Acta Univ. Sapientiae, Informatica, 1, 2 (2009) 149–164

Automated dynamic programming

Zoltán Kátai
Sapientia Hungarian University of Transylvania, Cluj, Department of Mathematics and Informatics
Tg. Mureş, Romania
email: katai_zoltan@ms.sapientia.ro

Ágnes Csiki
INTELES SRL
Str. Gheorghe Pitut nr.1
Oradea, Romania
email: csiki_agi@yahoo.com

Abstract. Since the first book in dynamic programming was published in 1957, this algorithm design strategy has become a current problem solving method in several fields of science. The dynamic programming problem solving process can be divided into two steps. Firstly, we establish the functional equation of the problem, a recursive formula that implements the principle of the optimality (mathematical part). Secondly, a computer program is elaborated that processes the recursive formula in bottom-up way (programming part). In this paper we are going to present a method and a software tool that automates the programming part of the dynamic programming process in case of several problems.

1 Introduction

Dynamic programming as optimizing method was proposed by Richard Bellman. Since the first book [1] in dynamic programming was published in 1957, this algorithm design strategy has become a current problem solving method in several fields of science (Applied mathematics [2], Computer sciences [3], Artificial Intelligence [5], Bioinformatics [4], Macroeconomics [9], etc.). The dynamic programming problem solving process can be divided into two steps. Firstly, we establish the functional equation of the problem, a recursive formula that implements the principle of the optimality. Secondly, a computer

AMS 2000 subject classifications: 68W40, 90C39, 68R10, 05C12
CR Categories and Descriptors: D.1 [PROGRAMMING TECHNIQUES]
Key words and phrases: dynamic programing, graph theory, shortest path algorithms

program is elaborated that processes the recursive formula in bottom-up way. We will refer to these two steps as mathematical and programming parts of the dynamic programming. Numerous researchers in the above mentioned various fields of applications are not experts in programming. In this paper we are going to present a method and a software tool that automates the programming part of the dynamic programming process in case of several problems.

2 The mathematical part

Dynamic programming is often used to solve optimizing problems. The problem usually consists of a target function, which has to be optimized through an optimal sequence of decisions. The dynamic programming is built on the principle of optimality: the optimal solution is built by optimal sub-solutions. This principle is expressed by a recursive formula (functional equation), which describes mathematically the way the more and more complex optimal sub-solutions are built from the simpler ones. Obviously, this is a formula where the way of the optimal (minimum or maximum) decision making has been built in. Once the functional equation is established, the problem can be considered mathematically solved.

We assume that the recursive branches of the functional equation have the following general form: $c(A) = \min / \max\{f_A(c(B_i)) | i = 1, 2, ..., n\}$, where c denotes the target function. $c(A)$ represents the optimum value attached to sub-problem A. This optimum directly depends on the optimum value of the one of sub-problems B_i. More exactly, it depends on $c(B_i)$, which optimizes (minimizes or maximizes) function f_A. Function f_A depends on the problem to be solved.

3 The programming part

The programming part of the problem solving process is built on another principle of the dynamic programming: the optimal values of the target function concerning the already solved sub-problems are stored (often in an array that we denote by C). According to the principle of the optimality we are interested only in the optimal solutions of the sub-problems. This technique, often called memoization or result catching, makes it possible to avoid the repeating computation for overlapped sub-problems, which are also characteristic for dynamic programming problems. The core of the computer program that implements the dynamic programming algorithm consists in computing

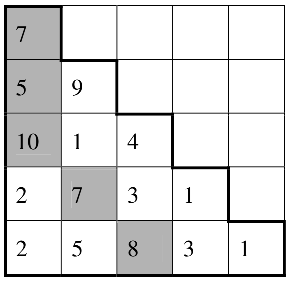


Figure 1: Array A associated to Triangle problem.

the corresponding elements of the array C in bottom-up way according to the strategy given by the recursive formula. An efficient strategy solves each sub-problem before its optimum value is needed by any other sub-problem. The complexity of this programming task varies from problem to problem. It is often nontrivial to write a code that evaluates the sub-problems in the most efficient order.

In [7] we presented three examples; since the computer program works on the elements of the array C, the recursive formula is generally drafted for these elements:

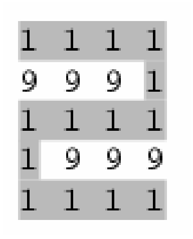
1. Triangle (International Olympiad in Informatics, Sweden, 1994): On and under the main diagonal of a square matrix with n rows there are natural numbers. We assume that the matrix is stored in the bi-dimensional array A. Determine the longest path from peak (element a_{11}) to the base (n-th row), considering the following:

- On a certain path element a_{ij} can be followed either by element $a_{i+1,j}$ (down), or by element $a_{i+1,j+1}$ (diagonally to the right), where $1 \leq i < n$ and $1 \leq j \leq i$.
- By the length of a path we mean the sum of the elements to be found along the path.

For example, should for $n = 5$ the matrix be the following (see Fig. 1.), then the longest path from the peak to the base is the shaded one and its length is 37.

2. Office-building_1: Let A be a matrix whose elements a_{ij} ($i = 1, \ldots, n, j = 1, \ldots, m$) represent an one-storied rectangular office building. The elements of the matrix represent the offices and they store the taxes to be paid by anyone who enters the respective room. There is a door between any two neighbouring elements. You can enter the building only at office with position $(1, 1)$ and leave it only at position (n, m). Which is the minimal loss of money you can get through the building with?

For example, see Fig. 2. ($n = 5$, $m = 4$). The minimal loss of money is 14, which we got by following the shaded path.



1	1	1	1
9	9	9	1
1	1	1	1
1	9	9	9
1	1	1	1

Figure 2: Array A associated to Office-building_1.

3. Office-building_2: The same problem with the following differences:

- There are offices where they do not take money, but they give a certain amount of money (”negative tax”).
- There are one-way doors (with one-side door-handles). Array B, whose elements b_{ij} ($i = 1, \dots, n, j = 1, \dots, m$) are binary strings with 4 characters ($'0'$ or $'1'$), stores the door-codes of the offices. The first binary character of the code represents the up-door, the second the right-door, the third the down-door and the fourth the left-door. For example, code "0101" means that we can leave the office only to right and left directions.
- We assume that there is no such office-tour of the building, going along which we could increase our amount of money.

Determine the most favorable way of getting through the building. For example, see Fig. 3. ($n = 5$, $m = 4$) and Fig. 4. The most favorable path goes through the same offices this time too, and means a loss of money of 7.

In the case of all the above-presented problems the array C is bi-dimensional, and the recursive formulas that implement the principle of optimality have the following forms:

1	1	1	1
19	19	19	1
3	1	3	1
-2	19	19	1
-6	-2	3	1

0111	0111	0111	0011
1110	1111	1111	1011
1110	1111	1111	1011
0110	1111	1111	1011
0100	1100	1101	1101

Figure 3: Array A and B associated to Office-building_2 problem.

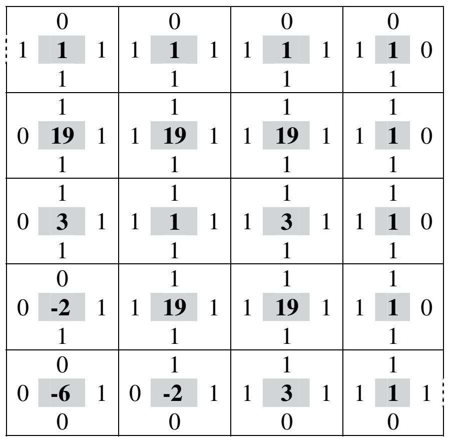


Figure 4: The Office-building.

Problem 1 (Triangle): (Element c_{ij} stores the length of the longest path from the position (i, j) to the nth row; The trivial sub-problems are represented by the cells from the n-th row, and the optimal value of the original problem is going to be stored in cell c_{11}.)

$c_{nj} = a_{nj}$, $1 \leq j \leq n$

$c_{ij} = a_{ij} + \max(c_{i+1,j}, c_{i+1,j+1})$, $1 \leq i < n, 1 \leq j \leq i$

Problems 2 and 3 (Office-buildings): (Element c_{ij} stores the length of the optimal path between the offices from the positions $(1, 1)$ and (i, j); The trivial sub-problem is represented by cell c_{11}, and the optimal value of the original problem is going to be stored in cell c_{nm}).)

$c_{11} = a_{11}$,
otherwise
$c_{ij} = a_{ij} + \min(c_{i-1,j}, c_{i,j+1}, c_{i+1,j}, c_{i,j-1})$
(assuming that the rooms with the respective positions exist,
and they have "proper doors")

In the first example (the triangle problem) the chain of the recursive calls is cycle-free. In such a situation, there is an elegant technique that does not require the programmer to establish the evaluation order of the sub-problems: recursion with result catching [11]. By catching the results of all recursive calls, the second and subsequent evaluations of any sub-problem become constant-time operations, reducing the overall running time considerably. Recursion with result catching is very easy to implement in softwares like Maple, Matlab, Mathematica, etc. (In Maple we use the `option remember` instruction.) These softwares and the recursion with result catching technique are within the programming reach of most of the researchers, even if they are not experts in programming.

For instance: On the one hand, procedure `triangle_A` is an immediate transcription of the recursive formula, but, unfortunately, this algorithm has exponential time complexity (inefficient divide and conquer strategy). On the other hand procedure `triangle_B` differs from `triangle_A` only in one line (`option remember;`) and it has polynomial complexity (dynamic programming technique).

```
triangle_A := proc(n, a, i, j)
  if i<n then
    return a[i,j]+max(triangle_A(n,a,i+1,j),
                      triangle_A(n,a,i+1,j+1));
  else return a[i,j];
  end if;
end proc:
```

```
triangle_B := proc(n, a, i, j)
  option remember;
  if i<n then
    return a[i,j]+max(triangle_B(n,a,i+1,j),
                      triangle_B(n,a,i+1,j+1));
  else return a[i,j];
  end if;
end proc:
```

In the case of the second and third sample-problems the chain of the recursive calls is circular. For example, the optimal value of cell c_{23} may depend on the optimal value of cell c_{33}, and, conversely, c_{33} may also depend on c_{23}. In such situation, the recursive approach is excluded (to avoid infinite recursive call). Furthermore, there are no easy dynamic programming solutions for these type of problems (for more details, see [7]). The method and software tool presented in this paper are especially useful in case of such problems.

4 Dynamic programming as optimal path algorithm in weighted digraphs

In the followings we are going to consider the recursive formula as an implicit description of a weighted digraph. By this approach several dynamic programming problems can be interpreted as optimal path problems between two specific vertices of this graph [8].

- The vertices of the graph represent the sub-problems. Thus, we can consider the used elements of array C storing the optimal values of the sub-problems as such ones, which represent the vertices of the graph.
- The arcs of the graph represent possible choices (optimize means here the choice of optimal). The graph has an arc from vertex B to vertex A if, the optimum value of the array-element corresponding to vertex A may directly depend on the optimum value of the array-element corresponding to vertex B, according to the recursive formula. For example, if $c_A = min/max\{f_A(c_{B_i})|i = 1, \ldots, n\}$, then there are arcs from vertices B_i to vertex A.
- The weights of the arcs reflect the weights of choices.
- The optimal sequence of decisions is represented by the optimal path between the vertex representing the trivial sub-problem and the vertex that represents the original problem. If the problem has more than one trivial sub-problem, we introduce a dummy trivial-node, which is connected to all trivial vertices.

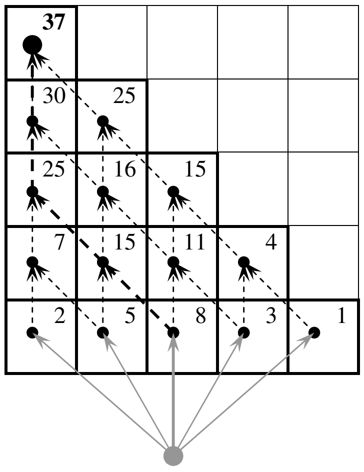


Figure 5: Cycle free digraph attached to the Triangle problem.

Figures 5 and 6 show the graphs behind the sample-problems. The big black node represents the original problem and the gray one the trivial sub-problem. In the case of the triangle problem we introduced a dummy trivial-node. The optimal paths are represented by thick arrows. The cells of array C store the optimum values of the corresponding sub-problems.

We distinguish three cases [8]:

- 1. The attached graph is cycle free. In this case the most efficient optimal path algorithm is based on the topological order of the vertices. The time complexity of this algorithm is $\mathsf{O}(\mathsf{N}+\mathsf{M})$ (N and M are the numbers of the vertices and arcs, respectively) [3].

- 2. The graph contains cycles, but there are no negative weight arcs. For this case the best choice is Dijkstra's shortest path algorithm. The time complexity of the most efficient implementation of this algorithm is $\mathsf{O}(\mathsf{N}\log\mathsf{N}+\mathsf{M})$ [3].

- 3. The graph has negative arcs, but it has no negative weight cycles. This shortest path problem is solved by the Belmann-Ford algorithm ($\mathsf{O}(\mathsf{NM})$) [3].

The optimal (shortest) path problem to be solved is the following. Given a weighted digraph ($\mathsf{G}(\mathsf{V},\mathsf{E},w)$, V: set of vertices, E: set of arcs, $w:\mathsf{E}\to\mathsf{R}$

weight function) with N vertices $1, 2, \ldots, N$ and M arcs, determine the shortest paths from the vertex s (source) to all the other vertices (destinations).

We denote with $c(v)$ (the optimum value attached to vertex v) the weight of the shortest path from the source (s) to vertex v. ($c(s) = 0, c(u) = \infty$ for all vertices u that are not reachable from vertex s)

The above mentioned shortest path algorithms are based on the following lemmas and propositions: (For proofs and further details see [3] and [6])

Lemma 1. Parts of any shortest path are also shortest paths. (Principle of optimality)

Lemma 2. All s-source shortest paths constitute an s-rooted tree called optimal-paths-tree.

Lemma 3. The optimal-paths-tree can be built progressively starting with vertex s. At each step the tree is extended with a new arc that attaches to the tree a new vertex. (Implementation of the principle of optimality)

Lemma 4. If vertex u is the immediate predecessor of vertex v on the optimal path from s to v, then: $c(v) = c(u) + w(u, v)$. (The optimum value of vertex v is based on the optimum value of vertex u, and can be computed on the basis of the weight of arc (u, v))

Corollary 1. The optimum values of all vertices that are reachable from s depend on the optimum value of the one of their in-neighbours.

Lemma 5. The optimum values have to be computed according to a topological order of the vertices with respect to the optimal-paths-tree.

Lemma 6. Assuming $c(s) = 0$, the building process of the optimal-paths-tree consist in applying the formula from Lemma 4 on all arcs of the optimal-paths-tree in their topological order.

Assuming that the optimum values attached to the vertices are going to be generated in array C, we define the following updating operation (operator `update`) on the basis of arc $(u, v) \in E$:

```
update(u,v)
    if $c_v > c_u + w(u,v)$ then $c_v = c_u + w(u,v)$
    end_if
end_update
```

Corollary 2. If $c_s = 0$ and $c_u = \infty$ for all $u \in V \backslash \{s\}$, then applying `operator update` on all arcs of the optimal-paths-tree in their topological order results in $c_u = c(u)$ for all $u \in V$.

Lemma 7. For all arcs $(u, v) \in E$ it is true that: $c_v \leq c_u + w(u, v)$.

Lemma 8. If $c_s = 0$ and $c_u = \infty$ for all $u \in V \backslash \{s\}$, then applying operator `update` on any arc-sequence that includes as sub-sequence the arc-sequence required by Lemma 6 results in $c_u = c(u)$ for all $u \in V$.

All the three shortest path algorithms mentioned above apply the following strategy:

- it generates such an arc-sequence that includes as sub-sequence the arc-sequence required by Lemma 6,
- it applies operator `update` on all arcs of the generated sequence.

Lemma 9. If G is cycle free, then the topological-sequence of all arcs of G includes as sub-sequence the arc-sequence required by Lemma 6.

Proposition 1. If G is cycle free, $c_s = 0$ and $c_u = \infty$ for all $u \in V \backslash \{s\}$, then applying operator `update` on all arcs in their topological order results in $c_u = c(u)$ for all $u \in V$.

Lemma 10. If all arcs in G have non negative weights and vertex u is a predecessor of vertex v on the optimal path from s to v, then: $c(u) \leq c(v)$.

Corollary 3. If all arcs in G have non negative weights, then the optimum values of all vertices v that are reachable from s may only depend on in-neighbours that have optimum values less or equal than $c(v)$.

According to Lemma 10 and Corollary 3 Dijkstra's algorithm determines the shortest paths according to the ascending order of their weights.

Proposition 2. (Dijkstra's algorithm) If all arcs in G have non negative weights, $c_s = 0$ and $c_u = \infty$ for all $u \in V \backslash \{s\}$, then the algorithm that

- starts with vertex s,
- in each step attaches the arc that links to the tree the vertex that is 'closest' to root s (according to the current values stored in array C) to the growing optimal-paths-tree,
- applies operator `update` on all out-arcs of the currently attached vertex,

results in $c_u = c(u)$ for all $u \in V$.

Proposition 3. (Bellman-Ford algorithm) If G has no negative cycles, $c_s = 0$ and $c_u = \infty$ for all $u \in V \backslash \{s\}$, then the algorithm that

- chooses an arbitrary sequence of all arcs in G,
- applies operator `update` to the chosen sequence, again and again, until no more changes in array C,

results in $c_u = c(u)$ for all $u \in V$.

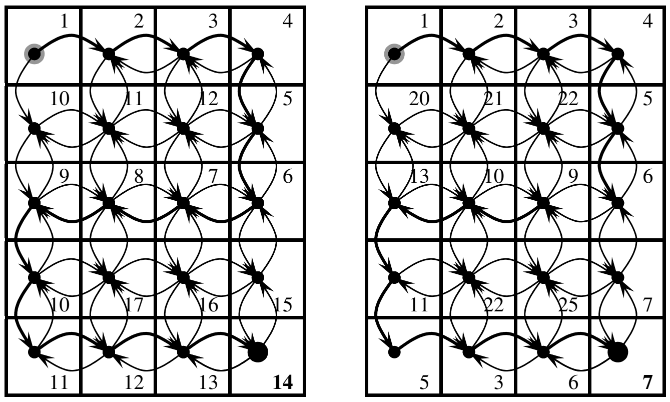


Figure 6: Digraphs (without negative arcs/without negative cycles) attached to the Office-buildings problems.

5 The method and the software tool

The core idea of the algorithm behind the software is that we represent explicitly the graph described implicitly by the recursive formula. Since the dimension of the array C varies from problem to problem we treat it as one-dimensional (row-major-index). The used cells (the vertices) of array C are going to store the optimum values of the corresponding sub-problems and pointers to their out-neighbour cells.

There are two strategies to transpose the functional equation of dynamic programming into an algorithm: the direct method (direct-conversion of the functional equation into an iterative/recursive procedure) and the successive approximation methods (after an initial approximation, the cells that are going to store the optimum values are successively updated –improved– either by the functional equation itself or by an equation related to it) [10].

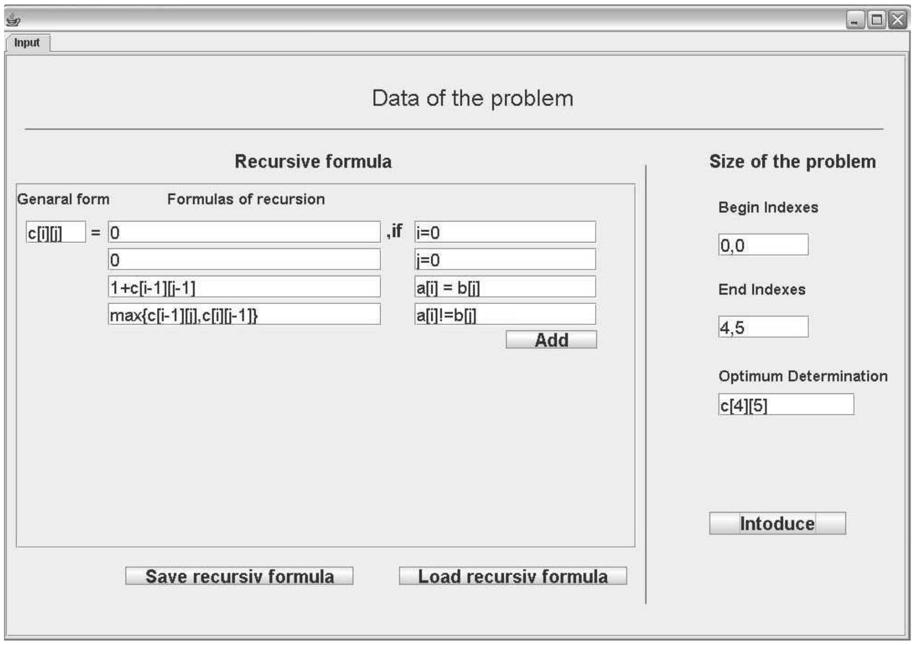
Another classification of the dynamic programming strategies is based on the way the optimum values of the sub-problems are computed. The so-called pull-approach computes directly (not by an updating process) the optimum value of the current node on the basis of the already computed optimum values of its immediate predecessors. This approach is an immediate application of the functional equation, and can be used only for the acyclic graphs. The recursion with result catching technique applies this approach [10].

The key idea in the case of the push-approach (adaptable to all three cases) is to propagate any improvement that has been made in the current vertex u to its out-neighbors. More exactly, if v is an out-neighbor of the current vertex u, then cell c_v (the cell corresponding to the sub-problem that is represented by the vertex v) is updated on the basis of the arc (u, v). In other words, if value $f_v(c_u)$ 'is better' than the current value of the cell c_v, then c_v is updated with the value $f_v(c_u)$. The algorithm ends when any other improvements cannot be performed [10]. All the three optimum path algorithms we are using in the software apply successive approximation and push-approach.

The topological algorithm traverses the vertices of the graph (starting with the (dummy)trivial-node) according to their topological order, and updates the c_v value of all out-neighbors of the current vertex u on the basis of the arc (u, v). At the moment we have arrived to a vertex, the corresponding element in array C already stores the optimum value. The algorithm only confirms this optimum. The succession the optimum values of the vertices are determined is predestinated by the topological order of these vertices. The topological order of the vertices can be established by a Depth First Search (DFS) procedure. The algorithm attempts to approximate with each arc at most once [6].

If the graph has no negative weight arcs, then it can be observed that the optimum values of the vertices are in ascending order along the shortest paths. Consequently, the Dijkstra algorithm traverses the vertices according to this order, and updates the c_v values of all out-neighbors of the current vertex u on the basis of the arc (u, v). It is evident that in this case the order the optimum values of the vertices are determined is unpredictable. Therefore, Dijkstra's algorithm determines this order on the fly (during the algorithm); if the vertex v is the closest (according to the current values of the array C) out-neighbour of the already confirmed shortest-path-tree, then c_v is confirmed as the optimum value of node v. The algorithm attempts to approximate with each arc at most once [6].

The Bellman–Ford algorithm goes through (in arbitrary order) all the arcs of the graph (and attempts to approximate with them) again and again. It needs at most $(N - 1)$ tours. During a last extra-tour the algorithm realizes that all elements of the array C have reached their optimal values. (There were not any updates) [6].


Input

Data of the problem

Recursive formula

Genaral form Formulas of recursion

c[i][j]	=	0	,if	i=0
		0		j=0
		1+c[i-1][j-1]		a[i] = b[j]
		max{c[i-1][j],c[i][j-1]}		a[i]!=b[j]

Size of the problem

Begin Indexes: 0,0

End Indexes: 4,5

Optimum Determination: c[4][5]

Figure 7: The Input-interface.

The algorithm is:

1. Input:
   (a) The recursive formula is introduced.
   (b) The index-limits (along every dimension) of the array C are introduced.
   (c) The indexes of the cell that represents the original problem are introduced.
2. The recursive formula is analyzed:
   (a) The software asks for the input data.
   (b) The digraph is built.
3. The type of the digraph is determined. (A DFS algorithm tests if the graph is acyclic or not, has negative arcs or not, and whether it contains negative cycles or not.)

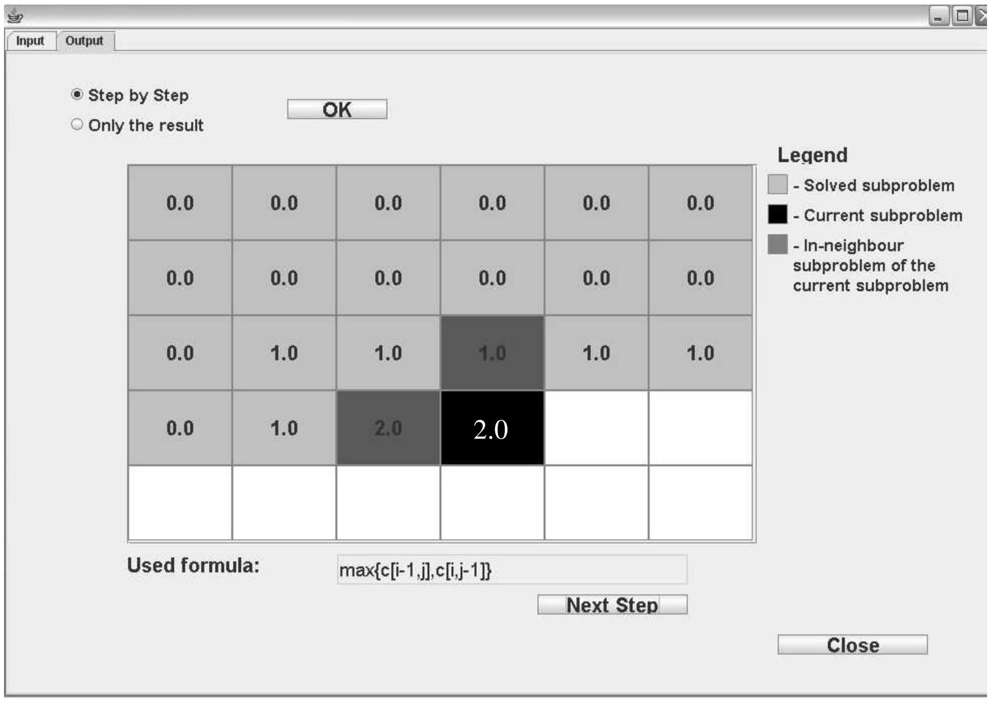
4. The proper optimal path algorithm is applied.

5. The solution (the optimum value corresponding to the original problem, and the cell-indexes along the optimal path) is printed.

Fig. 7 shows the input interface that implements steps 1/a, 1/b and 1/c of the algorithm. As a sample problem we have:

Given two sequences in arrays $\mathtt{a}[1..4]$ and $\mathtt{b}[1..5]$, determine the longest common subsequence.

Fig. 8 shows the output interface that presents a simulation of the dynamic programming solution building process. The optimum value of sub-problem $(3, 3)$ is computed on the basis of the optimum values of sub-problems $(2, 3)$ and $(3, 2)$.



[✓ Step by Step] Only the result

0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0
0.0	1.0	1.0	1.0	1.0	1.0
0.0	1.0	2.0	2.0		

Legend
- ▨ - Solved subproblem
- ■ - Current subproblem
- ▩ - In-neighbour subproblem of the current subproblem

Used formula: max{c[i-1,j],c[i,j-1]}

Figure 8: The Output-interface.

6 Conclusions

The above presented method treats different dynamic programming problems uniformly and in such a way that it makes possible the automation of the programming part of the problem solving process. The software is a very useful tool for all researchers who have to deal with dynamic programming problems, especially for those who are not experts in programming.

Acknowledgement

This research was supported by the Research Programs Institute of Sapientia University.

References

[1] R. Bellman, *Dynamic programming*, Princeton University Press, New Jersey, 1957. ⇒ 149

[2] R. Bellman, S. Dreyfus, *Applied dynamic programming*, Princeton University Press, New Jersey, 1962. ⇒ 149

[3] T. H. Cormen, C. E. Leirserson, R. L. Rivest, C. Stein, *Introduction to algorithms*, The MIT Press, 2003. ⇒ 149, 156, 157

[4] R. Durbin, S. Eddy, A. Krogh, G. Mitchison, *Biological sequence analysis*, Cambridge University Press, 1998. ⇒ 149

[5] Z. Feng, R. Dearden, N Meuleau, R. Washington, Dynamic programming for structured continuous Markov decision problems, *Proceedings of the 20th conference on Uncertainty in artificial intelligence, ACM International Conference Proceeding Series,* AUAI Press, **70**, 2004, pp. 154–161. ⇒ 149

[6] Z. Kátai, The single-source shortest paths algorithms and the dynamic programming, *Teaching Mathematics and Computer Sciences*, **6**, Special Issue (2008) 25–35. ⇒ 157, 160

[7] Z. Kátai, Dynamic programming strategies on the decision tree hidden behind the optimizing problems, *Informatics in Education*, **6**, 1 (2007) 115–138. ⇒ 151, 155

[8] Z. Kátai, Dynamic programming as optimal path problem in weighted digraphs, *Acta Mathematica Academiae Paedagogicae Nyíregyháziensis*, **24**, 2 (2008) 201–208. ⇒ 155, 156

[9] I. King, *A Simple Introduction to Dynamic Programming in Macroeconomic Models*, 2002.
http://researchspace.auckland.ac.nz/handle/2292/190 ⇒ 149

[10] M. Sniedovich, Dijkstra's algorithm revisited: the dynamic programming connexion, *Control and cybernetics*, **35**, 3 (2006) 599–620. ⇒ 159, 160

[11] D. Vagner, Power programming: Dynamic programming, *The Mathematica Journal*, **5**, 4 (1995) 42–51. ⇒ 154

Received: May 5, 2009.