Algorithms (X)

Yu Yu

Shanghai Jiaotong University

Review of the Previous Lecture

Set cover

The problem

A county is in its early stages of planning and is deciding *where to put schools*.

There are only two constraints:

- each school should be in a town,
- and no one should have to travel more than 30 miles to reach one of them.

What is the minimum number of schools needed?

This is a typical *set cover* problem. For each town x, let S_x be the set of towns within 30 miles of it. A school at x will essentially "cover" these other towns. The question is then, how many sets S_x must be picked in order to cover all the towns in the county?

Set cover problem

SET COVER

Input: A set of elements B, sets $S_1, \ldots, S_m \subseteq B$

Output: A selection of the S_i whose union is B.

Cost: Number of sets picked.

This problem lends itself immediately to a greedy solution:

> *Repeat until all elements of B are covered: Pick the set S_i with the largest number of uncovered elements.*

The greedy algorithm doesn't always find the best solution!

Performance ratio

Lemma

Suppose B contains n elements and that the optimal cover consists of k sets. Then the greedy algorithm will use at most $k \ln n$ sets.

Proof.

Let n_t be the number of elements still not covered after t iterations of the greedy algorithm (so $n_0 = n$).
Since these remaining elements are covered by the optimal k sets, there must be some set with at least n_t/k of them. Therefore, the greedy strategy will ensure that

$$n_{t+1} \leq n_t - \frac{n_t}{k} = n_t \left(1 - \frac{1}{k}\right),$$

which by repeated application implies

$$n_t \leq n_0 \left(1 - \frac{1}{k}\right)^t.$$

□

Chapter 6. Dynamic programming

Shortest paths in dags, revisited

The special distinguishing feature of a dag is that its nodes can be *linearized*:

they can be arranged on a line so that all edges go from left to right.

If we compute these `dist` values in the left-to-right order, we can always be sure that by the time we get to a node v, we already have all the information we need to compute $\texttt{dist}(v)$.

initialize all $\texttt{dist}(\cdot)$ values to ∞
$\texttt{dist}(s) = 0$
for each $v \in V \setminus \{s\}$, in linearized order **do**
    $\texttt{dist}(v) = \min_{(u,v)\in E}\{\texttt{dist}(u) + \ell(u, v)\}$

This algorithm is solving a collection of *subproblems*, $\{\texttt{dist}(u) \mid u \in V\}$.

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

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

$K(0) = 0$
for $w = 1$ **to** W **do**
    $K(w) = \max_{i:w_i \leq w} \left\{ K(w - w_i) + v_i \right\}$
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 $O(n^3)$.

Shortest paths

Shortest reliable paths

Suppose then that we are given a graph G with lengths on the edges, along with two nodes s and t and *an integer k*, and we want the shortest path from s to t that *uses at most k edges*.

For each vertex v and each integer $i \leq k$, let

$\texttt{dist}(v, i)$ = the length of the shortest path from s to v that uses i edges.

The starting values $\texttt{dist}(v, 0)$ are 1 for all vertices except s, for which it is 0.

$$\texttt{dist}(v, i) = \min_{(u,v) \in E} \{\texttt{dist}(u, i-1) + \ell(u, v)\}.$$

All-pairs shortest paths

What if we want to find the shortest path not just between s and t but *between all pairs of vertices*?

One approach would be to execute our general shortest-path algorithm from Section 4.6.1 (since there may be negative edges) $|V|$ times, once for each starting node. The total running time would then be $O(|V|^2|E|)$.

We'll now see a better alternative, the $O(|V|^3)$ dynamic programming-based **Floyd-Warshall** algorithm.

The subproblems

Number the vertices in V as $\{1, 2, \ldots, n\}$, and let $\texttt{dist}(i, j, k)$ denote the length of the shortest path from i to j in which only nodes $\{1, 2, \ldots, k\}$ can be used as intermediates.

Initially, $\texttt{dist}(i, j, 0)$ is the length of the direct edge between i and j, if it exists, and is ∞ otherwise.

For $k \geq 1$

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

The program

for $i = 1$ **to** n **do**
    for $j = 1$ **to** n **do**
        $\texttt{dist}(i, j, 0) = \infty$
for all $(i, j) \in E$ **do**
    $\texttt{dist}(i, j, 0) = \ell(i, j)$
for $k = 1$ **to** n **do**
    for $i = 1$ **to** n **do**
        for $j = 1$ **to** n **do**
            $\texttt{dist}(i, j, k) = \min\big\{\texttt{dist}(i, j, k-1),$
                $\texttt{dist}(i, k, k-1) + \texttt{dist}(k, j, k-1)\big\}$

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.

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