sum of the values that are attached to the elements from $A_x$. Finally, let $z=5$.

It can be noticed that only the value attached to the element 8 can be computed; consequently, the problem has no solution.

The above mentioned problem will be called *the DP-problem*, for reasons that will later be obvious. Its description is very general, as it results from the following analysis.

The set $A$ is an arbitrary set; particularly it can be either finite or infinite.

The set $B$ is arbitrary, too. The most common situations are the following:

- $B$ is $N$, $I$ or $R$;
- $B=\{0,1\}$; in this case, it is a decision problem (0 is associated with false, and 1 is associated with true);

- $B$ is a cartesian product $B=B_1 \times B_2 \times ... \times B_k$; in this case, a value from $B$ is actually a $k$-uple $(b_1, ..., b_k)$ of values, not necessarily of the same type.

The functions $f_x$ that are attached to the vertices $x \in A$ are arbitrary, too. The dependency of arguments can be expressed, for example, through an arithmetical expression, a minimum or a maximum.

The second example shows that the problem does not always have solutions. This also happens for example when $z \notin X$ and when for any $x \in A \setminus X$ we have $A_x \setminus X \neq \emptyset$ (meaning that $x$ „depends" on elements that do not belong to $X$).

We will consider that an element $x \in A$ is *accessible* (starting from the subset $X$) if the value that is associated with this element can be computed; obviously, all the elements from $X$ are accessible. It can be easily noticed that the following statement is valid: "*The DP-problem has a solution if the element $z$ is accessible*".

It is also important that the values associated with the elements $x \in A$ should be *correct*, which means that they should be the intended ones (they must be equal to $v(x)$). Therefore, we will work under the following hypotheses:

1. *the value initially associated with the elements $x \in X$ are the values $v(x)$, so they are correct;*
2. *it is known that the value associated with every $x \in A$ element can be computed only if the values associated with the elements from $A_x$ are known. Let be $A_x=\{a_1, ..., a_k\}$. We will always presume that the condition: $f_x(v(a_1), ..., v(a_k))=v(x)$ is fulfilled; this condition says that if the values associated with $a_1, ..., a_k$ are the intended ones (they are correct), then the value associated with $x$ is correct, too (and it is the intended one).*

The above hypotheses will be used under the name of *correctness hypotheses* (*assumptions*).

A very important result is the following one:

**Proposition.** *In the presence of the correctness hypotheses, for every element whose value can be computed, this value is correct.*

This result is intuitive because:

1. we start from correct values;
2. correct values are used every time a value is computed;
3. the functions $f_x$ applied to correct values lead to correct values.

We will give a rigorous proof of this result after some remarks.

A special case of the *DP*-problem is the one in which the functions $f_x$ are Boolean functions. In this case, it is not always necessary to know *all* the arguments of a function in order to get its value. For example, in the case of a disjunction, when we know that the value of a term is equal to $1$, the value of the function will be equal to $1$, irrespective of the values of the other terms. As a consequence, the values of the arguments that do not occur in the first term are of no importance and therefore they are not to be necessarily computed.

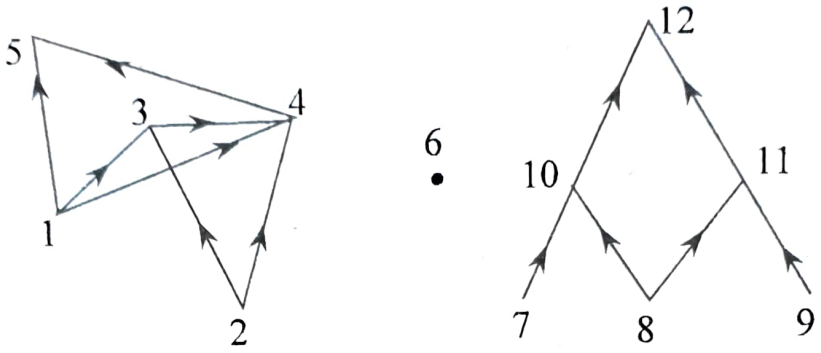
The *DP*-problem can be represented in a natural way on an oriented graph. The graph is called *dependency graph* and is constructed as follows:

1. every vertex corresponds to an element from $A$, and the correspondence is one to one;
2. for every $x \in A$, the edges that arrive in $x$ are exactly the ones that have the elements from $A_x$ as initial extremity.

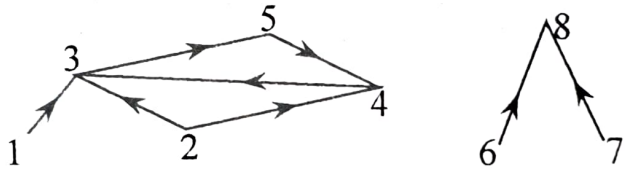
In a dependency graph, the *sons of a vertex* $x$ are considered as being the initial extremities of the edges that converge in $x$.

For the examples considered above, the dependency graphs are:

***Example 1:***




***Example 2:***




We can now rephrase our problem in graph terms:

*Let $G=(A,B,U,(f_x)_{x\in A},X)$ be a dependency graph, where $U$ is the set of edges. The aim is to attach values to the vertices from the set $B$. Initially, only the values of the elements of the set $X$ of the initial vertices (in which no edge does converge) are given. The value attached to a vertex $x$ is computed by applying the function $f_x$ to the values attached to the sons of $x$ (obviously, if these values are known). It is required to compute the value attached to a given vertex $z$.*

Let $G$ be a dependency graph. For every vertex $x$, let $O_x$ be the set of the *noticeable vertices*, i.e. those vertices for which there is a path that starts from them and reaches $x$. Moreover, we will say that *vertex* $x$ *depends on vertex* $y$ if the value of $x$ depends on the value of $y$, i.e. if $y \in O_x$.

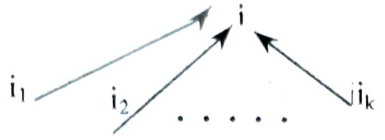
***Remarks:***

1. In the dependency graph, (the value which is attached to) $z$ does not depend in any way on the values of the vertices from $A \setminus O_z$ .
2. *The problem has a solution only if the directed subgraph associated to $O_z$ is acyclic.*

*Proof of the proposition.*

Let us consider a dependency graph. We can attach to it a universal algebra $U=(A,\Sigma,a)$ as follows:

- $A$ is the set of the vertices of the graph;
- for each dependence of the form:




we define a function $f_{i_1,\dots,i_k;i}$ in the following way:

$$\begin{cases} f_{i_1,\dots,i_k;i}(i_1,\dots,i_k) = i \\ f_{i_1,\dots,i_k;i}(\bullet,\dots,\bullet) \uparrow \text{ otherwise} \end{cases}$$

Let X be the set of the initial vertices. Let X′ be the subalgebra generated by X; then X′ represents the set of vertices whose values can be computed. Let p be a property on A. The principle of the algebraic induction implies that: *If* p *is true for the set* X*, then it is true for every vertex in* X′.

In the next sections we will introduce four ways to solve the *DP*- problem, namely:

- the method of the ascending sequence of sets;
- topological sorting;
- the traversal of the dependency graph;
- the Dynamic Programming method

For the first three ones we follow [3], [4], [5].

## 3. Methods to solve the DP-problem

### 3.1. The method of the ascending sequence of sets

This method can be applied to the *DP*-problem if the set A is finite. The successive determination of accessible points is tracked by forming up an ascending sequence of sets, as follows:

$$\begin{cases} X_0 = X \\ X_{n+1} = X_n \cup \left\{ x \in A \mid A_x \subset X_n \right\}, \forall n \geq 0 \end{cases}$$

The process of the successive computation of the sets from the ascending sequence $\{X_n\}$ can be stopped when the first n with $X_n=X_{n+1}$ is reached (no element is added to $X_n$). For this n, $X_n$ is the set of accesible elements.

On the other hand, it is natural to stop if we have reached the conclusion that z is accessible, because every time we will get a new accessible element, we will compute its attached value, too.

Thus, we obtain the following algorithm, where the function `calculus` which computes (using the $f_x$ functions) the value $v_x$ attached to the vertex x, is used:

```
U←X          { U is the current set of the accessible vertices }
V←A\X        { V is the complementary set of U }

repeat
    W←V
    for all x∈V
        if Ax⊂U
        then U←U∪{x}
```

```
|   |   |   V←V\{x}
|   |   |   v_x←calculus(x)
|   |   |   if x=z
|   |   |   then write(v_z); stop
|   |   |
|   |
└ until W=V
write(z," is not accessible")
```

We show how this method works for the examples considered in the previous section:

***Example 1:***

We have successively:

$X_0=\{1,2,6,7,8,9\}$,

$X_1=X_0\cup\{3,4,10,11\}=\{1,2,3,4,6,7,8,9,10,11\}$,

$X_2=X_1\cup\{5,12\}=\{1,2,3,4,5,6,\ldots,12\}$ with $v_5=5$.

***Example 2:***

We have successively: $X_0=\{1,2,6,7\}$, $X_1=X_0\cup\{8\}=\{1,2,6,7,8\}$, $X_2=X_1$ and consequently the following message will be printed '5 is not accessible'.

The method of the ascending sequence of sets is however not suitable for the *DP*-problem, since it is too much time-consuming, from the following reasons:

1. each time the **repeat** cycle is resumed, all the elements from V=A\U are checked, in order to see if they can be added to U;
2. the order in which the checking of the elements from V is done, is not specified; hopefully, a good order should take into account the chances these elements have to be included in the set U.

## 3.2. Topological sorting

In terms of theory of graphs, the problem of topological sorting can be described as follows:

*Let* G *be a directed graph. Distinctive labels* $label_1, label_2, \ldots, label_n \in \{1,2,\ldots,n\}$ *have to be attached to the vertices* $1,2,\ldots,n$, *so that for every edge* (i,j) *read from the input, the inequality* $label_i<label_j$ *holds.*

It can be easily noticed that:

1. The problem does not always have a solution; more precisely, there is a solution only if the graph is acyclic.
2. If a solution exists, then it is not necessarily unique. For example, if the graph does not have any edge, then any distinctive labeling represents a solution.
3. Let us consider an arbitrary algorithm that can solve the above problem. Then, if we substitute in it the operations by which the vertices x get values (labels) through the calculus of the $v_x$ value, according to the $f_x$ function, then we obtain a correct algorithm for computing the values of the vertices of a dependency graph.

We can always assume that $A=\{1,2,\ldots,n\}$. Let $m$ be the number of edges of the graph.

The algorithm consists in identifying successively an element $i$ that has no predecessors, in computing its attached value and in removing it from the graph (together with its adjacent edges). For this purpose we can use the following data structures:

1. for each $i\in\{1,2,\ldots,n\}$, $S_i$ is the list of the successors of $i$ in the current graph (the initial graph will successively „decrease" by removing vertices); these lists are initially empty;
2. the vector $nrpred$ of length $n$ stores in each $nrpred_i$ the number of predecessors of $i$ in the current graph (initially $nrpred_i=0$) ;
3. a queue $C$ that contains the vertices with no predecessors in the current graph.

In the initialization phase, the $m$ edges are read. For each such edge $(i,j)$, $j$ is added to the list $S_i$, while $npred_j$ is increased by one. Then, the queue $C$ is initialized in an obvious way. The time required for these operations is $O(m+n)$.

Afterwards, a vertex $i$ with no predecessors in the current graph is successively extracted from the queue, its value $v_i$ is computed and then $i$ is removed from the graph, together with the edges that diverge from it. In this way, new vertices without predecessors may appear; these vertices will be introduced in the queue:

```
while C≠∅
    i⇐C; v_i←calculus(i)
    if i=z then write(v_z); stop
    for all j∈S_i
        nrpred_j←nrpred_j-1
        if nrpred_j=0 then j⇒C
write('Vertex z is not accessible')
```

In the above algorithm the function `calculus` computes, using the function $f_i$, the value corresponding to its argument (vertex) $i$.

Obviously, the algorithm successively computes the values attached to accessible vertices. The algorithm stops either if the value of $v_z$ is obtained, or if a circuit has been detected in the subgraph defined by the vertices noticeable from the vertex $z$.

Each time the **while** cycle is executed, the number of computations is linear in $|S_i|$, so that the total number of computations is linear in $|S_1\cup S_2\ldots\cup S_n|=m$. Therefore the overall time complexity of the algorithm is $O(m+n)$, i.e. the algorithm is linear.

For *Example 2* from the previous section, the vertices 1,2,6,7,8 will be introduced and then extracted from the queue; consequently the following message will be displayed: `'Vertex z is not accessible'`.

Unfortunately, applying the topological sorting algorithm to dependency graphs is inadequate: the values of some vertices that are „uninteresting" to the proposed purpose

are also computed; these values belong to some vertices unnoticeable from z. Consequently, the execution time may increase dramatically.

### 3.3. The traversal of the dependency graph

The first idea is to apply the *Divide and Conquer* strategy: for each vertex x, the vertices from $A_x$ are traversed; this traversal includes the calculus of their values and enables us to compute the value $v_x$:

```
procedure DivImp(x)
  for all y∈Ax\X
    DivImp(y)
  vx←calculus(x)
end
```

where the function `calculus` has the same meaning as in the previous sections.

In order to obtain the value attached to `z`, the call `DivImp(z)` can be used.

This approach has the advantage that only the „strictly necessary" vertices used for obtaining $v_z$ are taken into account, i.e. the noticeable vertices (those ones from which a path to the root exists in the dependency graph).

On the other hand, the above algorithm has two major deficiencies:

- it is only applicable to acyclic graphs; in the case of a circuit consisting of „strictly necessary" vertices, the algorithm does not stop;
- it is possible to compute the value of the same vertex more than once; this may seriously increase the execution time.

Our purpose is to remove the two deficiencies presented above. For this purpose, we will follow two stages:

1. the vertices noticeable from `z` and the subgraph associated to them must be identified;
2. for the subgraph obtained in this way, the topological sorting algorithm presented in the last section can be applied.

In order to identify the noticeable vertices, a `DF` traversal of the dependency graph can be used. For this purpose it suffices to call the procedure `DF` with the argument `z`. The following recursive procedure uses the list `L`, initially empty, containing noticeable vertices:

```
procedure DF(x)
  x⇒L
  for all y∈Ax
    if y∉L then DF(y)
end
```

It is known that the execution time needed for a DF-traversal is `O(m+n)`. So, the same is valid for the entire algorithm.

In this way, the deficiencies of the previous algorithm have been removed. Yet, it is clear, it would be preferable that:

1. the set of the noticeable vertices should be known from the beginning;
2. the dependency graph should have a form that would allow an easier traversal of its vertices, ment to establish the associated values.

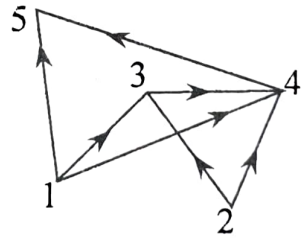
## 4. Describing the method of Dynamic Programming

### 4.1. The description of the method

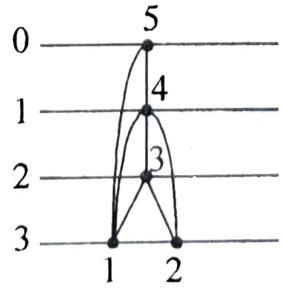
**Definition.** A *DP-tree with root* $z$ is an acyclic oriented graph with all vertices noticeable from $z$. Obviously, if a dependency graph is a *DP-tree*, then the *DP*-problem has actually a solution, which means that the value $v_z$ can be computed.

***Examples:***

- any tree in which the orientation of each of its edges is „towards the root" is a *DP*-tree;
- in the *Example 1* considered in the first section, the subgraph containing the noticeable vertices from vertex 5 is a *DP*-tree.




A *DP*-tree can be "arranged on levels" in the following way: each vertex is arranged on the level whose number is equal to the maximum length of the paths linking the vertex to the root. For the first example, we obtain:




For a *DP*-tree, a postorder numbering of the vertices can be performed, similarly as for the vertices of a tree. After the postorder numbering, the vertices are topologically sorted: if there is a path from the vertex $j$ to the vertex $i$, then $nrpostord_j < nrpostord_i$.

We can now describe the method of Dynamic Programming:

*The method of Dynamic Programming can be applied to the problems that aim to compute a value. The following steps are to be performed:*

1. *A dependency graph is associated to the DP-problem. In this graph a DP-tree with the same root is selected so that the order (corresponding to the postorder numbering) of the vertices is known and so that the initial problem is equivalent to compute the value associated to the root of the DP-tree.*
2. *The correctness hypotheses hold.*
3. *The vertices of the graphs are traversed in postorder, so that at the end the value of the root is obtained.*

A general form of the traversal algorithm is the following:

```
for all the vertices i
    if i∈X then reached_i←false
           else reached_i←true

postord(z)

procedure postord(i)
    for all j∈A_i and not reached_i
        postord(j)
    v_i←calculus(i); reached_i←true
end
```

The time complexity is $O(m+n)$, so that the algorithm is linear.

A proper knowledge of the *DP*-tree allows traversing its vertices in the descending order of their level number, without using recursivity; as a consequence the `reached` vector will be eliminated.

There are many cases when the *DP*-tree has some regularity, such as:

1. each vertex that is not a leaf has the same number $k$ of vertices on which they depend directly (meaning that $|A_x|=k$ for all the vertices $x$ that are not leaves);
2. there is a natural number $k$, so that for any edge $(i,j)$ the difference between the level numbers attached to $i$ and $j$ is at most $k$.

In such cases we can avoid keeping the values of *all* vertices. We will present just one very simple example, which additionally points out, that apparently surprising, the selection of the set $B$ is meaningful.

***The Fibonacci sequence***

*The terms of* ***Fibonacci sequence*** *are computed using the following rules:* $F_0=0$, $F_1=1$; $F_n=F_{n-1}+F_{n-2}$, $\forall n\geq 2$.

*For a given* $n\geq 2$, *the value* $F_n$ *has to be produced.*

A first obvious approach consists in the following choices:

$A=\{0,1,\ldots,n\}$; $X=\{0,1\}$; $B=N$

$A_0=A_1=\emptyset$; $A_k=\{k-1,k-2\}$, $\forall k\geq 2$

$v_k=F_k$, $\forall k\geq 2$; $f_0=0$; $f_1=1$; $f_k(a,b)=a+b$, $\forall k\geq 2$.

the *DP*-tree being exactly the dependency graph described above:

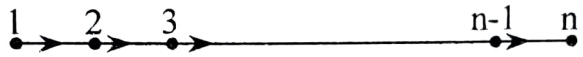

In a second approach, we will choose $B=N\times N$:

$A=\{1,2,\ldots,n\}$; $X=\{1\}$; $B=N\times N$

$A_1=\emptyset$; $A_k=\{k-1\}$, $\forall k\geq 2$

$v_k=(F_{k-1},F_k)$, $\forall k\geq 2$

$f_1=(0,1)$; $f_k(a,b)=(b,a+b)$, $\forall k\geq 2$




The advantage of this procedure is obvious: the dependency graph is a tree and so is the *DP*-tree; moreover, the tree is linear: the value associated to a vertex depends only on the value associated to the previous vertex.

It is also important not to implement automatically the above algorithm because in its general form it has the disadvantage that it implies storing the values attached to *all* *DP*-tree's vertices, which is not always necessary. For example, for computing the n-th term of Fibonacci sequence the algorithm becomes, according to the second approach, that well–known one:

```
read(n)
a←0; b←1
for i=2,n
   (a,b)←(b,a+b)
write(b)
```

## 4.2. Computing the sum of n numbers

Even if it is trivial, the problem of computing the sum of n numbers $a_1,\ldots,a_n$ will point out many interesting aspects of applying the Dynamic Programming method.

How do we choose the vertices? What significance do we give to the values that are attached to the vertices? Which are the dependencies between vertices? Finally, how do we choose the *DP*-tree that must be traversed in postorder in order to obtain the required value?

In his book „Three on two bicycles", *Jerome K. Jerome* said „A German is completely unable to understand basic things. But if you complicate them enough he will understand them at once!"

We will have to complicate things too, because it is obvious that we can not compute "from one stroke" the required sum and so it is necessary to compute some partial sums on the way. The questions are which partial sums are to be computed?

The most general choice consists in considering all the $2^n$ possible partial sums formed with $a_1,\dots,a_n$. The vertices have to correspond to these sums, so each vertex will be marked $[i_1,\dots,i_k]$ with $i_1<\dots<i_k$, and the attached value will be $v_{[i_1,i_2,\dots,i_k]} = a_{i_1} + a_{i_2} +\dots+ a_{i_k}$.

But are there not too many vertices? Yes, there are and this is due to the fact that the number of the vertices is exponential in the number $n$ of input data, which can possibly lead to considering a *DP*-tree with an exponential number of vertices, and thus to an exponential algorithm. That is why we will choose as vertices in the graph only the ones of the form $[i,i+1,\dots,j]$ with $1\le i\le j\le n$; for sake of simplicity, they will be marked $i..j$. So $A=\{i..j \mid 1\le i\le j\le n\}$ and $X=\{i..i \mid i=1,\dots,n\}$:

Obviously, the problem is to compute the value associated with the vertex $1..n$.

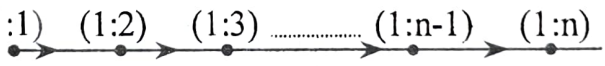
Many dependencies between the values attached to the vertices can be imagined; some of them are described below.

1. Let us consider the following dependency system:

$A_{i..i}=\emptyset$; $A_{i..j}=\{i..j-1\}$ for $1\le i<j\le n$ and $v_{i..i}=a_i$;

$f_{i..j}(i..j-1)=v_{i..j-1}+a_j$ for $1\le i<j\le n$.

Then a tree that gives the solution (by postorder traversal) is the following:

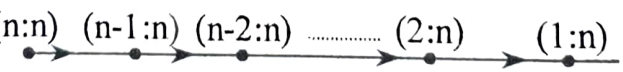



corresponding to the associativity to the left of the addition:

$a_1+a_2+\dots+a_n=(\dots((a_1+a_2)+a_3)+\dots)$.

2. Another dependency system we can consider is the following:

$A_{i..i}=\emptyset$; $A_{i..j}=\{i+1..j\}$ for $1\le i<j\le n$ and $v_{i..i}=a_i$ $f_{i..j}(i+1..j)= =a_i+v_{i+1..j}$ for $1\le i<j\le n$. A tree whose postorder traversal produces the solution is:




and corresponds to the associativity to the right of the addition:

$a_1+a_2+\dots+a_n=(\dots(a_{n-2}+(a_{n-1}+a_n))\dots)$.

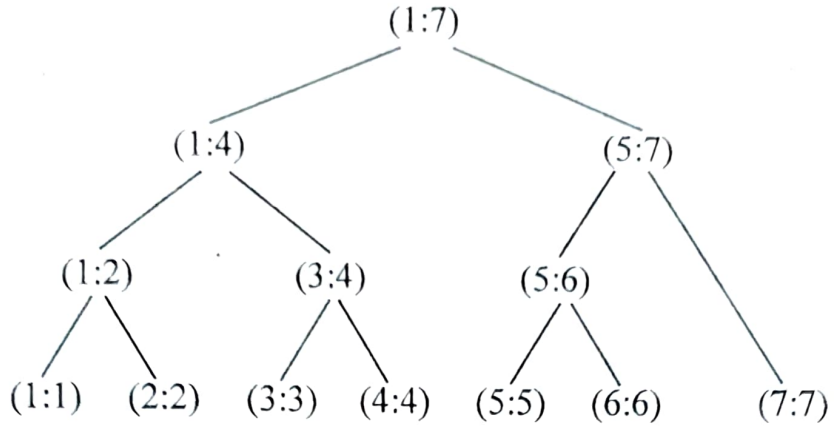
3. Let us consider now another dependency system:

$A_{i..i}=\emptyset$; $A_{i..j}=\{i..k, k+1..j\}$ for $1\le i<j\le n$ and $k=\lfloor (i+j)/2 \rfloor$,

$v_{i..i}=a_i$, $f_{i..j}(i..k, k+1..j)=v_{i..k}+v_{k+1..j}$, for $1\le i<j\le n$ and $k=\lfloor (i+j)/2 \rfloor$

which determines a *DP*-tree with the root $1..n$.

For example, for $n=7$ the DP-tree is the following:




The algorithm is the following one:

```
k←1
while k<n
  k2←k+k; i←1
  while i+k≤n
    a_i←a_i+a_{i+k}; i←i+k2
  k←k2
```

the result being obtained in $a_1$.

The above algorithm seems to be rather complicated, but it is important because it allows to write easily a parallel version, the time complexity of which is `O(log n)`.

This last discussion on this subject allows us to analyze more attentively the Dynamic Programming method as it has been presented in the previous section.

Gilles Brassard and Paul Bratley [2] emphasize the following characteristics of the Dynamic Programming method:

1. The multiple computation of the same value is avoided;
2. The process is a "bottom–up" one, going from simple subtasks towards more complex subtasks until the entire task is solved;
3. The principle of optimality is checked.

The first characteristic is obviously used in the previous example.

The second characteristic shows the difference between the Dynamic Programming method and the *Divide and Conquer* method. The last one starts "top-down": it successively decomposes the problems into (independent) subproblems of smaller size until directly solvable subproblems are reached; afterwards the method proceeds "bottom–up" by combining the solutions of the subproblems. For the last example, the *Divide and Conquer* method consisting in the function call `sum(1,n)`, where `sum` is the following recursive function:

```
function sum(i,j)
    if i=j then sum←a_i
           else k←⌊(i+j)/2⌋,
                sum←sum(i,k)+sum(k+1,j)
end
```

There is only a slight difference from the previous solution, but the reader will certainly notice it. Moreover, the „ambition" of going only bottom-up requires a more detailed analysis and possibly a less obvious, but sometime less time-consuming, algorithm.

The third alternative (the principle of optimality) is discussed in the following section.

For all three solutions the number of additions is $n-1$.

### 4.3. The link to the classic approach

*But what is the link with the way Richard Bellman presented the Dynamic Programming? Which is the optimization problem? Which are the states? Where is the principle of optimality?*

Let be the vector $a=(a_1,\dots,a_n)$. In the previous section we have presented many ways of computing the sum of the elements of a vector. Now we want to compute the value $minim=\min_{i=1,\dots,n} a_i$.

We will make the following changes in the algorithms from the previous section:
the significance of $v_{i..j}$ is now : $v_{i..j}=\min\{a_i,\dots,a_j\}$;
we will replaced the „+" sign with the operation of computing the minimum.

It is obvious that the algorithms obtained in this way will be correct, because the operation of computing the minimum has also the associativity property that was the basis of the computation of the sum.

Now, the problem became an optimization one. Actually, this was obvious because the equality often signifies an optimum; for example the question "*How many students are there in the classroom?*" is equivalent to "*Which is the maximum number of students present in the classroom?*"

Further, we will try to explain how the concepts of state of a problem and of decision do appear in our presentation.

Through *state of the problem* we shall understand the current set of vertices, whose value has been computed.

A *decision* consists in passing from an arbitrary $s$ to a state $s\cup\{x\}$, where $x\notin S$; in other words, a decision means to compute the value $v(x)$ associated to a vertex $x\notin S$. Obviously, the decision is possible only if $A_x\subset S$.

If we suppose that the DP-tree attached to the problem is identified and the correctness assumptions are verified, the application of the Dynamic Programming Method consists in taking a sequence of decisions (in applying a strategy), through which one can pass from the initial state to the final state.

It is still to be explained where the optimality principle "is hiding". Let us consider a strategy through which we pass from the initial state to the final state. It is obvious that it meets the optimality principle. Let us, indeed, consider the first decision in the strategy; let $x$ be the vertex whose value has been computed according to this decision. It is obvious that $x\notin X$ and $A_x\subset X$. According to the correctness assumptions, the value $v\{x\}$ is correct. It is clear now that the sequence of decisions that follows after the initial one is an optimal strategy for the problem in which the initial state is $X'=X\cup\{x\}$.

In the first section of this article it was made clear that the correctness assumptions must be fulfilled. One of these is the following: *for any vertex x, if the values attached to its sons are correct, then the value attached to x shall be correct, too.*

The optimality principle represents in fact the reciprocity of this statement: *if the value* $v_x$ *attached to a vertex* x (*computed using the function* $f_x$) *is correct, then it was obtained from correct values attached to its sons.*

In our opinion, both implications should be fulfilled, a fact which is actually implicitly performed in our approach.

Indeed, the optimality principle does not assure explicitly that the value attached to z is correct. In our approach, if the correctness assumptions hold, the correctness of the value v(z) is assured.

On the other hand, the approach proposed in this article suggests that the functions attached to the vertices should be computations of sums, minimum or maximum values, logical operations (disjunction or conjunction) etc. It is important to make clear that any kind of statement about the correctness of a program, of the values of some variables, etc., starts actually with "for any values of the input data". Considering this specification and the shape of the functions attached to the vertices, it follows that the optimality principle is verified.

However, it has still to be reminded that the optimality principle is used as well as a criterion to choose the Dynamic Programming Method for solving problems. In our approach, this criterion is replaced by checking if the problem to be solved is a *DP*-problem.

Finally, let us make some remarks about the set B. Usually, this set is chosen not only as the set of the values associated to the vertices. Its meaning has to include also the following aspects:

1. to allow us to reformulate the problem to be solved as a DP-problem;
2. to allow us to identify more simple structures of dependencies;
3. to include for each vertex that information that will allow us to obtain later the genesis of the value v(z). For example, when computing the minimum value of the elements of an array, B has to be chosen so that the position in the array of the current minimum value should be kept too.

## REFERENCES

[1] **Bellman R.E., Dreyfus S.E.** – *Applied Dynamic Programming*, Princeton University Press, Princeton, N.J, 1962.

[2] **Brassard G., Bratley P.** – *Algorithmics: Theory and Practice*, Prentice-Hall, London, 1988.

[3] **Cormen Thomas H., Leiserson Charles E., Rivest Ronald L.** – *Introduction to Algorithms*, MIT Press, Massachusetts Institute of Technology, 1990.

[4] **Horowitz E., Sahni S.** – *Fundamentals of Computer Algorithms*, Computer Science Press, New York, 1978.

[5] **Livovschi L., Georgescu H.** – *Sinteza şi analiza algoritmilor*, Editura Ştiinţifică, Bucureşti, 1986.

UNIVERSITY OF BUCHAREST, FACULTY OF MATHEMATICS, BUCHAREST, ROMANIA
*E-mail address:* hg@oroles.cs.unibuc.ro

"BABES-BOLYAI" UNIVERSITY, FACULTY OF MATHEMATICS AND COMPUTER SCIENCE, RO-3400 CLUJ-NAPOCA, ROMANIA
*E-mail address:* clara@cs.ubbcluj.ro