STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LI**, Number 2, 2006

DYNAMIC PROGRAMMING AND d-GRAPHS

KÁTAI ZOLTÁN

ABSTRACT. In this paper we are going to introduce a special graph, which we have called d-graph, in order to provide a special tool for such an optimum problem's analysis which breaks down into two or more subproblems by every decision.

1. Introduction

The dynamic programming as a method for resolution of optimizing problems was worked out by Richard Bellman. His first book about dynamic programming was published in 1957 [1]. Since then until his death in 1982 he wrote several books and articles in this area. In 1962 Bellman together with Dreyfus published the book Applied Dynamic Programming [2]. In this publication they drew attention to the fact that dynamic programming can be formulated as a graph search problem. Later this subject was largely analyzed in some papers. For example Georgescu and Ionescu introduced the PD-tree notion [3]. In this paper we are going to introduce a special graph, which we have called d-graph (from division-graph), in order to provide a special tool for such an optimum problem's analysis which breaks down into two or more subproblems by every decision (see later).

2. Optimizing problems

The efficient solving of numerous programming problems implies their optimal breaking down into subproblems. In the present paper we are dealing with such optimizing problems where the following conditions are true:

- There is a target function which has to be optimized.
- The optimizing of the target function implies to break down the problem into subproblems.
- This involves a sequence of decisions.

Received by the editors: May 20, 2006.

2000 *Mathematics Subject Classification.* D.1.0 [Programming Techniques]: General, G.22. [Discrete Mathematics]: Graph Theory - Graph Algorithms, Path and Circuit Problems, Trees.

1998 *CR Categories and Descriptors.* code [**Topic**]: Subtopic – *Detail*; code [**Topic**]: Subtopic – *Detail* .

- Concerning the division into subproblems, with each decision (cut) the problem is reduced to one (I. type optimizing problems) similar, but smaller size subproblem, or breaks into two or more (II. type optimizing problems) similar, but smaller size subproblems.
- The target function is defined on the set of the problem's subproblems.
- The principle of optimality is valid for the problem, according to which the optimal solution of the problem can be built from the optimal solutions of its subproblems (the optimal value of the target function referring to the problem can be determined from the optimal values referring to the subproblems).
- Out of the different possibilities of breaking down the problem, we consider optimal that one (or that sequence of decisions) which - in accordance with the basic principle of optimality - involves the optimal construction of the solution of the problem.
- We call a subproblem trivial when the value of the target function referring to it is given by the input data of the problem in a trivial way.

Such an optimizing problem is solved efficiently with the so called dynamic programming technique.

Example 1. *Let's calculate the result of the product of matrixes $A_1 \times A_2 \times \cdots \times A_n$ (the dimensions of the matrixes are: $d_0 \times d_1, d_1 \times d_2, \ldots, d_{n-1} \times d_n$). Due to the associativity of the multiplication, we can perform this in several ways. Let's determine such a parenthesis of the product (its breaking down into subproblems) where the corresponding order of the multiplication of matrixes involves a minimal number of basic multiplications (the target function).*

For example, if

$$n = 4,\ A_1(1 \times 10),\ A_2(10 \times 1),\ A_3(1 \times 10),\ A_4(10 \times 1)$$

The optimal breaking down into subproblems: $(A1 \times A2) \times (A3 \times A4)$, which involves 21 basic multiplications. A worst solution would imply 210 multiplications: $((A_1) \times (A_2 \times A_3)) \times (A_4)$.

The structure of an optimizing problem can be described by a d-graph (division graph) , defined in the followings.

3. d-graphs

Definition 1. *We call the connected weighted digraph $G_d(V, E, C)$ a d-graph if the following conditions are fulfilled:*

(1) $V = V_p \cup V_d$ *and* $E = E_p \cup E_d$
(2) V_p *- the set of the p type nodes of the graph (p-nodes).*
$V_p = \{p_1, p_2, \ldots, p_{nr_p}\}, nr_p$ *- number of p-nodes.*
(3) *Exactly one element of the set V_p is a source node (f).*

(4) *We assign the set of p type sink nodes of G_d with $S(G_d)$ (nr_s marks the number of sink nodes).*
(5) *V_d - the set of the graph's d type nodes (d-nodes); nr_d - number of d-nodes.*
(6) *All the neighbours of the d-nodes are p type and inversely, all the neighbours of the p-nodes are of d type. Each d-node has exactly one in-neighbour of p type, which we call p-father. The out-neighbours of the p-nodes are called d-sons. Each d-node has at least one p type out-neighbour and we are going to refer to these as p-sons.*
(7) *The d-nodes are identified with two indexes: For example the notation d_{ik} refers to the d-son identified as the kth d-sons of the p-node p_i.*
(8) *E_p - the set of p type arcs of the graph (p-arcs).*
$E_p = \{(p_i, d_{ik})/p_i \in V_p, d_{ik} \in V_d\}$.
(9) *E_d - the set of d type arcs of the graph (d-arcs).*
$E_d = \{(d_{ik}, p_j)/d_{ik} \in V_d, p_j \in V_p, i < j\}$. *We should notice that the p type descendent of any p-node have bigger indexes. So in case of any d-graph the source is the 1 node.*
(10) *The $C : Ep \to R$ function associates a cost to every p-arc. We consider the d-arcs of zero cost.*

Theorem 1. *Every d-graph is acyclic.*

Proof. Let's assume that an oriented cycle exists in one of the d-graphs. According to the sixth item of the definition the p and d type nodes alternate on the cycle. Should the cycle consist of one p-node and one d-node, then the p-node is the p-father and also the p-son of node d in the same time. But this contradicts the ninth item of the definition according to which the p-sons of a d-node have always bigger indexes than its p-father. In case there are at least two nodes of both types, then let's consider p_i and pj two consecutive p-nodes of the cycle. As pi is the ancestor and in the same time the descendent of p_j - also according to the ninth item of the definition - i should be smaller and also bigger than j, which is obviously impossible. So every d-graph is acyclic. □

Conclusion 1. *The p-nodes of any d-graph can be arranged in topological order.*

The following picture presents such a d-graph where each d-node has exactly two p-sons.

Definition 2. *We call the d-graph $g_d(v, e, c)$ the d-subgraph of the d-graph $G_d(V, E, C)$, if*

- $v_p \subseteq V_p$, $v_d \subseteq V_d$, $e_p \subseteq E_p$, $e_d \subseteq E_d$ *and* $S(g_d) \subseteq S(G_d)$
- $c : e_p \to R$ *and* $c(x) = C(x)$ *for any* $x \in e_p$
- *the set of the d, respectively p type sons of any p, respectively d type node of g_d are similar in the g_d and G_d d-graphs.*

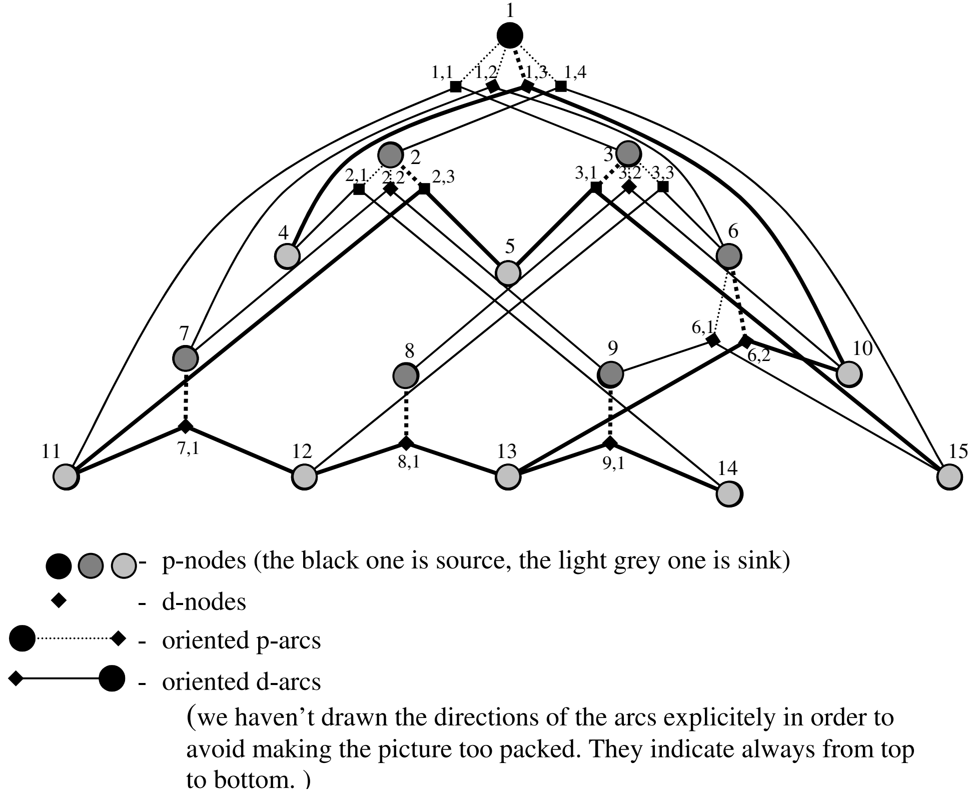

FIGURE 1. d-graph

It results from the above definition that every p-node of a d-graph unequivocally identifies the d-subgraph for which the respective node is its source.

Definition 3. *We call d-tree the d-graph where every p-node (except the sinks) has exactly one son. The source of a d-tree is called d-root and its sinks are called d-leaves. The set of leaves of the T_d d-tree are marked with $L(T_d)$.*

Definition 4. *We call the d-tree $t_d(vt, et, c)$ the d-subtree of the d-tree $T_d(Vt, Et, C)$ if*

- $vt_p \subseteq Vt_p$, $vt_d \subseteq Vt_d$, $et_p \subseteq Et_p$, $et_d \subseteq Et_d$ *and* $L(t_d) \subseteq L(T_d)$
- $c : et_p \to R$ *and* $c(x) = C(x)$ *for any* $x \in et_p$
- *the set of p-sons of any d-node of* t_d *corresponds in the* t_d *and* T_d *d-trees.*

Definition 5. *We call a d-tree $T_d(Vt, Et, c)$ the d-subtree of the d-graph $G_d(V, E, C)$ if*

- *$Vt_p \subseteq V_p$, $Vt_d \subseteq V_d$, $Et_p \subseteq E_p$, $Et_d \subseteq E_d$ and $L(T_d) \subseteq S(G_d)$*
- *$c : Et_p \to R$ and $c(x) = C(x)$ for any $x \in Et_p$*
- *the set of p-sons of any d-node of T_d corresponds in the d-tree T_d and the d-graph G_d.*

If the root of T_d corresponds to the source of G_d, then we can speak about a spanning d-subtree.

Definition 6. *By the costs of a d-tree we mean the total costs of its p-arcs.*

Definition 7. *We call the spanning d-subtree of a d-graph with the lowest costs minimal cost spanning d-subtree.*

Definition 8. *(the basic principle of optimality): We say that a d-graph has an optimal structure if every d-subtree of its optimal (having minimal costs) spanning d-subtree is itself an optimal spanning d-subtree of the d-subgraph determined by its root.*

4. Optimal Structure d-graphs

Let $G_d(V, E, C)$ be a d-graph. In the followings we are going to define a function C of p-arc-costs where every d-graph will be of optimal structure. Before doing that we are defining the node-weighing functions w_p and w_d. We mark the set of d-sons of the p-node p_i with d_son_set(p_i) and the set of p-sons of the d-node d_{ik} with p_son_set(d_{ik}).

The weight-function w_p :

$w_p : V_p \to R$

for every $p_i, i = 1 \dots nr_p$ p-node corresponds

$$w_p(p_i) = \underset{d_{ik} \in d_son_set(p_i)}{\text{optimum}} \{w_d(d_{ik})\}, \text{ if } p_i \notin S(G_d)$$

$w_p(p_i) = h_r$, if p_i is the rth sink of the d-graph

where $\{h_1, h_2, \dots h_{nr_s}\} \subset R$ is an input set which characterizes the G_d d-graph

The w_p weight of every p-node (except the sinks) is equal to the w_d weight of its "optimal d-son".

The weight function w_d :

$w_d : V_d \to R$

for every d_{ik} d-node corresponds

$$w_d(d_{ik}) = \varphi(\{w_p(p_j)/p_j \in p_son_set\ (d_{ik})\})$$

The function φ describes mathematically how the w_d weight of a d-node can be calculated from the w_p weights of its p-sons. The function φ also characterizes the G_d d-graph

After having introduced the above weight functions, we define the cost function C^* in the following way:

$$C^* : E_p \to R,\ C^*((p_i, d_{ik})) = |w_p(p_i) - w_d(d_{ik})|$$

Theorem 2. *Every d-graph $G_d(V, E, C^*)$ has optimal structure.*

Proof. As we have chosen the weight of the optimal d-sons as the weight of the p-nodes, every p-node is adjacent to at least one zero cost p-arc. It derives from this that the minimal cost spanning d-subtree and its every d-subtree will have zero costs. As C^*, by its definition, assigns positive costs to the p-arcs, it is natural that every d-subtree of the minimal cost spanning d-subtree will be a minimal cost spanning d-subtree of the d-subgraph which has a corresponding source of its root. □

5. Determination of the optimal spanning d-subtree with the implementation of the basic principle of optimality

Let $G_d(V, E, C^*)$ be an optimal structure d-graph. According to the basic principle of optimality, the optimal spanning d-subtree of any g_d d-subgraph of G_d can be determined from the optimal spanning d-subtrees of the son-d-subgraphs of g_d. Consequently we are going to determine the optimal spanning d-subtrees belonging to the nodes $p_i \in V_p (i = 1 \dots nr_p)$ in a reversed topological order. This order can be ensured if at the depth-traversing, we deal with the certain nodes at the moment we are leaving them.

We use the arrays $WP[1 \dots nr_p]$ and $WD[1 \dots nr_d]$ in order to store the weights of the p, respectively d type nodes of the d-graph G_d. At the beginning we fill up the elements of array WP corresponding to the sinks with their $h_i (i = 1 \dots nr_s)$ weights, the other elements with the value NIL. For the storage of the optimal spanning d-subtree we take array $ODS[1 \dots nr_p]$, which stores the optimal d-sons of the p-nodes. We initialize this array with the value NIL. The **initialization** procedure, depending on the nature of the optimum to be calculated, gives a suitable starting value to the array-element $WP[p_i]$ received as a parameter. The function **is_ better** analysis whether the first parameter is better than the second one, according to the nature of the optimum.

```
optimal_division(p_i)
        initialization(WP[p_i])
        for all d_ik ∈ d_son_set (p_i) do
                for all p_j ∈ p_son_ set (d_ik) do
                        if WP[p_j] = NIL then optimal_division(p_j)
                        endif
                endfor
                WD[d_ik] = φ({WP[p_j]/p_j ∈ p_son_set (d_ik)})
                if is_better(WD[d_ik], WP[p_i]) then
                        WP[p_i] = WD[d_ik]
                        ODS[p_i] = d_ik
                endif
        endfor
```

end optimal_division

Of course we call the **optimal_division** procedure for the source node, presuming that it is not a sink in the same time. The OSD values of the sinks remain NIL. The following recursive procedure, based on the ODS array prints the p-arcs of the optimal spanning d-subtree in a preorder order.

```
optimal_tree (p_i)
        write (p_i, ODS[p_i])
        for all p_j ∈ p_son_set (ODS[p_i]) do
                if ODS[p_j] ∉ NIL then optimal_tree (p_j)
                endif
        endfor
end optimal_tree
```

6. The optimizing problems and the d-graphs

A d-graph can be associated to any optimizing problem described in the introduction.

- The p-nodes represent the different subproblems given by the breaking down of the problem. The source represents the original problem, the sinks the trivial ones.
- The numbering of the p-nodes and the acyclicity given by this go hand in hand with the fact that, in the course of the breaking down, we reduce the problem to simpler and simpler subproblems.
- A p-node will have as many d-sons as the number of possibilities in which the subproblem represented by it can be broken down to its subproblems, by the respective decision. These decision possibilities are represented by the p-arcs.
- The d-nodes represent the way the respective subproblem breaks down into its subproblems with the choices given by the different decisions.
- A d-node will have as many p-sons, as the number of subproblems resulted after the disintegration - with the occasion of the decision represented by it - of the subproblem described by its p-father. This breaking down into subproblems is described by the d-arcs.
- If different sequences of decisions taken at the breaking down of a problem lead to the same subproblem, then the respective p-node will have identical p-descendents on different descent branches.
- The d-subgraphs of a d-graph express the way in which the subproblems represented by its sources can be broken down onto further, smaller subproblems.
- A certain subtree of a d-graph describes one of the breaking downs onto subproblems of the subproblem represented by its root. The spanning

subtrees of a d-graph represent the possibilities of breaking down the original problem onto its subproblems.

- The optimal structure of the d-graphs expresses the fact that the optimal solution of the problem is built from the optimal solutions of the subproblems. In other words, the corresponding subsequences of the optimal sequence of the decisions are also optimal.
- The optimal spanning d-subtree represents the optimal breaking down of the problem into subproblems (its every p-arc represents one of the decisions of the optimal sequence of decisions.).
- The wp function is nothing else but the returning of the target function to be optimized to the G_d d-graph.
- $h_1, h_2, \ldots, h_{nr_s}$ real values are the optimal values referring to the trivial subproblems of the target function, represented by the sinks.
- The nature of the optimum function is directly given by the target function of the problem and is often one of the minimum or maximum functions.
- The function φ is determined by the structure of the problem, the general rule according to which the solution of a subproblem is built from the solutions of its subproblems.

Hereby, an optimizing problem can be regarded as the determination of the weight of the source of a d-graph (the optimal value of the target function concerning the original problem) and of its optimal spanning d-subtree (optimal sequence of decisions, respectively optimal breaking down into subproblems).

We call the procedure **optimal_division**, which implements the basic principle of optimality, dynamic programming.

7. Solving a problem given as an example

Compression: A bit-sequence of n elements is given. We also have m other "shorter" sequences of bits, where there are sequences containing only one bit of 0 respectively of 1, too. Let's replace the first bit-sequences with the minimal number of short bit-sequences.

Example:

Let the first sequence of bits be 01011.

Further the short sequences are: 1:0, 2:1, 3:11, 4:010, 5:101. We can see that the original sequence of bits can be broken down to the given short sequences in several ways:

$$(0)(1)(0)(1)(1), (0)(1)(0)(11), (010)(11), (0)(101)(1), (010)(1)(1)$$

The optimal solution is of course represented by the third version, whose compressed code is 43.

How can the problem be broken down into its subproblems? In so far as the original bit-sequence is not one of the given short sequences, we cut it in two, thus

reducing its optimal compression to the compression of the sub-sequences of bits gained at the left and right side of the cut. We continue this until we get sequences of bits which appear in the given short sequences (trivial subproblems replaceable one short sequence's code).

The general subproblem is represented by the optimal compression of the subsequence $i \dots j$ of the original sequence of bits. These indexes will identify the p-nodes of the d-graph which can be assigned to this problem. The source of the problem given as an example is the p-node 15 (read one-five). The role of the WP array is played by the part of a bidimensional array $a[1 \dots n, 1 \dots n]$ situated on and above its diagonal. This part of the array can be interpreted as the implicit representation of the d-graph of the problem. The p-nodes are represented by the corresponding array-elements and we can consider as their w_p weight the length of the code of the optimal compression. The ij p-node -in so far as it is not a sink (the sequence $i \dots j$ is not part of the given short sequences)- will have a number of $(j - i + 1)$ d-sons, whose p-son-pairs will be the p-node-pairs $(ik, (k+1)j)(k = i \dots j-1)$. So the d-nodes, respectively the p and d type arcs are only implicitly present in this representation of the d-graph. Of course this also implies that we do not use a WD array. This is not necessary, as the weighing of the d-nodes of the d-graphs can be avoided by merging formulas (1) and (2) (the weight of any p-node which is not a sink can be determined from the weight of its direct p-descendents):

$$w_p(p_i) = \underset{d_j \in d_\text{son_set}\ (p_i)}{\text{optimum}} \{\varphi(\{w_p(p_k)/p_k \in p_\text{son_set}(d_j)\})\}, \text{ if } p_i \notin S(G_d)$$

In the role of the ODS array the part of the bidimensional array situated above the diagonal can be used. The array-element $a[j, i](i < j)$ implicitly represents the optimal d-son of the p-node ij by the storage of the optimal k value belonging to the optimal cut of the sequence of bits $i \dots j$. If the p-node $ij(i < j)$ is a sink, then the element $a[j, i]$ will get the value zero. The p-nodes $ii(i = 1 \dots n)$ are obviously all sinks.

The following picture (see Figure 2.) represents the d-graph of the sample problem, as it is hidden in the array **a** storing the optimal values of the subproblems. We have highlighted the optimal spanning d-subtree of every d-subgraph with the source ij.

With such a representation of the d-graph the traversing of the non-sink p-nodes in a reversed topological order can be achieved by the simple traversing of the array-elements situated above the main diagonal (for example row by row from below upwards left to right). As the optimal code of any sequence of bits is the concatenation of the optimal codes of the subsequences given by its optimal cut, the function φ is a simple additive function. As we are looking for the compression with the shortest code, the function **optimum** will calculate a minimum. The input data is stored by the variables $n, m, b[1 \dots n]$ and $sequence[1 \dots m]$. The

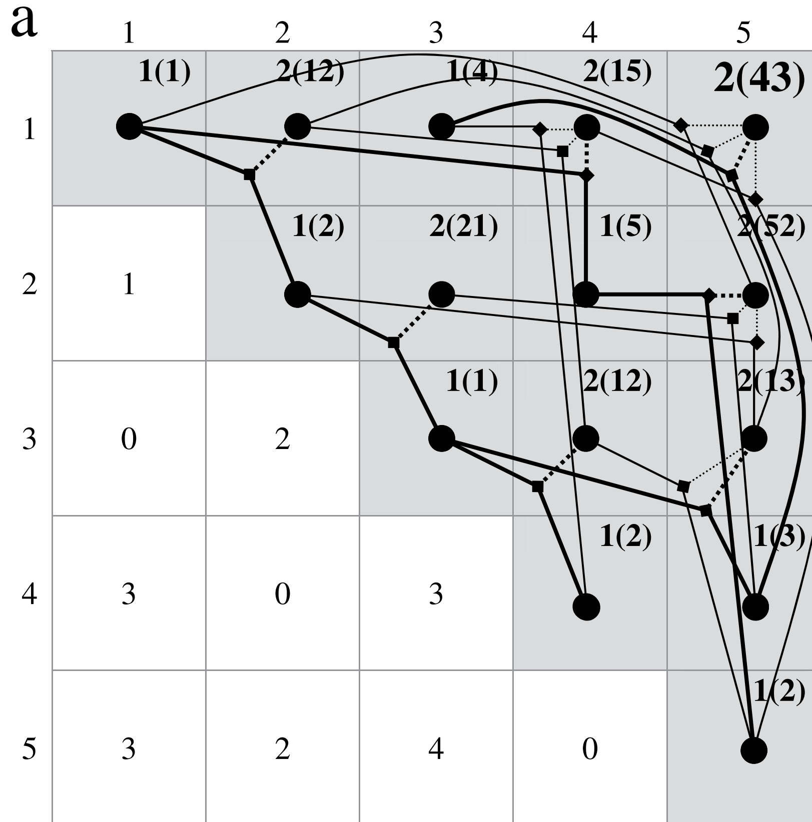

FIGURE 2. The d-graph stored in an implicit way in the array **a**

function **nr_sequence** $(\mathbf{i}, \mathbf{j})$ checks whether the sequence of bits $b[i \dots j]$ is present in the given short sequences. If yes, then it returns its code (its index from the array *sequences*), if no, it returns zero. In parallel with the filling up of the array a we store the optimal codes themselves, too, in an array COD$[1 \dots n, 1 \dots n]$ (in the above picture we have represented them in brackets). The function **concate(cod1,cod2)** concatenates the codes received as parameters.

```
for i=1,n,1 do
    a[i,i]=1
    COD[i,i]=nr_sequence(i,i)
endfor
```

```
for i=n-1,1,-1 do
      for j=i+1,n,1 do
          cod=nr_sequence(i,j)
          if cod>0 then
               a[i,j]=1
               COD[i,j]=cod
               a[j,i]=0
          else
               a[i,j]=0
               for k=i,j-1,1 do
                   if a[i,k]+a[k+1,j]>a[i,j] then
                       a[i,j]=a[i,k]+a[k+1,j]
                       COD[i,j]=concate(COD[i,k],COD[k+1,j])
                       a[j,i]=k
                   endif
               endfor
          endif
      endfor
endfor
```

The optimal code of the original sequence of bits gets into the array COD[1,n]. In case we would also like the optimal parenthesis of the bit-sequence, we can obtain this by traversing in depth the optimal spanning d-subtree of the d-graph based on array a.

```
depthfirst(i,j)
      write '('
      if i=j OR a[j,i]=0 then
          for k=i,j,1 do
               write b[k]
          endfor
      else
          depthfirst(i,a[j,i])
          depthfirst(a[j,i]+1,j)
      endif
      write ')'
end depthfirst
```

8. Conclusions

First of all it is interesting to remark that in case of the I. type optimizing problems the attached d-graphs can be reduced to a "normal graphs" (since every d-node has an unique p-son they can be left out from the graph by matching their

p-father directly with their unique p-son). In this special situation the optimal solution will be represented by the optimal root-leaf path of the graph. By introducing the d-graphs, the consistent discussion of several optimizing problems has become possible, and also the theoretical basis of the dynamic strategies related to them. The relation is similar to the one between the greedy algorithm and the theory of matroids.

References

[1] R. Bellman, Dynamic Programming, Princeton University Press, New Jersey, 1957.
[2] R. Bellman, S. Dreyfus, Applied Dynamic Programming, Princeton University Press, New Jersey, 1962.
[3] H. Georgescu, C. Ionescu, The Dynamic Programming Method, a New Approach, STUDIA Universitatis Babes-Bolyai, Cluj, 43, 1999, pp. 23-38.

E-mail address: `katai_zoltan@ms.sapientia.ro`