Principles of dynamic programming

Sapienza Università di Roma

Outline

Why not enumerate all the solutions?

Divide, conquer and combine

Problem properties required by DP algorithms

Steps for designing a DP algorithm

The DP algorithms of this module

Typical mistakes

In summary

Learning objectives

At the end of this chapter you will be able to:

- explain why enumerating all the feasible solutions fails, and what a DP algorithm does instead;
- state the divide, conquer and combine principles, and the difference between independent subproblems and overlapping subproblems;
- check the optimal substructure and the overlapping subproblems on a problem;
- list the steps of a DP algorithm: recurrence, base cases, lookup table, and reconstruction of an optimal solution.

Prerequisites

- From the module *Combinatorial Optimization Problems*, the chapter «Combinatorial optimization: definitions»: the combinatorial optimization problems, their feasible solutions, and the fact that their number grows exponentially.
- From the module *Combinatorial Optimization Problems*, the chapter «Knapsack problems»: the 0/1 knapsack problem (KP), the running example of the DP algorithms.
- From the module *Algorithms*: pseudocode (Chapter 2), recursion (Chapter 14), and the growth of the running time of an algorithm (Chapter 10 «Asymptotic growth»).

Why not enumerate all the solutions?

Introduction: why not enumerate all the solutions?

- The simplest exact algorithm lists all the feasible solutions and keeps the best one
- We see how fast its running time grows with the size of the instance
- We compare it with algorithms whose number of steps grows like n and n^2
- This is the reason for a smarter exact method: the DP algorithms of this part of the course

Brute-force algorithms

- A **brute-force algorithm** lists all the feasible solutions of a combinatorial optimization problem.
- Their number grows exponentially: for the KP there are 2^n subsets of items.
- The next table shows how the running time of an algorithm with $\mathcal{O}(n)$, $\mathcal{O}(n^2)$ and $\mathcal{O}(2^n)$ steps grows with the size n of the instance, at $1\,\mu$s per step.

Running times at 1 μs per step

	Size n					
Algorithm	10	20	30	40	50	60
$\mathcal{O}(n)$	.00001 seconds	.00002 seconds	.00003 seconds	.00004 seconds	.00005 seconds	.00006 seconds
$\mathcal{O}(n^2)$	.0001 seconds	.0004 seconds	.0009 seconds	.0016 seconds	.0025 seconds	.0036 seconds
$\mathcal{O}(2^n)$	.001 seconds	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries

- The polynomial algorithms stay below a hundredth of a second.
- The exponential one goes from a millisecond to centuries.

Technology will not help

- Technology will not help.[1]
- With a CPU 1000 times faster, in one hour:
  - an $\mathcal{O}(n)$ algorithm solves an instance 1000 times larger;
  - an $\mathcal{O}(2^n)$ algorithm solves an instance only 10 units larger (since $2^{10} = 1024$).

[1] M. Garey, D. Johnson, *Computers and Intractability: A Guide to NP-Completeness*.

Brute force does not work

Brute-force algorithms do not work:

a DP algorithm avoids the enumeration
by **reusing the optimal values of the subproblems**.

Summary

Brute force:

- It lists all the feasible solutions: 2^n subsets of items for the KP
- With $\mathcal{O}(2^n)$ steps the running time explodes: centuries for $n = 60$
- A CPU 1000 times faster gains only 10 units of size

The way out:

- A DP algorithm reuses the optimal values of the subproblems instead of enumerating

Divide, conquer and combine

Introduction: divide, conquer and combine

- In this section we define what a DP algorithm is
- We see the three principles of the paradigm on which it is built: divide, conquer and combine
- We meet the key idea that makes DP efficient: the optimal value of each subproblem is stored in a lookup table and never recomputed

DP algorithms

Definition 1: DP algorithm

*A **Dynamic Programming (DP) algorithm** is an exact algorithm to find an optimal solution of a class of optimization problems with specific properties (next section). It decomposes the original problem into smaller subproblems of the same type, and it reuses the information computed when solving them.*

Brute force versus DP

- A brute-force algorithm lists all the feasible solutions, and their number grows exponentially (2^n subsets of items for the KP).
- A DP algorithm avoids this enumeration by reusing the optimal values of the subproblems.

How does a DP algorithm decompose a problem?
It follows a classical paradigm: **divide, conquer and combine**.

The divide, conquer and combine paradigm

A DP algorithm exploits the ***divide, conquer, and combine paradigm***, whose main principles are:

1. **Dividing principle** $\rightsquigarrow$ decompose the problem into a number of subproblems that are smaller problems of the same type;
2. **Conquering principle** $\rightsquigarrow$ solve the subproblems in a recursive manner. If the subproblem sizes are small enough just solve the subproblems in a straightforward manner;
3. **Combining principle** $\rightsquigarrow$ combine optimal solutions of the subproblems to obtain an optimal solution for the original problem.

Independent subproblems

- DP algorithms divide the problem into ***independent subproblems***, solve them recursively, and then combine their solutions to solve the original problem.
- Here, **independent** means that each subproblem generated by the division can be solved without affecting the others.

The lookup table

- In a DP algorithm, the same subproblem may be generated several times during the recursive solution process.
- In order to avoid redundant computations, the optimal value of each subproblem is stored in a ***lookup table***.
- Whenever the same subproblem is generated again, its optimal value is retrieved from the lookup table instead of being recomputed.

Summary

DP algorithm:

- An exact algorithm that decomposes a problem into smaller subproblems of the same type and reuses the information computed when solving them

Divide, conquer and combine:

- Divide into smaller subproblems of the same type
- Conquer them recursively (directly when they are small enough)
- Combine their optimal solutions into an optimal solution of the problem

Lookup table:

- The optimal value of each subproblem is computed once and then retrieved

Problem properties required by DP algorithms

Introduction: problem properties required by DP algorithms

- DP does not work for every problem: the recursive decomposition must have two properties
- For each property we see the idea and the points to check in practice
- Checking them is the first thing to do before designing a DP algorithm

Two properties

DP algorithms work only for combinatorial optimization problems that allow a recursive decomposition process respecting the following properties:

1. ***Optimal Substructure***;

2. ***Overlapping Subproblems***.

Optimal Substructure

Optimal Substructure $\rightsquigarrow$ the idea is that, when a problem is decomposed into smaller subproblems, an optimal solution of the problem can be built from optimal solutions of the subproblems.

In practice, to verify this property one usually checks the following points:

- the problem can be decomposed into smaller subproblems of the same type;
- subproblems can be solved independently;
- optimal solutions of the subproblems can be combined to obtain an optimal solution of the original problem;
- a **recurrence relation** to compute the optimal value of the problem can be established.

Overlapping Subproblems

Overlapping Subproblems $\rightsquigarrow$ the idea is that, when a problem is recursively decomposed into smaller subproblems, the same subproblems may be generated several times.

In practice, to verify this property one usually checks the following points:

- the recursive decomposition process generates the same subproblem more than once;
- it is possible to store the optimal values of the subproblems in a lookup table; then whenever the same subproblem is required again, its optimal value can be retrieved from the lookup table instead of being recomputed.

When can DP algorithms be used?

DP algorithms can be used when a problem has both

the Optimal Substructure

and

Overlapping Subproblems.

Summary

Optimal Substructure:

- An optimal solution of the problem is built from optimal solutions of the subproblems
- Check: same type, independent subproblems, combination, recurrence relation

Overlapping Subproblems:

- The recursive decomposition generates the same subproblem several times
- Check: repeated subproblems, optimal values stored in a lookup table

DP needs both properties

Steps for designing a DP algorithm

Introduction: steps for designing a DP algorithm

- We put the principles together into a recipe for designing a DP algorithm
- First the recursive decomposition and the check of the two properties, then two main steps
- Every DP algorithm later in the course follows this recipe

The recursive decomposition process

Identify the ***recursive decomposition process*** that, starting from a given problem, generates its subproblems, which are smaller problems of the same type, and verify the following two properties:

1. the Optimal Substructure property, by checking that an optimal solution of the problem can be obtained by combining the optimal solutions of its subproblems;
2. the Overlapping Subproblems property, by checking whether the recursive decomposition process generates the same subproblems several times.

The two main steps

The construction of a DP algorithm usually follows two main steps:

1. computation of the **optimal value**;
2. construction of an **optimal solution**.

We now look at each step in turn.

Step 1: computation of the optimal value

- define a ***recurrence relation*** that expresses the optimal value of a problem in terms of the optimal values of its subproblems;
- specify the ***base cases***, i.e., the subproblems whose optimal solution is immediate (the optimal value then follows directly from it);
- determine a DP algorithm that computes the optimal values of all relevant subproblems, typically in a ***bottom-up fashion***, storing each value in a lookup table so that it is computed only once and retrieved whenever needed again.

Step 2: construction of an optimal solution

- specify the information that must be additionally stored in order to reconstruct an optimal solution of the problem;
- identify the way to construct an optimal solution from the information stored.

Summary

Before designing:

- Identify the recursive decomposition process and verify the two properties

Step 1, the optimal value:

- Recurrence relation, base cases, bottom-up computation with a lookup table

Step 2, an optimal solution:

- Additional information stored, and the way to build the solution from it

The DP algorithms of this module

Introduction: the DP algorithms of this module

- A map of the DP algorithms that we will see later in the course
- For each problem: the state of the table, what the recurrence runs over, and the number of steps
- We see why these running times are called pseudo-polynomial

Two steps for every algorithm

- Every DP algorithm of this module follows the two steps above: a recurrence with its base cases gives the optimal value, an additional table gives an optimal solution.
- In the next slides: the problems, and for each one the state of the table, what the recurrence runs over, and the number of steps.

The problems

Acronym	Problem
KP	0/1 knapsack problem
UKP	unbounded knapsack problem
BKP	bounded knapsack problem
SSP	subset sum problem
MCKP	multiple-choice knapsack problem
MDKP	multidimensional knapsack problem
KPS	knapsack problem with setups
MKP	multiple knapsack problem
RCP	rod cutting problem
CCP	coin change problem
ULS	uncapacitated lot sizing problem
CLS	capacitated lot sizing problem

States, recurrences and steps (1/2)

Problem	State	Recurrence over
KP	$v(j, k)$	items and capacities
UKP	$v(k)$	capacities
BKP	$v(j, k)$	items and capacities
SSP	$r(j, k)$	items and target values
MCKP	$v(h, k)$	classes and capacities
MDKP	$v(j, k_1, \ldots, k_m)$	items and capacity vectors
KPS	$v(h, k)$, $u(h, i, k)$	classes, items and capacities
MKP	$v(j, k_1, \ldots, k_m)$	items and residual capacities
RCP	$v(\ell)$	rod lengths
CCP	$v(k)$	amounts of change
ULS	$v(t)$	periods
CLS	$v(t, k)$	periods and inventories

States, recurrences and steps (2/2)

Problem	State	Steps
KP	$v(j,k)$	$\Theta(nc)$
UKP	$v(k)$	$\Theta(nc)$
BKP	$v(j,k)$	$\Theta(nc\, b_{\max})$
SSP	$r(j,k)$	$\Theta(nc)$
MCKP	$v(h,k)$	$\Theta(nc)$
MDKP	$v(j,k_1,\ldots,k_m)$	$\Theta(m\,n\prod_d(c_d+1))$
KPS	$v(h,k),\ u(h,i,k)$	$\Theta(nc)$
MKP	$v(j,k_1,\ldots,k_m)$	$\Theta(m\,n\prod_i(c_i+1))$
RCP	$v(\ell)$	$\Theta(n^2)$
CCP	$v(k)$	$\Theta(nc)$
ULS	$v(t)$	$\Theta(m^2)$
CLS	$v(t,k)$	$O(mDC)$

Pseudo-polynomial running times

- In all the numerical examples the data are integers: the number of states is finite, and each state is a table cell.
- The running times $\Theta(nc)$, $\Theta(nc\, b_{\max})$ and $O(mDC)$ contain numbers of the input (a capacity, a bound, a demand), not only its size: they are **pseudo-polynomial**.
- With a vector of capacities (MDKP, MKP) the table has one dimension for each capacity: the number of steps is pseudo-polynomial only for a fixed number m of dimensions or knapsacks.

Summary

The DP algorithms of this module:

- All follow the two steps: recurrence and base cases for the optimal value, an additional table for an optimal solution
- Knapsack problems, rod cutting, coin change and lot sizing

Running times:

- One table cell for each state
- Numbers of the input in the running time: pseudo-polynomial
- Vectors of capacities: pseudo-polynomial only for a fixed m

Typical mistakes

Introduction: typical mistakes

- We list the mistakes that are made most often when using DP
- They concern the two properties, the base cases, and the difference between the optimal value and an optimal solution
- Knowing them helps to design correct DP algorithms in the chapters that follow

Observation 1: Typical mistakes (1/2)

- ***Using DP without the optimal substructure.*** *If an optimal solution of the problem cannot be built from optimal solutions of the subproblems, the recurrence is wrong.*
- ***Using DP without overlapping subproblems.*** *If every subproblem appears once, storing the values in a lookup table gives nothing: the DP algorithm is a plain recursion.*

Observation: Typical mistakes (2/2)

- ***Forgetting the base cases.*** *The recurrence needs the subproblems whose optimal value is immediate, otherwise the recursion does not stop.*
- ***Confusing the optimal value with an optimal solution.*** *The table of the optimal values does not give the solution: an additional table of the decisions is needed.*

Summary

Typical mistakes:

- No optimal substructure: the recurrence is wrong
- No overlapping subproblems: the lookup table gives nothing
- No base cases: the recursion does not stop
- The table of the optimal values is not an optimal solution: store the decisions too

In summary

Introduction: in summary

- We collect the key ideas of the chapter
- They are the common ground of all the DP algorithms that follow in the course

Summary

- A DP algorithm is an exact algorithm that divides a problem into subproblems of the same type, solves them recursively and combines the optimal solutions.
- It works when the problem has the optimal substructure and overlapping subproblems; the optimal values are stored in a lookup table.
- The design has two steps: the optimal value (recurrence, base cases, bottom-up table) and an optimal solution (decisions stored and read backwards).
- Brute force takes exponential time; DP takes a time that is polynomial in the number of states, and pseudo-polynomial when a number of the input is a table dimension.