Dynamic Programming

Sequential Decision Making and Optimization

Joshua Knowles

School of Computer Science

The University of Manchester

COMP60342 - Week 3 2.15, 28 March 2014

Solving problems by solving subproblems

Dynamic programming is a very powerful technique for solving optimization problems.

It is generally an exact method, which gives optimal solutions to problems very efficiently.

It is much more general than the greedy method, yet it can approach the complexity of greedy methods, often giving $O(n^2)$ or $O(n^3)$ methods.

Dynamic programming works by solving subproblems and building up solutions out of the subproblems

Dynamic programming works by solving subproblems and building up solutions out of the subproblems

The building up of solutions can be thought of as solving the problem in ***stages***.

At each stage of the problem, we have a choice of ***actions***, and the best action to take depends on the previous solutions to subproblems already calculated. The crucial information about those subproblem solutions are referred to as ***states***.

We can say that dynamic programming solves ***multi-stage planning*** problems, or problems of ***sequential decision making*** . But in fact it can be adapted to many types of problem.

Let's look at some examples...

Maximum Sum Contiguous Subsequence

Here is an **optimization problem** that can be solved efficiently by dynamic programming.

MAXIMUM SUM CONTIGUOUS SUBSEQUENCE
Instance: A sequence of numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
Output: The contiguous subsequence of maximum sum, where a subsequence of length 0 has sum equal zero.

For example

$$5, 15, -30, 10, -5, 40, 10$$

Solution: $10, -5, 40, 10 = 55$

How could we solve this in general?

Note: The problem follows p177 of Dasgupta et al., *Algorithms*. Mc-Graw-Hill.

A Naive Approach

We don't need to look at every subset of elements, because we are only considering contiguous subsequences.

So we don't have 2^n to look at.

Instead, we can look at only n^2 subsequences, like this

```
bestval:=0
for ( $i$:=1 to $n$)
    for ( $j$:=1 to $n$)
        if (sum of numbers $a_i, a_i + 1, \ldots, a_j > $ bestval)
            update bestval, remember subsequence
output bestval and stored subsequence
```

This has quadratic complexity, but we can do it ***in linear time*** if we use dynamic programming !

The Dynamic Programming Solution

The trick to dynamic programming is to see that optimal solutions to a problem are often **made up of optimal solutions to subproblems**.

This is the case here.

If we find the optimal contiguous subsequence ***ending at position*** j, for $j \in \{1, 2, \ldots, n\}$, then we can always build our next solution out of previous ones.

We can solve for the optimal subsequence ending at position 1 (stage 1) as follows

$$S^*(1) = a_1.$$

$$S^*(1) = a_1.$$

Now to solve the problem of stage 2, we can see that it would be

$$\begin{aligned} S^*(2) &= \max(a_1 + a_2, a_2) \\ &= \max(S^*(1) + a_2, a_2). \end{aligned}$$

We just add the next number from the input sequence to the previous optimal solution, or we use only the next number (i.e., start a new subsequence).

We can continue to do this, so in general we have

$$S^*(j) = \max(S^*(j-1) + a_j, a_j).$$

This is a recursive formula for $S^*(j)$. Afterwards, we need to choose which $S^*(j)$ is the best, including the possibility that it is the null solution . . .

The Dynamic Programming Solution

It looks like this

```
$S^*(0)$:=$0$, *bestval*:=0
for ( $j$:=1 to $n$)
    $S^*(j)$ := $\max(S^*(j-1) + a_j, a_j)$
for ( $j$:=0 to $n$)
    if ( $S^*(j) >$ *bestval* )
        update *bestval*, remember subsequence
output *bestval* and stored subsequence
```

So now it is done in two passes of size n. This is $O(2n) = O(n)$ time.

This is a big improvement over $O(n^2)$.

The Dynamic Programming Solution

We solved the subsequence problem using what is known as

The Bellman Principle*:
The optimal solution to a problem possessing ***optimal substructure*** consists of optimal solutions to each of its subproblems.

The ***Bellman Equation*** is a solution written in a recursive form. It is sometimes referred to as a functional equation or a recurrence.

Let's look at another problem that has optimal substructure and so can be solved using the Bellman principle, and hence dynamic programming.

* Also known as the ***Principle of Optimality***

The Courier's Problem

When should a courier buy and sell his bicycles ***over a five year period***, given the following information?

New Bike Cost:	$500		
Maintenance:	$30 (yr 1)	$40 (yr 2)	$60 (yr 3)
Resale value:	-$400 (after yr 1)	-$300 (after yr 2)	-$250 (yr 3)

He must sell after at most three years

He has no bicycle now, and must have one at all times.

Problem is taken from http://www.sce.carleton.ca/faculty/chinneck/po/Chapter15.pdf

The Courier’s Problem

Formulate the Courier’s Problem as a DP:

Let’s define the ***stages***, ***states***, ***actions***, ***constraints***, ***objective***, ***recurrence*** and approach to solving it.

The Courier's Problem: Components

Stage:	year
State:	years to go
Actions:	how long to keep bicycle
Objective:	minimize spending
Constraints:	must have bicycle, must maintain it, etc
Recurrence:	$g(t) = \min_x\{c_{tx} + g(x)\}$ for $t = 0, 1, 2, 3, 4$.

c_{tx} is the net cost of purchasing bike at year t and keeping it until year x

$g(t)$ is the min cost of operating from year t to year 5.

The approach: start at year 4 and work backwards, remembering solutions to all subproblems. This way, the first subproblem we solve does not depend on any others. Let's see it. . .

The Courier's Problem: Solution

$$\begin{aligned}
g(4) &= c_{45} = 130. \text{ Why?} \\
g(3) &= \min\{c_{35}, c_{34} + g(4)\} \\
&= \min\{270, 130 + 130\} \\
&= 260 \\
g(2) &= \min\{c_{23} + g(3), c_{24} + g(4), c_{25} + g(5)\} \\
&= \min\{130 + 260, 270 + 130, 380 + 0\} \\
&= \min\{390, 400, 380\} \\
&= 380 \\
g(1) &= \min\{c_{12} + g(2), c_{13} + g(3), c_{14} + g(4)\} \\
&= \min\{130 + 380, 270 + 260, 380 + 130\} \\
&= \min\{510, 530, 510\} \\
&= 510 (tie) \qquad \text{Keep second bike 1 year or 3 years} \\
g(0) &= \min\{c_{01} + g(1), c_{02} + g(2), c_{03} + g(3)\} \\
&= \min\{130 + 510, 270 + 380, 380 + 260\} \\
&= \min\{640, 650, 640\} \\
&= 640 (tie) \qquad \text{Keep first bike 1 year or 3 years}
\end{aligned}$$

How long should the courier keep each new bike? 1,1,3; 1,3,1; 3,1,1

Forward versus Backward Recursion

We solved the max sum contiguous subsequence problem by starting at the initial stage and proceeding to the final stage. However, we could have worked backwards.

We solved the Courier's Problem by working from the final stage back to the first stage. But we could have gone the other way.

Depending on which way we go, we may choose to label the stages differently.

There may be an efficiency difference for one choice or another. (It depends if one way reduces the number of subproblems that really need to be solved more than the other. It also depends on whether subproblems are calculated in the right order to allow them to be used without recursion).

But what matters more is not to ***recalculate*** subproblem solutions over and over...

Memoization

Memoization is an important concept in efficiently solving DP problems.

Look back at the solution to the Courier's problem. g(3) is used in the calculation of g(2), g(1) and g(0), so **three times**. And g(3) calls g(4) as well !

To make sure we don't call a function that computes g(3) again each time, we can remember it (memoize it).

Usually this means simply storing the results of subproblem computations in an indexed table or array so that we can easily find them again.

Inventory Problems

Industrial inventory problems are very similar to the Courier's Problem, but on a much larger scale.

- A business sells N different types of items
- There are current stock levels for each item
- Have expected customer orders
- Have storage limitation constraints
- Need to decide how many of each item to make or order at each of several times
- Maximize sales or profit over a period

When customer orders are known precisely, then exactly like Courier Problem.

But usually have expected values or probability distributions. Stochastic problem: Next Lecture

Solving Knapsack Problems with DP

0/1 Knapsack Problem

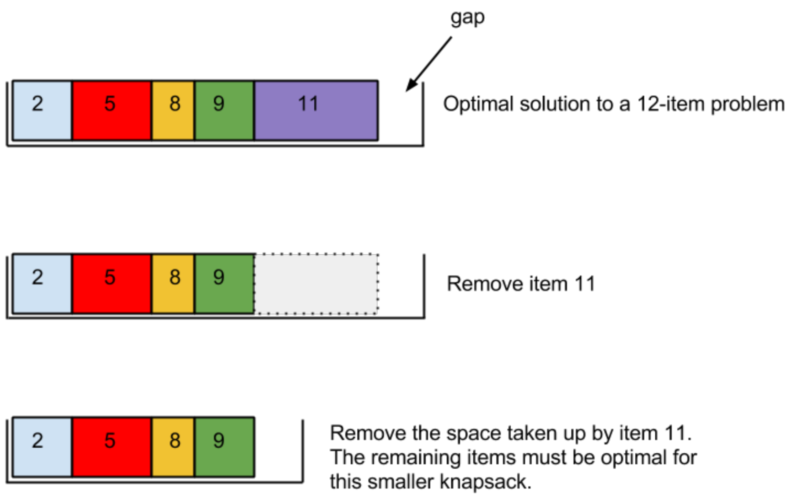
Knapsack problems are ***not*** problems of ***sequential decisions***. They do not normally have *stages*.

But we can approach them as if they were to be done in stages, and apply DP.

This gives a very efficient approach for many instances.

0/1 Knapsack - Tabular Method

The Key Insight How ***optimal substructure*** works: If I remove one item from an optimal solution then the remaining solution must have been optimal for the remaining items and the remaining capacity after removing the space (or weight) taken up by the item removed, as illustrated below.


gap
2
5
8
9
11
Optimal solution to a 12-item problem
2
5
8
9
Remove item 11
2
5
8
9
Remove the space taken up by item 11.
The remaining items must be optimal for
this smaller knapsack.

0/1 Knapsack - Tabular Method

We also have this base case:

⊔ **optimal solution to a 0–capacity problem (for any set of items)**

Given these ideas, we can use ***memoization*** to solve the problem with

states	= the value packed
stages	= the item to pack

Notice: For the states, we only need the value, not the list of items. Why?

0/1 Knapsack - Tabular Method

The Bellman Equation for Knapsack is

$$\begin{aligned} V^*(i,w) &= \max\{V^*(i-1,w), V^*(i-1,w-w_i)+v_i\} \text{ for } w \geq w_i \\ &= V^*(i-1,w) \text{ for } w < w_i, \text{ and} \\ V^*(0,w) &= 0, \text{ for all } w \text{ up to } C, \end{aligned} \tag{1}$$

where $V^*(i,w)$ is the optimal value of the knapsack (sub)problem that considers items 1 to i and a knapsack capacity of w, and w_i and v_i are respectively the weight and value of the ith item.

For you to do.

Convince yourself that the above recurrence works, i.e. it would compute the 0/1 knapsack solution if we call $V^*(n,C)$ where n is number of items and C is knapsack capacity. Be ready to give an explanation.

0/1 Knapsack DP Pseudocode

Idea: Fill up an $(n+1) \times (C+1)$ table.

```
**Algorithm**: DP for Knapsack
**input**: $n, v_1, \ldots, v_n, w_1, \ldots, w_n, C$
**for** $w = 0$ to $C$
    V[$0, w$]=0
**for** $i = 1$ to $n$
    **for** $w = 1$ to $C$
        **if** $w_i > w$
          V[$i, w$] $=$ V[$i-1, w$]
        **else**
          V[$i, w$] $=$ max$\{$ V[$i-1, w$], V[$i-1, w-w_i$]+$v_i$ $\}$
**return** V[$n, C$]
```

0/1 Knapsack: Typical Solution Table

Here is part of a table for a problem with 8 items and $C = 30$

	Subproblem capacity w						
	0	1	2	3	4	5	…	30
$V^*(0,w)$	0	0	0	0	0	0	…	0
$V^*(1,w)$	0	0	0	6	6	6	…	6
$V^*(2,w)$	0	0	5	6	6	11	…	11
$V^*(3,w)$	0	0	5	6	6	11	…	16
⋮					…			
$V^*(8,w)$	0	0	5	6	6	11	…	**47**

The optimal solution to the whole problem is 47 (at bottom right).

We can also infer:

Item 1 has weight 3 and value 6,
Item 2 has weight 2 and value 5,
Item 3 has weight > 1 and value 5.

Typical Approach to Develop Dynamic Programming Computer Solution

1. Formulate the problem into states, stages and decisions
2. Work out a recurrence relation based on the *principle of optimality*
3. Calculate the optimal value that can be achieved by solving the recurrence, making use of memoization
4. **Compute the optimal solution by reconstructing the *decisions* that led to the optimal value.**

Reconstructing the Solution for 0/1 Knapsack

So far our DP method only computes the value of the optimum. We need to record the items that were selected along the way and reconstruct the final solution.

To record items in the solution as well:
Maintain binary array Keep[i, w]
Every time an item is added to the knapsack, we record that *decision* by putting Keep[i, w] $= 1$.
Else if the item is not added, put Keep[i, w] $= 0$.

To reconstruct the solution from the recorded items:

```
$K = C$
for $i = n$ down to 1
   if Keep[$i, K$] $= 1$
      output $i$
      $K = K - w_i$
```

A Nonlinear Integer Knapsack Problem

Real-world knapsack problems are sometimes not 0/1; instead you can choose to pack an ***integer*** number of each item (i.e. 0, 1, 2, ...)

Real-world problems are also sometimes ***nonlinear***. That is, the value of packing two of a particular item may not be 2x the value of packing one (it may be more or less).

Example: A hospital might be faced with the problem of purchasing pieces of equipment so as to maximise the number of patients it can treat. Generally, this will be both integer and non-linear.

A Nonlinear Integer Knapsack Problem

Example: Hospital Purchasing Instance

Decide what equipment a hospital should purchase this year to maximize the number of patients it can treat next year, given this information:

Budget $C = 2.4$ million

Item	Cost	Persons treated per year per item		
		1st item	2nd item	3rd item
X-Ray CT scanner	80,000	1,000	850	750
NMR scanner	400,000	9,000	2,500	2,000
radiation therapy machine	120,000	7,500	4,000	2,700
		first 1-100 items	each item after 100th	
bed	8,000	95		80

Can we solve this using DP? What has to change in the algorithm?

Note: we can purchase 0, 1, 2, or 3 of the scanner/machines (not more). We can purchase any (non-negative) number we like of the hospital beds.

Can we solve this using DP? What has to change from the basic algorithm?

(***You will solve this in lab today !***)

A Nonlinear Integer Knapsack Problem

Bellman is:

$$\begin{aligned} V^*(i,w) &= \max_{0<n_i\le\lfloor\frac{w}{w_i}\rfloor}\{v_i(n_i)+V^*(i-1,w-n_iw_i)\} \\ V^*(0,w) &= 0 \qquad \forall w \end{aligned} \tag{2}$$

where

$V^*(i,w)$	is the required optimum for items 1 to i and capacity w
i	is the item index
w	is the budget, which ranges from 0 to C the knapsack capacity
n_i	is number of items of i packed, and ranges
	from 0 to the number that will fit within w (the current budget)*
$v_i(n_i)$	is the value of packing n_i of item i, and
	is a non-linear ***function*** of n_i

* Note: n_i ranges over 0,1,2,3 for the X-ray, NMR and RT machines in the example problem as at most 3 of each can be purchased.

The Efficiency of Dynamic Programming

Why Can't DP Solve TSP More Efficiently?

We have seen that DP solves many problems efficiently. It does so when we can see how to solve the complete problem by solving a relatively small number of subproblems.

DP works well for some path planning problems, e.g. it can solve shortest path problems in graphs in $O(E + V \log V)$ time. For dense graphs this is $O(V^2)$.

However, for the traveling salesperson problem the running time of DP is $O(n^2 2^n)$. (This is less than $n!$ for naïve enumeration, but is still exponential in n).

Why do you think the DP method does not work better for TSP?

Summary

Dynamic programming can often be used to solve problems of ***sequential decision-making***

But it can still be used even when decisions don't really need to be made in stages

It often yields efficient searches for problems that appear to be intractable, but only if ***optimal substructure*** exists

Applications are numerous: inventory control; knapsack problems; alignment problems; regular expression matching; puzzle-solving;...

Next lecture: Using DP with probabilities