W4231: Analysis of Algorithms

11/16/99 (revised 11/18/99)

- Dynamic Programming
- NP and NP-completeness

Shortest Path with Budget Constraint

We are given a directed graph $G = (V, E)$, where each edge $(u, v) \in E$ has a positive integer **length** $l_{u,v}$ and a positive integer **cost** $c_{u,v}$.

We are given a positive integer budget B and two vertices $s, t \in V$. We want to find a path from s to t whose total cost is less than B, and whose total length is minimized.

Solution

- Formulate the problem recursively.
- Define the dynamic programming table.
- Give an iterative algorithm to fill the table.
- Explain how to extract a solution from the filled table.

Parse a sequence of characters into words

You are given in input a sentence from which all spaces, commas, etc. have been removed, like in

howdoyouexplainschooltoahigherintelligence

You have also access to a dictionary that tells you whether a given sequence of characters is a word or not.

You want to find a way to break the input into a list of words.

Longest common subsequence

A **subsequence** of a string is obtained by taking a string and possibly deleting elements.
If $x_1 \cdots x_n$ is a string and $1 \le i_1 < i_2 < \cdots < i_k \le n$ is a strictly increasing sequence of indices, then $x_{i_1} x i_2 \cdots x_{i_k}$ is a subsequence of x.

E.g. **art** is a subsequence of **al**go**r**i**t**hm.

Given strings x and y we want to find the longest string that is a subsequence of both.

E.g. **art** is the longest common subsequence of **al**go**r**i**t**hm and p**ar**achu**t**e.

Reducing to a smaller subproblem

The length of the l.c.s. of $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_m$ is either

- The length of the l.c.s. of $x_1 \cdots x_{n-1}$ and $y_1 \cdots y_m$ or;
- The length of the l.c.s. of $x_1 \cdots x_n$ and $y_1 \cdots y_{m-1}$ or;
- $1 +$ the length of the l.c.s. of $x_1 \cdots x_{n-1}$ and $y_1 \cdots y_{m-1}$, if $x_n = y_m$.

Definition of the matrix

For every $0 \le i \le n$ and $0 \le j \le m$, $M[i,j]$ contains the length of the l.c.s. between $x_1 \cdots x_i$ and $y_1 \cdots y_j$.

-$M[i,0] = 0$
-$M[0,j] = 0$
-and

$$M[i,j] = \max\{ M[i-1,j] \quad M[i,j-1] \quad M[i-1,j-1] + eq(x_i,y_j)\}$$

where $eq(x_i,y_j) = 1$ if $x_i = y_j$, $eq(x_i,y_j) = 0$ o/w.

Example

	λ	p	a	r	a	c	h	u	t	e
λ	0	0	0	0	0	0	0	0	0	0
a	0	0	1	1	1	1	1	1	1	1
l	0	0	1	1	1	1	1	1	1	1
g	0	0	1	1	1	1	1	1	1	1
o	0	0	1	1	1	1	1	1	1	1
r	0	0	1	2	2	2	2	2	2	2
i	0	0	1	2	2	2	2	2	2	2
t	0	0	1	2	2	2	2	2	3	3
h	0	0	1	2	2	2	3	3	3	3
m	0	0	1	2	2	2	3	3	3	3

Efficiency

- We have seen several problems that have very efficient algorithms. Even more efficient than one would think possible (median).
- Some (indeed, several) important problems have no known efficient algorithmic solution.
- We would like to *prove* that this is the case because efficient algorithms do *not* exist for such problems (as opposed to because algorithm researchers are not smart enough).

Examples of problems for which we only have exponential-time algorithms

- Find the prime factors of a given integer.
- Solve the Hamiltonian cycle problem.
- Solve the Knapsack problem with exponentially big costs/volumes.
- Hundreds of problems we have not seen, including almost every problem arising in VLSI design, artificial intelligence and operations research.

Usefulness of "negative" results

- Avoid hopeless work.
- Find most appropriate formalization of the problem.
- Explore alternative kind of solution.

Efficiency = Polynomial time

- From now on we will say that an algorithm is "efficient" when it runs in time polynomial in the length of the input.
- A polynomial time algorithm may be inefficient (what about n^{100}?) but an efficient algorithm is almost always polynomial.
- So if we prove that an algorithm does not have polynomial time algorithms we prove for a stronger reason that is has no efficient algorithm.

Proving that problems are hard

For some other problems (e.g. halting problem) we know how to prove that they are unsolvable (regardless of efficiency issues)

For some other problems (e.g. implicit circuit value) that are solvable in exponential time, we know how to prove they do not have polynomial time algorithms.

But for the really interesting problems that we do not know how to solve in polynomial time, we do not know how to prove that exponential time is necessary.

NP-completeness is a theory that gives partial evidence of unsolvability.

NP-completeness

- Thousands of important problems belong to a class of problems called NP-complete problems. Efficient algorithms may or may not exist for them, and we do not know.
- But either *all* of these problems have efficient algorithms or *none* of them does.
- It is often not too difficult, given a new problem which is NP-complete, to prove that it is indeed NP-complete.

Use of the theory

- You are looking for algorithms for a new problem.
- There is a simple brute-force algorithm that runs in exponential time; but despite major effort, no efficient algorithm comes to mind.
- You are able to prove it is NP-complete.
- Then you know that thousands of people have worked decades (millenia!) on essentially the same problem.
- You shift focus.

Decision Problem

To simplify the theory, one only deals with **decision problems**

In a decision problem, the solution for a given input instance is always a YES/NO answer.

Examples:
-Given in input a directed graph, is it acyclic?
-Given in input an undirected graph, does it have a Hamiltonian path?
-Given in input numbers $a_1, \ldots, a_n$ is there are subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} a_i = \sum_{i \notin S} a_i$?

Generality of a theory based on decision problems

Suppose we want to argue about the unsolvability of a certain optimization problem O, even if our theory only deals with decision problems.

We define a decision problem D that is easy if O is easy.
Def. of D: on input an instance x of O, does there exists a solution of cost k or better?

Then we prove that D is unsolvable. Then also O is unsolvable.

Example

We are interested in the following optimization problem: given a graph and a vertex s, find the longest simple path that starts at s.

We define the following decision problem: given a graph, a vertex s and a parameter k, is there a simple path of length at least k starting from s in the graph?

Reductions

Informal Definition: We say that A is *reducible* to B if we can solve A by accessing once a subroutine that solves B.

Consequence: If A is reducible to B, and B has an efficient algorithm, then A has an efficient algorithm.

Additional consequence: If A is reducible to B and A does not have an efficient algorithm, then neither does B.

Formal Definition of Reduction

For a decision problem A, we use the notation $x \in A$ to mean "x is an input instance for which the right solution according to problem A is YES".

We say that A is reducible to B is there is a polynomial time computable function f such that

$$x \in A \text{ if and only if } f(x) \in B$$

Use of a Reduction