Acta Mathematica Academiae Paedagogicae Nyíregyháziensis
24 (2008), 201–208
www.emis.de/journals
ISSN 1786-0091

DYNAMIC PROGRAMMING AS OPTIMAL PATH PROBLEM IN WEIGHTED DIGRAPHS

KÁTAI ZOLTÁN

Abstract. In this paper we are going to present a method that can help us to deal with several dynamic programming problems as optimal path problems in weighted digraphs, even problems which apparently don't have anything in common with graph theory. Furthermore, this approach makes possible such a classification of the dynamic programming strategies which can help in the comprehension of the essence of this programming technique.

1. Introduction

Several books treat the problem of dynamic programming by presenting the principles standing at the basis of the technique and then giving a few examples of solved problems. For instance the book called Algorithms by Cormen, Leiserson and Rivest [2], mentions the optimal substructures and the overlapped subproblems as elements of the dynamic programming. Răzvan Andone and Ilie Garbacea are talking about the three basic principles of the dynamic programming in their book, Basic Algorithms [1]:

(1) avoid the repeated solving of identical subproblems by saving the optimal subsolutions
(2) we solve the subproblems advancing from the simple toward the complex
(3) the principle of optimality.

The book Programming Techniques by Tudor Sorin [4] gives a certain classification of the dynamic programming strategies: forwards method, backwards method and mixed method. In this paper we would like to go further in the study and classification of the dynamic programming strategies. By presenting the characteristics of certain dynamic programming strategies on the decision tree hidden behind the optimizing problems we offer such a clear tool for their study and classification which can help in the comprehension of the essence of this programming technique.

2000 *Mathematics Subject Classification.* 90C39, 49L20, 68R10, 05C12, 05C38.
Key words and phrases. dynamic programming, graph theory, optimal path algorithms.

Dynamic programming is often used to solve optimizing problems. Usually the problem consists of a target function which has to be optimized through a sequence of (optimal) decisions. In this paper we are dealing with problems in case of which the problem is reduced, by each decision, to *one* similar problem of smaller size (see the figure below).

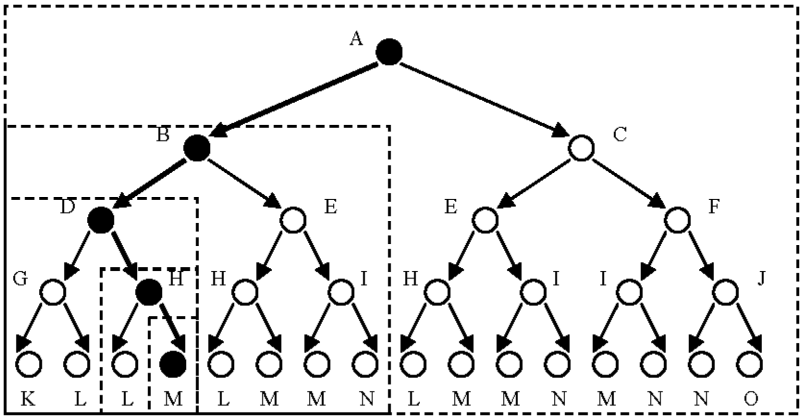

FIGURE 1. Decision tree

Considering the fact that the subproblems are similar, we can speak about their general form. By general form we always mean a form with parameters. To comprehend the structure of a problem means, amongst others, to clarify the followings:

- what is the general form of the subproblems
- which parameters describe this
- which are the parameter values for which we get the original problem, respectively the trivial subproblems as marginal cases of the general problem.

2. The Theory of Optimality

The dynamic programming is built on the principle of optimality. We could even say that it is the implementation of this basic principle. We can express the principle of optimality in the following way: *the optimal solution is built by optimal subsolutions.* In other words the optimal solution of the problem can be built with the optimal solutions of the subproblems. The dynamic programming follows this strategy: starts from the optimal solutions of the trivial subproblems and builds the optimal solutions of the more and more complex subproblems and eventually of the original problem. This is why we say that it advances from the simple to the complex or that it solves the problem from the bottom to the top (bottom-up way).

An other feature of the dynamic programming is that it records the optimal values of the optimal solutions of the already solved subproblems (the optimal values of the target function to be optimized concerning the respective subproblems). For this we are generally using an array we are going to represent with c. This array will have one, two or more dimensions depending on the number of parameters describing the general form of the subproblems.

The basic principle of optimality is expressed by a recursive formula, which describes mathematically the way in which the solutions of the more and more complex subproblems are built from the simpler subproblems' optimal solutions. Obviously this is a formula where the way of the optimal decision making has been built in. The recursive formula has been drafted for the elements of the array c, so it is working with the optimal values of the subproblems.

The core of the algorithm is represented by the filling of the corresponding elements of the array c according to the strategy given by the recursive formula. Although the array c stores one to one only the optimal values of the subproblems, still it contains enough information to reconstruct the sequence of the optimal decisions. It is often suitable to somehow store the optimal choices themselves when filling the array c. This could facilitate or even make faster the reconstruction of the optimal decision-sequence.

When the principle of optimality is valid for a problem, this could considerably reduce the time necessary to build the optimal solution, because in building it we can rely exclusively on the optimal solutions of the subproblems.

3. Dynamic programming problems as optimal path problems

In the followings we are going to consider array c (where the optimal values which represent optimal solutions of subproblems are stored) as implicit representation of a weighted digraph. By this approach all dynamic programming problems (which are treated in this paper) can be traced back to a graph theory problem, more precisely to the problem of determining the optimal weight paths leading from a given node to all the others. Let us bring to life this graph:

- The graph's nodes represent the subproblems. Thus, we can consider the used elements of array c storing the optimal solutions of the subproblems as such ones which represent the nodes of the graph.
- The graph's edges represent possible choices. These appear in the representation of graph only implicitly - through the recursive formula. The graph has a directed (A, B) edge if, according to the recursive formula, the array-element corresponding to node A depends directly from the array-element corresponding to node B.
- The weights of edges basically reflect the “weights” of choices and in general they arise immediately from the input data.
- The optimal sequence of decisions will obviously be represented by an optimal path in the graph. As we have referred to it, the costs of the optimal cost paths will be stored in the elements of array c.

We can distinguish three cases [3], for each of which there are well-known graph-algorithms:

(1) The graph is cycle free. In this case there is an algorithm based on the topological order of the nodes, with the $O(N+M)$ complexity (N and M are the number of the nodes, respectively the edges of the graph) [2].
(2) The graph contains circles, but it has no negative edges. For this case there is the Dijstra algorithm, which determines the minimal weight paths with a priority queue in the increasing order of the optimal values. The grade of complexity of the Dijstra algorithm is $O(N^2)$, but it can be improved, if we implement the priority queue with a binary heap $(O((N+M)\lg N)$ or with a Fibonacci heap $(N \lg N + M)$ [2].
(3) The graph has negative edges, but has no negative total weight circle reachable from the root. This problem is solved by the Belmann-Ford algorithm $(O(NM))$ [2].

Let us look at examples for each cases

Problem 1. There are two sequences which are stored in arrays $a[1\dots n]$ and $b[1\dots m]$. Determine there longest common subsequence.

Example. $n=4, m=3$, $a=(2,\mathbf{9},\mathbf{5},7)$; $b=(\mathbf{9},3,\mathbf{5})$. The longest common subsequence: 9, 5

The general subproblem. Determine the longest common subsequence of array segments $a[i\dots n]$ and $b[j\dots m]$. Since, we have two independent parameters, the optimal value of the general subproblem will be stored in the cell $c[i][j]$ of bidimensional array c. We get the original problem for the values $i=1$ and $j=1$, and the trivial ones for values $i=n+1$ or $j=m+1$ (at least one of the array segments is empty). The optimum value of the trivial subproblem is obviously zero.

The recursive formula.

```
c[i][m+1]=0, for all  i=1,n+1
c[n+1][j]=0, for all  j=1,m+1
for all  1 ≤ i ≤ n, 1 ≤ j ≤ m
       c[j][j]=1+c[i+1][j+1],  if  a[i]=b[j]
       c[j][j]=max(c[i][j+1],c[i+1][j]), if  a[i] ≠ b[j]
```

The $G(V,E,w)$ graph. Set of nodes:

$$V=\{(i,j) : 1 \le i \le n+1,\ 1 \le j \le m+1\}$$

Set of edges:

$$\begin{aligned} E=\{((i_1,j_1),(i_2,j_2)) : 1 \le i_1, i_2 \le n+1, 1 \le j_1, j_2 \le m+1,\\ (i_1+1=i_2)\wedge(j_1+1=j_2)\wedge(a[i_1]=b[j_1])\vee\\ ((i_1=i_2)\wedge(j_1+1=j_2)\vee(i_1+1=i_2)\wedge(j_1=j_2))\wedge(a[i_1]\neq b[j_1])\}\end{aligned}$$

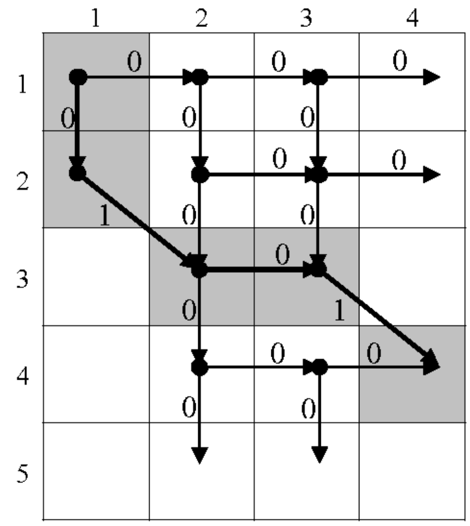

FIGURE 2. Array c as a weighted digraph. We highlighted the optimal path

The weight function: $w\colon E \to R$,

$$w((i_1, j_1), (i_2, j_2)) = 1 \text{ if } (i_1 + 1 = i_2) \wedge (j_1 + 1 = j_2)$$
$$w((i_1, j_1), (i_2, j_2)) = 0 \text{ if } (i_1 = i_2) \wedge (j_1 + 1 = j_2) \vee (i_1 + 1 = i_2) \wedge (j_1 = j_2)$$

The optimal path problem. To determine the longest common subsequence of two sequence is identical with determining the maximal weight path of graph G from node $(1, 1)$ to one of the $(i, m+1)$ or $(n+1, j)$ $(1 \le i \le n+1, 1 \le j \le m+1)$ form nodes. Since the graph G is circle free, a topological order based algorithm solves the problem in the most efficient way. We highlighted in the above figure the optimal sequence of decision represented by the maximal weight path.

Although this problem apparently doesn't have any connections with graphs, the method presented above helped us to trace it back to an optimal path problem.

Problem 2. A labyrinth from Disneyland is represented through a bidimensional array, $L[1 \ldots n][1 \ldots m]$ (zero means wall, and the non zero values represent how much money the visitors receive when they pass that position). In labyrinth we can move up, down, left and right. Determine the path which leads from the position (x, y) to the outside of the labyrinth and which is the most "advantageous" to the company who finances the labyrinth.

Example. $n = 8, m = 8$, white-free place, grey-wall, $x = 3, y = 4$ (we didn't load the figure with the amounts of money which can be received in free places)

The general subproblem. Determine the most "advantageous" path which leads from position (x, y) to position (i, j). Since we have two independent parameters, the optimal value of the general subproblem will be stored in cell $c[i][j]$

of the bidimensional array c, which has the same sizes than array L. We have the original problem when $i = 1$ or $j = 1$ or $i = n$ or $j = m$, and the trivial subproblem for the values $i = x$ and $j = y$. The optimum value of the trivial subproblem will obviously be zero.

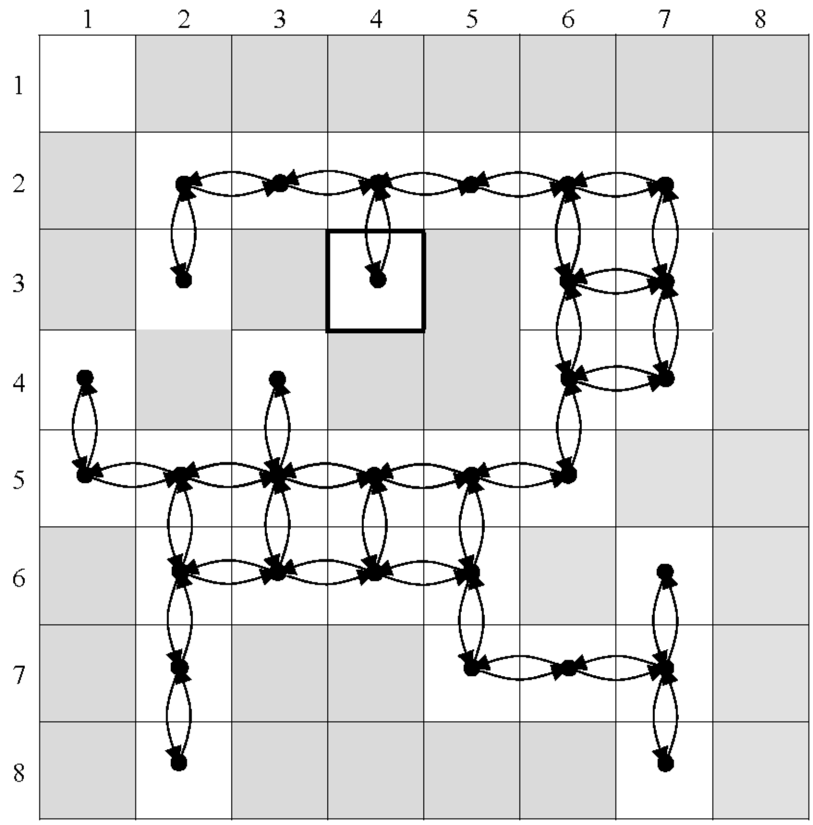

FIGURE 3. The graph attached to the problem 2 and 3

The recursive formula.

```
c[x][y]=0
for 1≤i≤n, 1≤j≤m, where L[i][j]≠0:
     c[i][j]=L[i][j]+min(c[i-1][j], c[i][j+1], c[i+1][j], c[i][j-1])
```

supposing that this neighbour positions exist and they aren't walls.

The $G(V, E, w)$ graph. Set of nodes:

$$V = \{(i, j) : 1 \le i \le n, 1 \le j \le m, L[i][j] \ne 0\}.$$

Set of edges:

$$E = \{((i_1, j_1), (i_2, j_2)) : (i_1 = i_2) \wedge |j_1 - j_2 = 1| \vee |i_1 - i_2 = 1| \wedge (j_1 = j_2)\}.$$

The weight function: $w\colon E \to R$,

$$w((i_1, j_1), (i_2, j_2)) = L[i_2][j_2]$$

The optimal path problem. The problem of the most "advantageous" getting out from the labyrinth is identical with determining the minimal weight path from node (x, y) of the graph G to one of the nodes $(i, 0), (i, m), (0, j)$ or (n, j) $(1 <= i <= n,\ 1 <= j <= m)$. Since G contains circles but doesn't have negative edges, the problem can be solved by Dijstra algorithm.

Problem 3. We have the same problem with one difference:

- Array L also contains negative elements which represent positions where the visitors lose the concerning amount of money.
- There are neighbor (free) cells with unidirectional connections between them.
- The visitors can't do loops in deficit.

Although we can attach to the problem the same graph that in case of the second problem, this graph would contain negative edges but without containing circles with negative weight. Regarding that, the minimal weight path will be determined by Bellman Ford algorithm.

Although the last two problems can be considered as optimal path problems by themselves, we didn't use this particularity of them. We attached the nodes of graph to the subproblems which had been derived from problem's breaking-down and the edges were defined through recursive formula.

4. Conclusions

There are two fundamental characteristics of dynamic programming problems:

- The problem's breaking down leads to several identical subproblems.
- The principle of optimality is valid for the problem.

The first characteristic in general guarantees a polynomial nodes' number for the graph. The second, through the recursive formula, makes the defining of edges possible. This graph theory approach of dynamic programming problems makes possible a certain classification of them and can help us to choose the correct strategy for subproblems solving.

An other strength of this approach is its illustrability.

References

[1] R. Andone and I. Garbacea. *Fundamental Algorithms a C++ Perspective.* Libris Press, 1995. in Romanian.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA, 1990.

[3] Z. Kátai. Dynamic programming strategies on the decision tree hidden behind the optimizing problems. *Informatics in Education*, 6(1):1–24, 2007.

[4] S. Tudor. *Programming Techniques.* L&S InfoMat Press. in Romanian.

Received January 16, 2007.

Sapientia University,
Mathematics and Computer Sciences Department,
Tirgu Mures/Corunca, Soseaua Sighisoarei, 1C,
540485 Romania
E-mail address: katai_zoltan@ms.sapientia.ro