Algorithms (XI)

Yu Yu

Shanghai Jiaotong University

Review of the Previous Lecture

Chapter 6. Dynamic programming

Longest increasing subsequences

The problem

In the **longest increasing subsequence problem**, the input is a sequence of numbers $a_1, \ldots, a_n$.

A **subsequence** is any subset of these numbers taken in order, of the form

$$a_{i_1}, a_{i_2}, \ldots, a_{i_k}$$

where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$, and an **increasing** subsequence is one in which the numbers are getting strictly larger.

The task is to find the increasing subsequence of *greatest* length.

Graph reformulation

Let's create a graph of all *permissible transitions*: establish a node i for each element a_i, and add directed edges (i, j) whenever it is possible for a_i and a_j to be consecutive elements in an increasing subsequence:

$$i < j \text{ and } a_i < a_j$$

- This graph G = (V,E) is a dag, since all edges (i, j) have $i < j$
- There is a one-to-one correspondence between increasing subsequences and paths in this dag.

Therefore, our goal is simply to find *the longest path in the dag*!

The algorithm

for $j = 1$ **to** n **do**
$\quad L(j) = 1 + \max\{L(i) \mid (i,j) \in E\}$
return $\max_j L(j)$

$L(j)$ is the length of the longest path – *the longest increasing subsequence* – ending at j plus 1.

Any path to node j must pass through one of its predecessors, and therefore $L(j)$ is 1 plus the maximum $L(\cdot)$ value of these predecessors.
If there are no edges into j, we take the maximum over the empty set, zero.

The final answer is the largest $L(j)$, since any ending position is allowed.

This is dynamic programming

In order to solve our original problem, we have defined a collection of subproblems $\{L(j) \mid 1 \leq j \leq n\}$ with the following key property that allows them to be solved in *a single pass*:

> *There is an ordering on the subproblems, and a relation that shows how to solve a subproblem given the answers to "smaller" subproblems, that is, subproblems that appear earlier in the ordering.*

In our case, each subproblem is solved using the relation

$$L(j) = 1 + \max\{L(i) \mid (i,j) \in E\}$$

Edit distance

The problem

When a *spell checker* encounters a possible misspelling, it looks in its dictionary for other words that are *close by*.

What is the appropriate notion of closeness in this case?

A natural measure of the distance between two strings is the extent to which they can be *aligned*, or matched up.
Technically, an alignment is simply a way of writing the strings one above the other.

The **cost** of an alignment is the number of columns in which the letters differ.
And the **edit distance** between two strings is the cost of their best possible alignment.

Edit distance is so named because it can also be thought of as *the minimum number of edits* – **insertions**, **deletions**, and **substitutions of characters** – needed to transform the first string into the second.

A dynamic programming solution

When solving a problem by dynamic programming, the most crucial question is,

What are the subproblems?

Our goal is to find the edit distance between two strings

$$x[1, \ldots, m] \text{ and } y[1, \ldots, n].$$

For every i, j with $1 \leq i \leq m$ and $1 \leq j \leq n$ let

$E(i, j)$: *the edit distance between some prefix of the first string,* $x[1, \ldots, i]$, *and some prefix of the second,* $y[1, \ldots, j]$.

$$E(i, j) = \min \left\{1 + E(i-1, j), 1 + E(i, j-1), \texttt{diff}(i, j) + E(i-1, j-1)\right\}$$

where $\texttt{diff}(i, j)$ is defined to be 0 if $x[i] = y[j]$ and 1 otherwise.

The algorithm

for $i = 0$ **to** m **do**
    $E(i, 0) = i$
for $j = 1$ **to** n **do**
    $E(0, j) = j$
for $i = 1$ **to** m **do**
    for $j = 1$ **to** n **do**
        $E(i, j) = \min \{1 + E(i-1, j), 1 + E(i, j-1),$
        $\texttt{diff}(i, j) + E(i-1, j-1)\}$
return $E(m, n)$

The over running time is $O(m \cdot n)$.

Knapsack

The problem

During a robbery, a burglar finds much more loot than he had expected and has to decide what to take. His bag (or "knapsack") will hold a total weight of at most W pounds. There are n items to pick from, of weight $w_1, \ldots, w_n$ and dollar value $v_1, \ldots, v_n$. What's the most valuable combination of items he can put into his bag?

There are two versions of this problem:

- there are unlimited quantities of each item available;
- there is one of each item.

We shall see in Chapter 8, neither version of this problem is likely to have a polynomial time algorithm.

However, using dynamic programming they can both be solved in $O(n \cdot W)$ time, which is reasonable when W is small, but is not polynomial since the input size is proportional to $\log W$ rather than W.

Knapsack with repetition

For every $w \leq W$ let

$$K(w) = \text{maximum value achievable with a knapsack of capacity } w.$$

We express this in terms of smaller subproblems:

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

$K(0) = 0$
for $w = 1$ **to** W **do**
    $K(w) = \max_{i:w_i \leq w} \{K(w - w_i) + v_i\}$
return $K(W)$

The over running time is $O(n \cdot W)$.

Knapsack without repetition

For every $w \leq W$ and $0 \leq j \leq n$ let

$K(w, j)$ = maximum value achievable with a knapsack of capacity w and items $1, \ldots, j$.

We express this in terms of smaller subproblems:

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

Initialize all $K(0, j) = 0$ and all $K(w, 0) = 0$
for $j = 1$ **to** n **do**
    for $w = 1$ **to** W **do**
        if $w_j > w$ **then** $K(w, j) = K(w, j - 1)$
        $K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$
return $K(W, n)$

The over running time is $O(n \cdot W)$.

Chain matrix multiplication

The problem

Suppose that we want to multiply four matrices, A, B, C, D, of dimensions 50×20, 20×1, 1×10, and 10×100, respectively.

Multiplying an $m \times n$ matrix by an $n \times p$ matrix takes $m \cdot n \cdot p$ multiplications.

Parenthesization	Cost computation	Cost
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	120,200
$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	60,200
$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	7,000

How do we determine the optimal order, if we want to compute $A_1 \times A_2 \times \cdots \times A_n$, where the A_i's are matrices with dimensions $m_0 \times m_1$, $m_1 \times m_2$, $\ldots$, $m_{n-1} \times m_n$, respectively?

Subproblems

For $1 \leq i \leq j \leq n$ let

$$C(i,j) = \text{minimum cost of multiplying } A_i \times A_{i+1} \times \cdots \times A_j$$

$$C(i,j) = \min_{i \leq k < j} \left\{ C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j \right\}.$$

The program

for $i = 1$ **to** n **do** $C(i, j) = 0$
for $s = 1$ **to** $n - 1$ **do**
    for $i = 1$ **to** $n - s$ **do**
        $j = i + \mathrm{s}$
        $C(i, j) = \min_{i \leq k < j} \left\{ C(i, k) + C(k + 1, j) + m_{i-1} \cdot m_k \cdot m_j \right\}$
return $C(1, n)$

The over running time is $O(n^3)$.

The traveling salesman problem

A traveling salesman is getting ready for a big sales tour. Starting at his hometown, he will conduct a journey in which each of his target cities is visited *exactly once* before he returns home. Given the pairwise distances between cities, what is the best order in which to visit them, so as to *minimize the overall distance traveled*?

Denote the cities by $1, \ldots, n$, the salesman's hometown being 1, and let $D = (d_{ij})$ be the matrix of intercity distances. The goal is to design a tour that starts and ends at 1, includes all other cities exactly once, and has minimum total length.

The brute-force approach is to evaluate every possible tour and return the best one. Since there are $(n-1)!$ possibilities, this strategy takes $O(n!)$ time.

The subproblems

For a subset of cities $S \subseteq \{1, 2, \ldots, n\}$ *that includes* 1*, and* $j \in S$*, let* $C(S, j)$ *be the length of the shortest path visiting each node in* S *exactly once, starting at* 1 *and ending at* j*.*

When $|S| > 1$, we define $C(S, 1) = \infty$.

For $j \neq 1$ with $j \in S$ we have

$$C(S, j) = \min_{i \in S : i \neq j} C(S \setminus \{j\}, i) + d_{ij}.$$

The program

$C(\{1,\},1) = 0$
for $s = 2$ **to** n **do**
    for all subsets $S \subseteq \{1,\ldots,n\}$ of size s and containing 1 **do**
        $C(S,1) = \infty$
        for all $j \in S$ and $j \neq 1$ **do**
            $C(S,j) = \min_{i \in S: i \neq j} C(S \setminus \{j\}, i) + d_{ij}$
return $\min_j C(\{1,\ldots,n\},j) + d_{j1}$

There are at most $2^n \cdot n$ subproblems, and each one takes linear time to solve.

The total running time is therefore $O(n^2 \cdot 2^n)$.

Independent sets in trees

The problem

A subset of nodes $S \subseteq V$ is an **independent set** of graph $G = (V, E)$ if there are no edges between them.

Finding the largest independent set in a graph is *believed to be intractable*. However, when the graph happens to be a tree, the problem can be solved in linear time, using dynamic programming.

The subproblems

$I(u) =$ size of largest independent set of subtree hanging from u.

$$I(u) = \max \left\{ 1 + \sum_{\text{grandchildren } w \text{ of } u} I(w), \sum_{\text{children } w \text{ of } u} I(w) \right\}.$$

Chapter 7. Linear programming and reductions

An introduction to linear programming

In a linear programming problem we are given *a set of variables*, and we want to *assign real values to them* so as to

(1) satisfy a set of linear equations and/or linear inequalities involving these variables, and

(2) maximize or minimize a given *linear objective function*.

Example: profit maximization

A boutique chocolatier has two products:

- its flagship assortment of triangular chocolates, called **Pyramide**,
- and the more decadent and deluxe **Pyramide Nuit**.

How much of each should it produce to maximize profits?

- Every box of Pyramide has a a profit of \$1.
- Every box of Nuit has a profit of \$6.
- The daily demand is limited to at most 200 boxes of Pyramide and 300 boxes of Nuit.
- The current workforce can produce a total of at most 400 boxes of chocolate per day.

LP formulation

Objective function	$\max\ x_1 + 6x_2$
Constraints	$x_1 \leq 200$
	$x_2 \leq 300$
	$x_1 + x_2 \leq 400$
	$x_1, x_2 \geq 0$

A linear equation in x_1 and x_2 defines *a line in the two-dimensional (2D) plane*, and a linear inequality designates *a half-space*, the region on one side of the line.

Thus the set of all **feasible solutions** of this linear program, that is, *the points* (x_1, x_2) *which satisfy all constraints*, is the intersection of five half-spaces.

It is a convex polygon.

The optimal solution

We want to find the point in this polygon at which the objective function – the profit – is maximized.

The points with a profit of c dollars lie on the line $x_1 + 6x_2 = c$, which has a **slope** of $-1/6$.

As c increases, this "profit line" moves parallel to itself, up and to the right. Since the goal is to maximize c, we must move the line as far up as possible, while still touching the feasible region.

The optimum solution will be *the very last feasible point* that the profit line sees and must therefore be a vertex of the polygon.

The optimal solution (cont'd)

It is a general rule of linear programs that *the optimum is achieved at a vertex of the feasible region*.

The only exceptions are cases in which there is no optimum; this can happen in two ways:

1. The linear program is **infeasible**; that is, the constraints are so tight that it is impossible to satisfy all of them. For instance,

$$x \leq 1, \qquad x \geq 2.$$

2. The constraints are so loose that the feasible region is **unbounded**, and it is possible to achieve arbitrarily high objective values. For instance,

$$\begin{aligned} \max \ & x_1 + x_2 \\ & x_1, x_2 \geq 0 \end{aligned}$$

Solving linear programs

Linear programs (LPs) can be solved by *the simplex method*, devised by George Dantzig in 1947.

This algorithm starts at a vertex, and repeatedly looks for an *adjacent vertex* (connected by an edge of the feasible region) of better objective value.

In this way it does *hill-climbing* on the vertices of the polygon, walking from neighbor to neighbor so as to steadily increase profit along the way.

Upon reaching a vertex that has no better neighbor, simplex declares it to be optimal and halts.

Why does this local test imply global optimality? By simple geometry – think of the profit line passing through this vertex. Since all the vertex's neighbors lie below the line, the rest of the feasible polygon must also lie below this line.

More products

The chocolatier decides to introduce a third and even more exclusive line of chocolates, called **Pyramide Luxe**. One box of these will bring in a profit of \$13.

Let x_1, x_2, x_3 denote the number of boxes of each chocolate produced daily, with x_3 referring to Luxe.

The old constraints on x_1 and x_2 persist, although the labor restriction now extends to x_3 as well: the sum of all three variables can be at most 400.

What's more, it turns out that Nuit and Luxe require the same packaging machinery, except that Luxe uses it *three times as much*, which imposes another constraint $x_2 + 3x_3 \leq 600$.

LP

$$
\begin{aligned}
\max \;\; & x_1 + 6x_2 + 13x_3 \\
& x_1 \leq 200 \\
& x_2 \leq 300 \\
& x_1 + x_2 + x_3 \leq 400 \\
& x_2 + 3x_3 \leq 600 \\
& x_1, x_2, x_3 \geq 0
\end{aligned}
$$

The space of solutions is now *three-dimensional*.
Each linear equation defines a 3D plane, and each inequality a half-space on one side of the plane. The feasible region is an intersection of seven half-spaces, *a polyhedron*.

A profit of c corresponds to the plane $x_1 + 6x_2 + 13x_3 = c$. As c increases, this profit-plane moves parallel to itself, further and further into the positive **orthant** until it no longer touches the feasible region.
The point of final contact is the optimal vertex: $(0, 300, 100)$, with total profit \$3100.

How would the simplex algorithm behave on this modified problem? A possible trajectory:

$$\underset{\$0}{(0,0,0)} \longrightarrow \underset{\$200}{(200,0,0)} \longrightarrow \underset{\$1400}{(200,200,0)} \longrightarrow \underset{\$2800}{(200,0,200)} \longrightarrow \underset{\$3100}{(0,300,100)}$$

A magic trick called duality

Here is why you should believe that $(0, 300, 100)$, with a total profit of \$3100, is the optimum:

Recall

$$\begin{aligned} \max \ & x_1 + 6x_2 + 13x_3 \\ & x_1 \leq 200 \\ & x_2 \leq 300 \\ & x_1 + x_2 + x_3 \leq 400 \\ & x_2 + 3x_3 \leq 600 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

Add the second inequality to the third, and add to them the fourth multiplied by 4.

The result is the inequality

$$x_1 + 6x_2 + 13x_3 \leq 3100.$$

Example: production planning

The company makes *handwoven carpets*, a product for which the demand is extremely seasonal.

Our analyst has just obtained demand estimates for all months of the next calendar year: $d_1, d_2, \ldots, d_{12}$, ranging from 440 to 920.
We currently have 30 employees, each of whom makes 20 carpets per month and gets a monthly salary of \$2000. We have no initial surplus of carpets.

How can we handle *the fluctuations in demand*? There are three ways:

1. **Overtime**, but this is expensive since overtime pay is 80% more than regular pay. Also, workers can put in at most 30% overtime.
2. **Hiring** and **firing**, but these cost \$320 and \$400, respectively, per worker.
3. **Storing surplus production**, but this costs \$8 per carpet per month. We currently have no stored carpets on hand, and we must end the year without any carpets stored.

LP formulation

w_i	$=$	number of workers during ith month; $w_0 = 30$.
x_i	$=$	number of carpets made during ith month.
o_i	$=$	number of carpets made by overtime in month i.
h_i, f_i	$=$	number of workers hired and fired, respectively, at beginning of month i.
s_i	$=$	number of carpets stored at end of month i; $s_0 = 0$.

First, all variables must be nonnegative:

$$w_i, x_i, o_i, h_i, f_i, s_i \geq 0, i = 1, \ldots, 12.$$

The total number of carpets made per month consists of regular production plus overtime:

$$x_i = 20w_i + o_i$$

(one constraint for each $i = 1, \ldots, 12$).

LP formulation (cont'd)

The number of workers can potentially change at the start of each month:

$$w_i = w_{i-1} + h_i - f_i.$$

The number of carpets stored at the end of each month is what we started with, plus the number we made, minus the demand for the month:

$$s_i = s_{i-1} + x_i - d_i.$$

And overtime is limited:

$$o_i \leq 6w_i.$$

The objective function is to minimize the total cost:

$$\min \quad 2000 \sum_i w_i + 320 \sum_i h_i + 400 \sum_i f_i + 8 \sum_i s_i + 180 \sum_i o_i.$$

Integer linear programming

The optimum solution might turn out to be *fractional*; for instance, it might involve *hiring* 10.6 *workers in the month of March.*

This number would have to be **rounded** to either 10 or 11 in order to make sense, and the overall cost would then increase correspondingly.
In the present example, most of the variables take on fairly large (double-digit) values, and thus rounding is unlikely to affect things too much.

There are other LPs, however, in which rounding decisions have to be made very carefully in order to end up with an integer solution of reasonable quality.

In general, there is a tension in linear programming between *the ease of obtaining fractional solutions* and *the desirability of integer ones*.

As we shall see in Chapter 8, finding the optimum integer solution of an LP is an important but very hard problem, called **integer linear programming**.