Dynamic programming in package Mathematica

Predrag S. Stanimirović[1,2] , and *Artem* Stupin[2*]

[1] University of Niš, Faculty of Sciences and Mathematics, Višegradska 33, 18000 Niš, Serbia
[2] Laboratory "Hybrid Methods of Modelling and Optimization in Complex Systems", Siberian Federal University, Prosp. Svobodny 79, 660041 Krasnoyarsk, Russia

Abstract. Dynamic programming (DP) is a powerful algorithmic technique for solving optimization problems by breaking them down into simpler subproblems. This paper presents an implementation of DP algorithms for two classic optimization problems: the Knapsack problem and the Traveling Salesman Problem (TSP). The solutions are developed and demonstrated using the Mathematica® programming language. For the Knapsack problem, we present two variants: with and without item repetition. The paper describes the mathematical formulation of each variant and provides detailed Mathematica code for their implementation. Examples are given to illustrate the effectiveness of the algorithms in finding optimal solutions. The TSP implementation demonstrates how DP can be applied to find the shortest Hamiltonian contour in a given network. The paper outlines the mathematical model, recurrent formula, and step-by-step solution process. An example with a four-node network is provided to showcase the algorithm's application. This work highlights the versatility of dynamic programming in solving complex optimization problems and demonstrates the effectiveness of Mathematica as a tool for implementing and visualizing these solutions. The presented algorithms and code snippets serve as valuable resources for researchers and practitioners working on similar optimization challenges.

1 Introduction

In computer science, dynamic programming is a method for solving complex problems by dividing them into collections of smaller, simpler subproblems, solving each of those subproblems only once, and storing their result in memory. The next time we encounter that same subproblem, instead of solving it again, we can get the desired data by simply reading from memory, thus reducing the overall time needed to solve the original problem. Each of the subproblems is indexed using some properties so that later searching for the solution of a certain subproblem, which we have already solved, would be more efficient. The technique of storing solutions to subproblems instead of recomputing them is called "memoization" [1].

Dynamic programming (DP) algorithms are often used for optimization. An algorithm using dynamic programming will examine previously solved subproblems and combine their solutions to obtain the best solution for a given problem. In comparison, the "greedy" algorithm views the solution as a series of specific steps and at each step tries to find a local

* Corresponding author: arstupin@gmail.com

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

optimal solution. Using the "greedy" algorithm does not guarantee us an optimal solution, because searching for local optimal solutions can result in a bad global solution, but it is often a more efficient solution in terms of the time required for calculation. DP is a top-down technique. This technique starts with simplest parts of the problem. Proper combinations of previous solutions lead to solutions of instances with larger dimensions. The procedure is repeated until a solution to the requested problem is generated [1-3].

DP is a very powerful algorithmic strategy that solves problems in the following way:

- a collection of subproblems of the main problem is identified;
- that collection is linearized - subproblems are solved linearly, starting from the smallest to the largest subproblem, our starting problem;
- a formula is identified by which each subproblem is expressed as a function of previous, simpler subproblems

Dynamic programming has its downside. It consists in the fact that this type of programming can be very memory demanding, and we can very easily run out of memory, which would cause a problem if you were working with a program that uses disk space. Memory fragmentation can also occur, and the main disadvantage of DP would be an increase in memory requirements. Likewise, it is very often not straightforward to write code that considers subproblems in the most efficient way.

It is important to note that we can use dynamic programming only for those problems that, when divided into subproblems, would give overlapping subproblems. If by division we get subproblems that do not overlap, then it is preferable to use the divide-and-conquer technique to optimize our algorithm.

In this article we describe solutions of some problems using DP. Implementation of these solutions is developed in the package Mathematica® [4,5]. Mathematica is a fully integrated environment for technical calculations. The emergence of the Mathematica package is often said to mark the beginning of modern technical computing. Mathematica united all these aspects into a coherent system in a unique way. The key idea was the discovery of a symbolic programming language that could manipulate the very large range of objects that appear in technical calculations. Mathematica plays a key role in many important discoveries and is the basis for thousands of scientific papers .

Section 2. is aimed to application of dynamic programming to solving the Knapsack problem. Application of dynamic programming to solving the traveling salesman problem is presented in Section 3. Concluding remarks are given in Section 4.

2 Application of dynamic programming to solving the Knapsack problem

During the robbery, the thief found more valuable things than he expected and must decide what to take. A maximum of W kilograms of something can be placed in his backpack. In total, he found n types of objects, with weights w1,…,wn whose values are v1,…,vn respectively. The problem is to determine the most valuable collection of items he can put in his knapsack [1, 3, 4].

Table 1. Consider for example W = 10 and the following knapsack problem.

Item	Weight	Value
1	6	30$
2	3	14$
3	4	16$
4	2	9$

There are two versions of the problem. If there is an unlimited amount of each item, the optimal choice would be to take item 1 once and item 4 twice, where the total value of the products from the backpack would be $48. On the other hand, if there is one item of each type, then the optimal choice would be items 1 and 3 (total value $46). Using dynamic programming, both types of problems can be solved in O(nW) time complexity, which is rational when W is a small number.

2.1 Knapsack problem with repetition

Let us start with the version which allows repeating items. As always, the main question in DP is, what are subproblems? In this case, we can reduce the original problem in two ways: we can solve the same problems for knapsacks with smaller maximum weights $w \le W$, or we can solve subproblems for a smaller number of items (for example, items 1,2,…,j, for $j \le n$). It usually takes a bit of experimentation to figure out which way is better.

In the first way, we consider knapsacks of smaller capacity than W. Let us define

K(w) = optimal choice of items for backpack capacity w.

Can we represent K(w) in terms of smaller subproblems? If the optimal solution for K(w) includes item i, then by removing that item from the backpack we get the optimal solution for K(w – wi). In other words, K(w) is simply K(w – wi) + vi, for some i. Since we do not know which ones, we will try all possibilities.

$$K(w) = \max_{i: w_i \le w} \{K(w - w_i) + v_i\}$$

where, as usual, the maximum of the empty set is 0.

This algorithm fills a one-dimensional array of length W+1, from left to right. Each element of that array takes O(n) time to calculate, which means the complexity of the algorithm is O(nW). From this algorithm, we can only get information about the highest value of the items that we can put in the backpack. If we want to know the objects that make up that most valuable collection, we have to introduce one more parameter for each of the subproblems, which represents the number of initial objects that we can use, and then we will need a two-dimensional matrix to remember the subproblem.

Mathematica code is as follows.

```
Knapsack[n_, weights_, values_, T1_] := Module[{a = {}, p = {},
   solution = {}, price = 0, T = T1},
  For[i = 1, i <=  T + 1, i++,
   AppendTo[a, 0];
   AppendTo[p, -1];
   ];
  For[i = 1, i <= T + 1, i++,
   For[j = 1, j <= n, j++,
    If[weights[[j]] <= i,
      If[a[[i]] < a[[i - weights[[j]]]] + values[[j]],
        a[[i]] = a[[i - weights[[j]]]] + values[[j]];
        p[[i]] = j;
      ];
     ];
    ];
   ];
  T++;
  While[p[[T]] >= 0,
    AppendTo[solution, p[[T]]];
    price += values[[p[[T]]]];
```

```
    T -= weights[[p[[T]]]];
  ];
  Print["Selected objects: ", solution];
  Print["Optimal value: ", price];
  ];
Example 1.
n=4;
weights={6,3,4,2};
values={30,14,16,9};
T=10;

Knapsack[n,weights,values,T];
Selected objects: {1,4,4}
Optimal value: 48
```

2.2. Knapsack problem without repetition

Now we will analyze the second version of the problem, when we can take one product from each type of item. The previous solution to the problem now becomes useless. For example, it does not mean anything to us if we know the solution to K(w -wi), because we do not know if we have already used item i. That's why we have to look at the whole concept of subproblems differently and remember additional information about the objects that have been used. We will add another parameter, $0 \leq j \leq n$, such that

K(w,j) = optimal selection of items for backpack capacity w using items 1,..., j.

The solution to the problem in this case will be K(W,n).

How, in this case, will we express the subproblem K(w,j) in terms of smaller subproblems? Simple: either object j is needed to achieve the optimal choice, or it is not:

$$K(w,j) = max\{K(w-w_j, j-1) + v_j, K(w, j-1).$$

The algorithm then consists of filling a two-dimensional table, with W + 1 rows and n + 1 columns. Each element of the table takes a constant amount of time to compute, so even though the table is larger than in the previous example, the time complexity remains the same, O(nW).

Mathematica® code is as follows.

```
Knapsack1[n_, weights_, values_, T1_] := Module[{a = {}, p = {},
   solution = {}, price = 0, T = T1},
  a = Table[0, {i, T + 1}, {j, n + 1}];
  p = Table[-1, {i, T + 1}, {j, n + 1}];
  For[i = 2, i <= n + 1, i++,
   For[c = 1, c <= T + 1, c++,
     a[[c, i]] = a[[c, i - 1]];
     p[[c, i]] = p[[c, i - 1]];
     If[weights[[i - 1]] < c,
       If[a[[c,i]] <= a[[c - weights[[i-1]],i-1]] + values[[i-1]],
         a[[c,i]] =  a[[c - weights[[i-1]],i-1]] + values[[i-1]];
         p[[c,i]] = i - 1;
       ];
     ];
   ];
   ];
   i = n + 1;
  T = T + 1;
```

```
While[p[[T, i]] >= 0 && i > 0,
  AppendTo[solution, p[[T, i]]];
  price += values[[p[[T, i]]]];
  i = p[[T, i]];
  T -= weights[[i]];
 ];
Print["Selected objects: ", solution];
Print["Optimal value: ", price];
];
```

Example 2.

```
n=4;
weights={6,3,4,2};
values={30,14,16,9};
T=10;

Knapsack1[n,weights,values,T];
Selected objects:  {3,1}
Optimal value:  46
```

3 Application of dynamic programming to solving the traveling salesman problem

The graph is represented by an ordered pair G=(N,L), where N = {1,…,n} is a set of vertices, and $L \subseteq \{(i,j)|\ i \in N, j \in N\}$ set of edges. If certain numbers or functions are associated with the elements of the vertices N and/or the elements of the set of edges L of the finite graph G = (N,L), such a graph is called a network. The length of each branch (i, j) will be denoted by cij. The label Γ(i) represents the set of nodes that follow node i, i.e. $\Gamma(i) = \{j \in N|\ (i,j) \in L\}$, while the label $\Gamma^{-1}(i)$ represents the set of nodes preceding node i, i.e. $\Gamma^{-1}(i) = \{j \in N|\ (j,i) \in L\}$. The starting and ending nodes are marked with s and t, respectively.

The task of finding the shortest Hamiltonian contour on a given network is called the Traveling Salesman Problem (TSP). A Hamiltonian contour is a path that passes through all nodes of the graph once and only once and ends at the initial node

$$hk = (i_1, i_2, \dots i_{n-1}, i_n, i_1), i_p \neq i_k, \forall p, k\{1, \dots, n\}, p \neq k.$$

Finding the largest Hamiltonian contour resembles the realistic task of a traveling salesman planning a tour of multiple cities, starting from one city and returning to the same one.

The mathematical model of the problem is formed as follows.

Given is a network G = (N,L) with branch lengths cij , for each (i,j) ∈ L, while cij =1 for (i, j) ∈L. Without loss of generality, we can assume that node 1 is the starting and ending node. We will define the set $Q \subseteq N\backslash\{1\}$, as any subset of nodes of the graph that does not contain node 1. We will denote by f(Q, j), j ∉ Q the length of the shortest path starting from node 1, passing through all nodes from the set Q and ending in the node j. By definition of a Hamiltonian contour, the shortest Hamiltonian contour will have length $f(N\backslash\{1\},1), i.e.$ will be equal to the length of the shortest path that starts from the first node, passes through all nodes except the first, and ends at the first node. Based on this, the following recurrent dynamic programming formula can be defined as

$$\forall Q \subseteq N\backslash\{1\})(\forall j \in N\backslash Q)\ f(Q,j) = \min_{i \in Q}\{f(Q\backslash\{i\},i) + c_{ij}\},$$

$$f(\emptyset, j) = c_{1j}.$$

Based on the above, the optimal solution is obtained as a solution to the following problem:

$$f(N\backslash\{1\},1) = \min_{i \in N\backslash\{1\}} \{f(N\backslash\{i,1\},i) + c_{i1}\}.$$

Example 3. Determine the shortest Hamiltonian contour for the network given in Fig. 1.

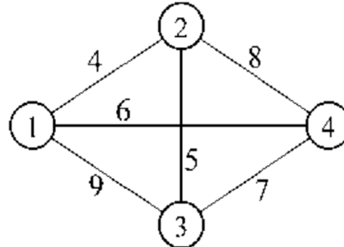

Fig. 1. View of a given network.

Solution. Since the zero stage is trivial, we will start from the first one:
f({2}, 3) = f(∅,2) + c23 = c12 + c23 = 4 + 5 = 9
f({3}, 2) = f(∅, 3) + c32 = c13 + c32 = 9 + 5 = 14
f({2}, 4) = f(∅, 2) + c24 = c12 + c24 = 4 + 8 = 12
f({4}, 2) = f(∅, 4) + c42 = c14 + c42 = 6 + 8 = 14
f({3}, 4) = f(∅, 3) + c34 = c13 + c34 = 9 + 7 = 16
f({4}, 3) = f(∅, 4) + c43 = c14 + c43 = 6 + 7 = 13:
The second stage can be expressed by the following equations:

$$f(\{2,3\},4) = \min_{i \in \{2,3\}} \begin{Bmatrix} f(\{3\},2) + c_{24} \\ f(\{2\},3) + c_{34} \end{Bmatrix} = \min \begin{Bmatrix} 14 + 8 \\ 9 + 7 \end{Bmatrix} = 16$$

$$f(\{2,4\},3) = \min_{i \in \{2,4\}} \begin{Bmatrix} f(\{3\},2) + c_{23} \\ f(\{2\},3) + c_{43} \end{Bmatrix} = \min \begin{Bmatrix} 14 + 5 \\ 12 + 7 \end{Bmatrix} = 19$$

$$f(\{2,3\},4) = \min_{i \in \{3,4\}} \begin{Bmatrix} f(\{4\},3) + c_{32} \\ f(\{3\},4) + c_{42} \end{Bmatrix} = \min \begin{Bmatrix} 13 + 5 \\ 16 + 8 \end{Bmatrix} = 18.$$

And finally, in the third stage we get the final solution, is described by the following equalities:

$$f^* = f(2,3,4,1) = \min_{i \in 2,3,4} f(N\{1,i\},i) + c_{i1}$$

$$= \min \begin{Bmatrix} f(\{3,4\},2) + c_{21} \\ f(\{2,4\},3) + c_{31} \\ f(\{2,3\},4) + c_{41} \end{Bmatrix} = \min \begin{Bmatrix} 18 + 4 \\ 19 + 9 \\ 16 + 6 \end{Bmatrix} = 22.$$

The shortest Hamiltonian contour of the given network is of length 22. The order of the nodes is obtained as follows previously obtained minimum values which are underlined. Two shortest paths are found: (1, 2, 3, 4, 1) and (1, 4, 3, 2, 1).

4 Conclusion

Dynamic programming is a technique for solving optimization problems with a recursive structure. In this article we describe solutions of two optimization problems using DP, such as the Knapsack and the traveling salesman problem. Implementation of these solutions is developed in the programming language Mathematica. Developed computer programs

confirm that the programming language involved in Mathematica is an effective tool for solving such kinds of problems.

This work is supported by the Ministry of Science and Higher Education of the Russian Federation (Grant No. 075-15-2022-1121).

References

1. S. Dasgupta, C. H. Papadimitriou, U. V. Vazirani, *Algorithms* (McGraw-Hill Higher Education, 2006).
2. M. Vugdelija, *Dynamic programming* (Society of Mathematicians of Serbia, Belgrade, 1999).
3. M. Vujošević, *Optimization Methods* (Society of Operations Researchers, Belgrade, 1996).
4. R.E. Maeder, *Computer science with Mathematica: theory and practice for science, mathematics, and engineering* (Cambridge Univrsity Press, 2000).
5. S. Wolfram, *The Mathematica Book* (Wolfram Media, Inc., 2003).
6. D. Podgorelec, B. Žalik, D. Mongus, D. Vlahek, Mathematics, **12(13),** 1987 (2024). https://doi.org/10.3390/math12131987
7. A. Bauer, S. Nakajima, K.-R. Müller, Mathematics, **11(12),** 2628 (2023). https://doi.org/10.3390/math11122628
8. V.R. Kristalinskii, S.N. Chernyi, International Journal of Open Information Technologies **7(2),** 42-48 (2019)