BU CLA CS 530: Analysis of Algorithms
Spring 1999

Selected Solutions for Problem Set 12

Problem 1.

We use the *iteration method*, as developed in Section 4.2 in [CLR]. Assuming n is large enough so that we can carry out several iterations and recognize that a pattern has developed:

$$\begin{aligned}
T(n) &= 3T(n/2) + n^2 - n && \text{after 1 iteration} \\
&= 3\big(3T(n/4) + n^2/4 - n/2\big) + n^2 - n \\
&= 9T(n/4) + n^2\big(1 + (3/4)\big) - n\big(1 + (3/2)\big) && \text{after 2 iterations} \\
&= 9\big(3T(n/8) + n^2/16 - n/4\big) + n^2\big(1 + (3/4)\big) - n\big(1 + (3/2)\big) \\
&= 27T(n/8) + n^2\big(1 + (3/4) + (9/16)\big) - n\big(1 + (3/2) + (9/4)\big) && \text{after 3 iterations}
\end{aligned}$$

Therefore, the general pattern after i iterations is:

$$T(n) = 3^i T(n/2^i) + n^2\Big(\sum_{j=0}^{i-1}(3/4)^j\Big) - n\Big(\sum_{j=0}^{i-1}(3/2)^j\Big)$$

By formula (3.3), page 44 in [CLR], we have:

$$\sum_{j=0}^{i-1}(3/4)^j = -4(3/4)^i + 4 \qquad \text{and} \qquad \sum_{j=0}^{i-1}(3/2)^j = 2(3/2)^i - 2$$

Hence, we also have:

$$\begin{aligned}
T(n) &= 3^i T(n/2^i) - 4n^2(3/4)^i + 4n^2 - 2n(3/2)^i + 2n \\
&= 3^{\log n} - 4n^2(3/4)^{\log n} + 4n^2 - 2n(3/2)^{\log n} + 2n \\
&\qquad \text{after posing } i = \log n \text{ and noting that } T(n/2^{\log n}) = T(1) = 1 \\
&= n^{\log 3} - 4n^{\log 3} + 4n^2 - 2n^{\log 3} + 2n \\
&\qquad \text{using the fact that } c^{\log n} = n^{\log c} \text{ for any positive integer } c \\
&= 4n^2 - 5n^{\log 3} + 2n
\end{aligned}$$

Problem 2.

Suppose we already have a procedure NEXT-PERM that generates, one by one, all the permutations of the vertices $V = \{1, \dots, n\}$ by placing them in the same array $P[1\,..\,n]$ each in turn. Specifically, if the array $P[1\,..\,n]$ contains the first permutation, then NEXT-PERM(P) returns the second permutation in the same array $P[1\,..\,n]$ in constant time, NEXT-PERM(NEXT-PERM(P)) returns the third permutation in the same array $P[1\,..\,n]$ in constant time, etc. The total time to generate all $n!$ permutations is therefore $\mathcal{O}(n!)$. Suppose also that NEXT-PERM works on the following assumptions:

1. The first permutation is $P_{\text{first}} = [1, 2, 3, \dots, n-1, n]$.
2. The last permutation is $P_{\text{last}} = [n, n-1, n-2, \dots, 2, 1]$.
3. NEXT-PERM(P_{last}) returns the array $[0, 0, \dots, 0]$ (all 0's).

The desired algorithm PRINT-HAM-CYCLE takes as input the directed graph $G = (V, E)$ where $V = \{1, \dots, n\}$. The test "$(P[i], P[(i+1) \mod n]) \in E$" is *true* or *false* depending on whether $(P[i], P[(i+1) \mod n])$ is an edge of G or not.

```
PRINT-HAM-CYCLE(G)
    P ← [1, 2, 3, ..., n − 1, n]
    repeat  ok ← true
            for i ← 1 to n do
                ok ← ok ∧ (P[i], P[(i + 1) mod n]) ∈ E
            if ok = true then print P
            P ← NEXT-PERM(P)
    until   P = [0, 0, ..., 0]
```

The **repeat-until**-loop is executed $n!$ times. The nested **for**-loop runs in time $\mathcal{O}(n)$. Hence, PRINT-HAM-CYCLE(G) runs in time $\mathcal{O}(n \cdot n!)$. We need to reduce the run time to $\mathcal{O}(n!)$. For this, just observe that we can restrict attention to all permutations beginning with 1, and there are $(n-1)!$ of these. We need to adjust both NEXT-PERM and PRINT-HAM-CYCLE accordingly (not done here).

Problem 3.

The NP-completeness of PARTITION depends on the fact that exponentially large numbers must be created. Indeed, any instance of PARTITION with "small numbers" can be solved in polynomial time. There are two approaches for explaining this, which we outline below.

An instance of PARTITION is a set $S = \{s_1, \dots, s_n\}$ where $n \geqslant 1$ and each s_i is a positive integer. Let $N = s_1 + \cdots + s_n$ and $\ell = \max\{s_1, \dots, s_n\}$. Note that if $s_i \leqslant f(n)$ for every i, then $N \leqslant n \cdot f(n)$; so that if $f(n)$ is a polynomial function, then so is $n \cdot f(n)$.

First Approach: Dynamic Programming (already outlined in lecture). An appropriate dynamic programming algorithm is based on the recurrence

$$f(i, M) = \begin{cases} 0 & \text{if } i = 0 \text{ and } M \neq 0, \\ 1 & \text{if } i = 0 \text{ and } M = 0, \\ \max\{f(i-1, M), f(i-1, M - s_i)\} & \text{otherwise,} \end{cases}$$

where $f(i, M)$ is 1 or 0 depending on whether or not there exists a subset of the first i elements $\{s_1, \dots, s_i\}$ whose sum is M. Given an instance of PARTITION with n elements and a total sum of N, this algorithm produces an answer in $\mathcal{O}(n \cdot N^2)$ time. But the size of the input (counting the number of bits to store n numbers) is $\mathcal{O}(n \cdot \log N)$, which implies this is not a polynomial-time algorithm in general. However, it does run in polynomial-time whenever N is a polynomial function of n. Put differently, for the dynamic programming algorithm to run in exponential time, the input instance of PARTITION must involve numbers of size at least $\Omega(2^n)$. The dynamic programming algorithm thus provides an algorithm that is linear in n, the number of elements in S, but exponential in $\log N$.

Second Approach: Divide-and-Conquer. For any subset $R \subseteq S$, we have

$$\sum_{s \in R} s \leqslant \ell \cdot |R|$$

where $|R|$ is the number of elements in R. Thus, although there are $2^{|R|}$ subsets of R, there are only $1 + \ell \cdot |R|$ weights these subsets can have.

A divide-and-conquer algorithm proceeds as follows. We calculate recursively the set of all possible weights of subsets of $R \subseteq S$. If $R = \{s\}$, the possible weights are 0 and s. If $|R| \geqslant 2$, partition R into two disjoint sets R_1 and R_2 of roughly equal size and recursively produce the data for R_1 and R_2. For each pair of weights w_1 and w_2 in the sets computed for R_1 and R_2, respectively, calculate $w_1 + w_2$ and record it in the set for R. Since w_1 is the weight for some subset of R_1 and w_2 is the weight for some subset of R_2, it follows that $w_1 + w_2$ is the weight of their union, which is a subset of R; moreover, every subset of R is obtained as such as union.

Since the weights of subsets of R_i lie between 0 and $\ell \cdot |R_i|$, the calculation of all the weights $w_1 + w_2$ requires at most $\mathcal{O}(\ell \cdot |R_1| \cdot \ell \cdot |R_2|)$ operations. This gives rise to the recurrence

$$T(n) = 2T(n/2) + \mathcal{O}(\ell^2 n^2)$$

which, using the techniques of Chapter 4 in [CLR], has the solution $T(n) = \mathcal{O}(\ell^2 n^2)$. Thus, if $\ell = \ell(n)$ is a polynomial in n, then $T(n)$ is also a polynomial in n.

Problem 4.

Let $\mathcal{B}$ be a 3-CNF formula, where every clause has exactly 3 distinct literals. We wish to produce an undirected graph $G = (V, E)$, an integer $k \geqslant 1$, and distinguished vertices s_i, t_i, for $1 \leqslant i \leqslant k$, such that there exist k vertex-disjoint paths connecting s_i and t_i for all i iff $\mathcal{B}$ is satisfiable.

Let $\mathcal{C}$ be the set of clauses in $\mathcal{B}$, let X be the set of variables in $\mathcal{B}$, and let L be the set of occurrences of literals in $\mathcal{B}$. Let the set of vertices be

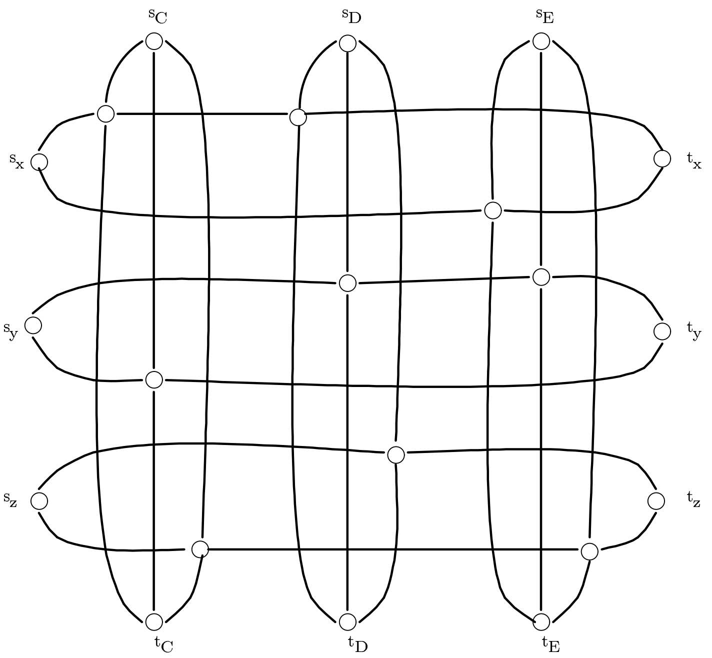
$$V = L \cup \{\ s_C, t_C \mid C \in \mathcal{C}\ \} \cup \{\ s_x, t_x \mid x \in X\ \}$$

and take $k = |\mathcal{C}| + |X|$. For every $C \in \mathcal{C}$, connect all occurrences of literals in C to s_C and t_C. For every $x \in X$, connect all the positive occurrences of x in $\mathcal{B}$ in a single path, connect all the negative occurrences of x in $\mathcal{B}$ in a single path, and connect one endpoint of each of these two paths to s_x and the other to t_x.

For example, consider the CNF formula

$$\mathcal{B} = \underbrace{(x \vee \neg y \vee \neg z)}_{C} \wedge \underbrace{(x \vee y \vee z)}_{D} \wedge \underbrace{(\neg x \vee y \vee \neg z)}_{E}$$

with variables $X = \{x, y, z\}$ and clauses $\mathcal{C} = \{C, D, E\}$. The construction produces the following graph:



If $\mathcal{B}$ is satisfiable, then for each $C \in \mathcal{C}$, take the path from s_C to t_C (of length two) through some true literal of c, and for each $x \in X$, take the path from s_x to t_x through the false literals. These k paths are vertex-disjoint.

Conversely, suppose there are k vertex-disjoint paths. For every $x \in X$, the path from s_x to t_x must go through either all and only the positive occurrences of x, or all and only the negative occurrences of x; any deviation would necessarily go through some s_C or t_C for some $C \in \mathcal{C}$. Assign x *true* if the path from s_x to t_x goes through the negative occurrences of x, and *false* if the path from s_x to t_x goes through the positive occurrences of x. Because the paths are vertex-disjoint, the paths from s_C to t_C, for all $C \in \mathcal{C}$, go through only true literals. Thus the truth assignment satisfies all the clauses.

The preceding establishes the NP-hardness of the problem. The problem is also in NP, and therefore NP-complete, because given a candidate solution (a "certificate") with at most k vertex-disjoint paths, it can be verified in polynomial time.

Assaf Kfoury

Created: 99.04.15
Modified: 99.04.28