COMS21103

Dynamic programming

Ashley Montanaro
ashley@cs.bris.ac.uk

Department of Computer Science, University of Bristol
Bristol, UK

4 November 2013

Introduction

Dynamic programming is a way of finding efficient algorithms for problems which can be broken down into simpler, overlapping subproblems.

The basic idea

- Start out with a problem you want to solve.
- Find a naïve exponential-time recursive algorithm.
- Speed up the algorithm by storing solutions to subproblems.
- Speed it up further by solving subproblems in a more efficient order.

- Rather than an individual algorithm, dynamic programming is a framework within which many algorithms can be developed.
- Using this framework can require creativity and insight, but can also lead to surprisingly efficient algorithms.

The history

- Dynamic programming was invented by Richard E. Bellman at the RAND Corporation circa 1950.
- The word “programming” is not used in its modern sense but in the sense of finding an optimal schedule or programme of activities.

Bellman explains the name

The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research... His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical... I thought dynamic programming was a good name. It was something not even a Congressman could object to.

Dynamic programming by example

Dynamic programming is a very general technique and the easiest way to understand it is via example. We will discuss three such examples:

- Computing Fibonacci numbers;
- Finding the largest empty square in an image;
- Computing the edit distance between two strings.

Fibonacci numbers

The Fibonacci numbers are defined as follows:

- $F_0 = 0$;
- $F_1 = 1$;
- $F_n = F_{n-1} + F_{n-2} \quad (n \geq 2)$.

Pic: Wikipedia

They occur (for example) in biology. The first few are:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

Calculating the Fibonacci numbers

Imagine we want to calculate the n'th Fibonacci number F_n. The following algorithm is immediate from the definition:

RecFib(n)

```
1. if $n \leq 0$
2.         return 0
3. if $n = 1$
4.         return 1
5. return RecFib($n-1$) + RecFib($n-2$)
```

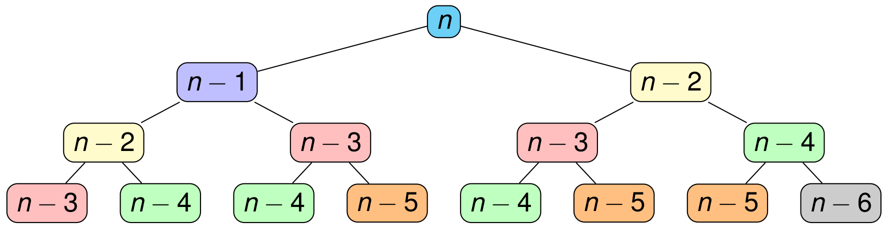
However, RecFib(n) has running time exponential in n.

Exercise: prove this.

RecFib(n)

```
1. if $n \leq 0$
2.         return 0
3. if $n = 1$
4.         return 1
5. return RecFib($n-1$) + RecFib($n-2$)
```

This naïve algorithm is inefficient: it repeatedly recomputes the answers to subproblems.



Improving the algorithm

We can make the algorithm more efficient by storing the results of these recursive calls in an array F.

MemoFib(n)

```
1. if $n \leq 0$
2.         return 0
3. if $n = 1$
4.         return 1
5. if $F[n]$ undefined
6.         $F[n] \leftarrow$ MemoFib($n-1$) + MemoFib($n-2$)
7. return $F[n]$
```

This process is known as memoization (NB: not a typo...).

The performance of this algorithm

MemoFib(n)

```
1. if $n \leq 0$
2.         return 0
3. if $n = 1$
4.         return 1
5. if $F[n]$ undefined
6.         $F[n] \leftarrow$ MemoFib($n-1$) + MemoFib($n-2$)
7. return $F[n]$
```

- Each entry in the memory is only computed once, so there are only $O(n)$ integer additions. All other operations take time $O(1)$.
- Each integer addition can be performed in time $O(n)$, so the total running time is $O(n^2)$.

Improving the algorithm further

Something a bit unnatural about this algorithm: the numbers are requested from the top down, but filled in from the bottom up.

MemoFib(n)

```
1. if $n \leq 0$
2.         return 0
3. if $n = 1$
4.         return 1
5. if $F[n]$ undefined
6.         $F[n] \leftarrow$ MemoFib($n-1$) + MemoFib($n-2$)
7. return $F[n]$
```

- That is, the F array is computed in the order $F[0], F[1], \dots, F[n]$.
- This leads to an unnecessarily large number of recursive calls being made.

Improving the algorithm further

We can get rid of the recursion by simply computing the Fibonacci numbers in ascending order.

AscFib(n)

```
1. $F[0] \leftarrow 0$
2. $F[1] \leftarrow 1$
3. for $i = 2$ to $n$
4.         $F[i] \leftarrow F[i-1] + F[i-2]$
5. return $F[n]$
```

- This algorithm clearly uses $O(n)$ additions and stores $O(n)$ integers.
- This may be the natural algorithm one would come up with when first looking at the problem, but the point is that here we found it almost completely mechanically from the original recurrence.

$F_n$al notes on Fibonacci numbers

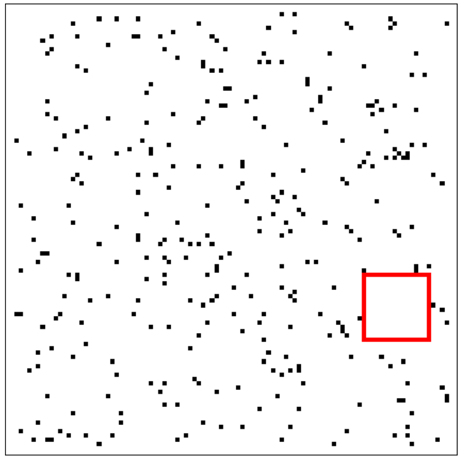
Although this problem was very simple, it illustrates the basic concepts behind dynamic programming:

1. Start out with a problem which can be presented recursively in terms of overlapping subproblems.
2. Write down a naïve recursive algorithm based on this presentation.
3. Memoize the recursive algorithm.
4. Finally, restructure the algorithm to obtain a “bottom-up” algorithm which computes solutions in an efficient order, with no recursion.

Optional exercise: give an improved algorithm which computes F_n using $o(n)$ integer additions.

Example: largest empty square

Consider the following problem: given an $n \times n$ monochrome image, find the largest empty square, i.e. square avoiding any black points.

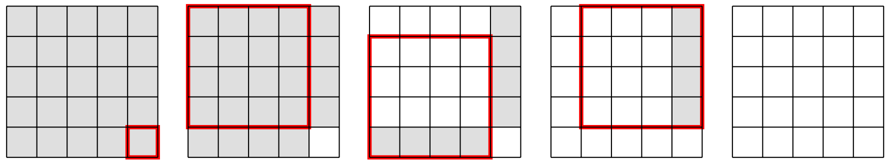


Dynamic programming to the rescue

A recursive formulation of this problem is as follows.

- An $m \times m$ square of pixels S is empty if and only if:
  - The bottom right pixel in S is empty;
  - The three $(m-1) \times (m-1)$ squares in the top left, top right and bottom left of S are all empty.

Proof by picture:



Exercise: Write out this proof in words.

Dynamic programming to the rescue

So let LES(x, y) be the size (i.e. width and height) of the largest empty square whose bottom right-hand corner is at position (x, y).

Then:

- If the pixel (x, y) isn't empty, LES$(x, y) = 0$.
- If (x, y) is empty and in the first row or column, LES$(x, y) = 1$.
- If (x, y) is empty and not in the first row or column, then

$$\text{LES}(x,y) = \min(\text{LES}(x-1,y-1), \text{LES}(x,y-1), \text{LES}(x-1,y)) + 1.$$

This immediately suggests a recursive algorithm!

A recursive algorithm

The following inefficient algorithm computes the size of the largest empty square whose bottom right-hand corner is (x, y).

```
LES($x, y$)
1. if $(x, y)$ not empty
2.         return 0
3. if $x = 1$ or $y = 1$
4.         return 1
5. return min(LES($x-1, y-1$), LES($x, y-1$), LES($x-1, y$)) + 1
```

Once this has been done, taking the maximum of LES(x, y) over all x, y gives the size of the largest empty square in the whole image.

A memoized recursive algorithm

Next step: memoize this algorithm...

MemoLES(x, y)

```
1. if $(x, y)$ not empty
2.         return 0
3. if $x = 1$ or $y = 1$
4.         return 1
5. if LES$[x, y]$ undefined
6.         return min(MemoLES($x - 1, y - 1$), MemoLES($x, y - 1$),
                      MemoLES($x - 1, y$)) $+ 1$
7. return LES$[x, y]$
```

- Each element of the LES array is only computed once.
- So this algorithm now only makes $O(n^2)$ integer additions.

A bottom-up version of the algorithm

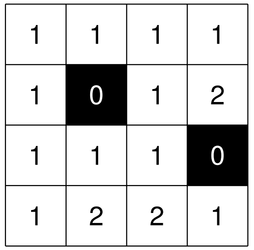
Finally, observe that the LES array gets filled in from the top left. Rewriting this as an iterative algorithm, we get

IterLES(n)

```
1. for $x = 1$ to $n$
2.     for $y = 1$ to $n$
3.         if $(x, y)$ not empty
4.             LES$[x, y]$ = 0
5.         else if $x = 1$ or $y = 1$
6.             LES$[x, y]$ = 1
7.         else
8.             LES$[x, y]$ = min(LES$[x$-1,$y$-1], LES$[x$,$y$-1], LES$[x$-1,$y$]) + 1
```

The runtime is clearly $\Theta(n^2)$.

Example



1	1	1	1
1	0	1	2
1	1	1	0
1	2	2	1

An example monochrome image, filled in with the contents of the corresponding LES array.

Edit distance

- The final problem we discuss is computing the edit distance between two strings s and t.
- The edit distance ED(s,t) is defined as the minimal number of insertions, deletions and replacements of individual characters required to change s into t. Each such operation is called an edit.
- For example, consider the strings `BRISTOL` and `HUSTLE`, with edits

```
BRISTOL
BRISTOLE
~~B~~RISTOLE
HISTOLE
HUSTOLE
HUST~~O~~LE
HUSTLE
```

- So ED(`BRISTOL`, `HUSTLE`) ≤ 5. (In fact, ED(`BRISTOL`, `HUSTLE`) $= 5$.)

Optimal alignment

- An equivalent way of looking at the edit distance problem is as the problem of optimal alignment of strings.
- Given two strings s and t, an alignment of the strings is simply writing one above the other, possibly with gaps ('–') between the letters.
- The cost of an alignment is the number of positions at which the aligned strings differ.
- For example, the following alignment of `BRISTOL` and `HUSTLE` with cost 5 corresponds to the sequence of edits on the previous slide:

```
B  R  I  S  T  O  L  -
-  H  U  S  T  -  L  E
```

Note that in an optimal alignment there is never any need to have two –'s in the same column.

Applications of edit distance

"Traditional" applications include:

- Word processing: identifying the closest dictionary word to a mistyped input
- Plagiarism detection: determining whether two documents are suspiciously close

Computing edit distance has also become a basic task in bioinformatics:

- If s and t are strings of nucleotides representing genes (i.e. something like `AGTCATTC...`), the edit distance can be used to measure how similar the genes are.
- This can be used to determine genes which are expected to have a similar function, or to infer evolutionary relationships between organisms.
- Here insertions, deletions and substitutions all have a biological meaning and may incur different costs.
- Variants include "Smith–Waterman", "Needleman–Wunsch", ...

Writing down a recurrence

- Let s be length n, and t be length m.
- The key to computing ED(s,t) is a recurrence which expresses it in terms of the edit distance of substrings of s and t.
- For any integer k between 1 and n, let $s[k]$ denote the k'th character of s, and let $s[1,\dots,k]$ denote the substring formed from the first k characters of s; define $t[k]$ and $t[1,\dots,k]$ similarly for $1 \le k \le m$.
- Consider any optimal alignment of non-empty strings s and t. The last pair of characters in the alignment must be one of the following three possibilities:

$$\begin{array}{ccc} s[n] & s[n] & - \\ t[m] & - & t[m] \end{array}$$

- We now calculate the cost of each of these possibilities in turn.

Writing down a recurrence

The last pair of characters in the alignment must be one of the following three possibilities:

$$\begin{array}{ccc} s[n] & s[n] & - \\ t[m] & - & t[m] \end{array}$$

In each case in turn, the cost of the optimal alignment of this form is the same as the cost of the optimal alignment of ...

1. the strings $s[1,\dots,n-1]$ and $t[1,\dots,m-1]$, plus 1 (if $s[n] \neq t[m]$).
2. the strings $s[1,\dots,n-1]$ and t, plus 1
3. the strings s and $t[1,\dots,m-1]$, plus 1

So the globally optimal alignment is the minimal cost of all three. This lets us write down a recurrence...

Writing down a recurrence

Assuming that $n \geq 1$, $m \geq 1$, we have the following recurrence:

Recurrence

$$\begin{aligned}\text{ED}(s,t) = \quad \min\{ \; & \text{ED}(s[1,\dots,n-1], t[1,\dots,m-1]) + [s[n] \neq t[m]], \\ & \text{ED}(s[1,\dots,n-1], t) + 1, \\ & \text{ED}(s, t[1,\dots,m-1]) + 1\}\end{aligned}$$

(NB: we use the notation [X] for an expression which evaluates to 1 if X is true, 0 otherwise.)

- If $n = 0$ (i.e. s is empty), then ED(s,t) $= m$; similarly, if $m = 0$ (i.e. t is empty), then ED(s,t) $= n$.
- Now we have the above recurrence, as before we can easily write down a memoized algorithm computing edit distance efficiently.

Writing down an algorithm

The following algorithm computes the edit distance between $s[1,\dots,p]$ and $t[1,\dots,q]$ for arbitrary integers p and q.

MemoED(s,t,p,q)

```
1. if $p = 0$ return $q$
2. if $q = 0$ return $p$
3. if ED[$p$, $q$] undefined
4.         ED[$p$, $q$] $\leftarrow$ min ( MemoED($s$, $t$, $p-1$, $q-1$) + [$s[p] \neq t[q]$],
                                 MemoED($s$, $t$, $p-1$, $q$) + 1,
                                 MemoED($s$, $t$, $p$, $q-1$) + 1)
5. return ED[$p$, $q$]
```

- By the previous discussion, MemoED(s, t, n, m) returns ED(s, t).

A bottom-up algorithm

It suffices to compute edit distances in increasing order of p and q. Therefore, we can write down the following bottom-up algorithm.

IterED(s,t)

```
1. for $q = 0$ to $m$
2.         for $p = 0$ to $n$
3.                 if $p = 0$
4.                         ED[$p$, $q$] $\leftarrow q$
5.                 else if $q = 0$
6.                         ED[$p$, $q$] $\leftarrow p$
7.                 else
8.                         ED[$p$, $q$] $\leftarrow$ min ( ED[$p-1$, $q-1$] + [$s[p] \neq t[q]$],
                                            ED[$p-1$, $q$] + 1,
                                            ED[$p$, $q-1$] + 1)
9. return ED[$n$, $m$]
```

Example

The completed table corresponding to computing the edit distance between `BRISTOL` and `HUSTLE` is:

		B	R	I	S	T	O	L
	0	1	2	3	4	5	6	7
H	1	1	2	3	4	5	6	7
U	2	2	2	3	4	5	6	7
S	3	3	3	3	3	4	5	6
T	4	4	4	4	4	3	4	5
L	5	5	5	5	5	4	4	4
E	6	6	6	6	6	5	5	**5**

So ED($\texttt{BRISTOL}, \texttt{HUSTLE}$) $= 5$.

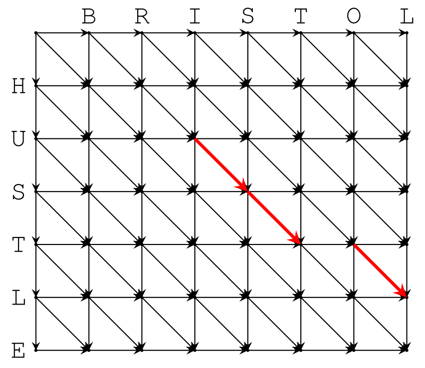
Notes on this algorithm

- The algorithm runs in time $O(mn)$.
- As with many other dynamic programming algorithms, the above algorithm can easily be modified to output an optimal sequence of edits, rather than just the edit distance.
- Step (8) is modified to store which one of the three values compared is smallest. The first corresponds to a replacement, the second an insertion, and the third a deletion.
- This process can also be seen as choosing a path in a certain directed graph...

Edit distance and shortest paths

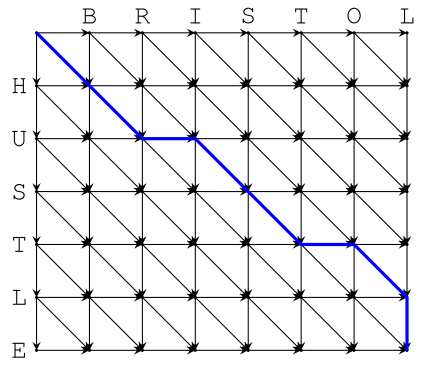
- A graph is created on a grid of $(m+1) \times (n+1)$ vertices where the shortest path from the top left to the bottom right is precisely the edit distance between s and t.
- Each vertex corresponds to a position in the tables filled in by the dynamic programming algorithm.
- All vertices have an edge with weight 1 to adjacent vertices immediately to the right and below
- There are also diagonal edges right and downwards; these have weight 0 wherever the corresponding characters in the strings are the same, and weight 1 otherwise.
- This is a directed acyclic graph, so a shortest path can be found in time $O(mn)$ (see CLRS §24.2).

Example



- In the above diagram, thick red diagonal edges have weight 0, and all other edges have weight 1.

Example



- The thick blue edges give a shortest path with weight 5, corresponding to an optimal sequence of edits.

When does dynamic programming succeed (or fail)?

- The three problems that we have seen seem quite different, but share a feature which makes them good candidates for an efficient dynamic programming algorithm: optimal substructure.
- A problem has optimal substructure if any optimal solution to the problem can be constructed out of optimal solutions to subproblems (which may overlap).
- If this property holds, there is a hope that a dynamic programming algorithm can efficiently construct and combine solutions to the subproblems.
- Dynamic programming is not a panacea: in some cases, a more efficient algorithm can be developed using other techniques that exploit problem structure more closely.

Dynamic vs. greedy and divide-and-conquer

- There is an important distinction between dynamic programming and two other kinds of algorithms you have seen: greedy algorithms and divide-and-conquer.
- A greedy algorithm makes the locally optimal choice at each step; however, this may not be the globally optimal choice.
- A dynamic programming algorithm takes a global view of the input but relies on the problem having the optimal substructure property.
- A divide-and-conquer algorithm also works recursively, by dividing the problem being solved into subproblems; but the subproblems do not overlap.
- Dynamic programming is a more general and powerful technique than either of these.

Summary

- Dynamic programming is a powerful technique for developing efficient algorithms.
- The process of developing such an algorithm can sometimes be almost completely mechanical:
  1. Start out with a problem which can be presented recursively in terms of overlapping subproblems.
  2. Write down a naïve recursive algorithm.
  3. Memoize the recursive algorithm.
  4. Finally, restructure the algorithm to compute solutions in an efficient order.
- However, sometimes applying dynamic programming can require a good deal of thought and creativity!
- Sometimes it can be more efficient to leave the algorithm in its memoized, top-down form. In particular, this can be the case when some subproblems do not need to be computed to solve the problem.

Further reading

- Introduction to Algorithms
  T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein.
  MIT Press/McGraw-Hill, ISBN: 0-262-03293-7.
  - Chapter 15 – Dynamic Programming
- Algorithms
  S. Dasgupta, C. H. Papadimitriou and U. V. Vazirani
  `http://www.cs.berkeley.edu/~vazirani/algorithms.html`
  - Chapter 6 – Dynamic Programming
- Algorithms lecture notes, University of Illinois
  Jeff Erickson
  `http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/`
  - Lecture 5 – Dynamic programming