11.3 $\mathcal{NP}$-HARD GRAPH PROBLEMS

The strategy we adopt to show that a problem L_2 is $\mathcal{NP}$-hard is:

1. Pick a problem L_1 already known to be $\mathcal{NP}$-hard.

2. Show how to obtain (in polynomial deterministic time) an instance I' of L_2 from any instance I of L_1 such that from the solution of I' we can determine (in polynomial deterministic time) the solution to instance I of L_1 (see Figure 11.3).

3. Conclude from step (2) that $L_1 \propto L_2$.

4. Conclude from steps (1) and (3) and the transitivity of $\propto$ that L_2 is $\mathcal{NP}$-hard.

For the first few proofs we go through all the above steps. Later proofs explicitly deal only with steps (1) and (2). An $\mathcal{NP}$-hard decision problem L_2 can be shown to be $\mathcal{NP}$-complete by exhibiting a polynomial time nondeterministic algorithm for L_2. All the $\mathcal{NP}$-hard decision problems we deal with here are $\mathcal{NP}$-complete. The construction of polynomial time nondeterministic algorithms for these problems is left as an exercise.

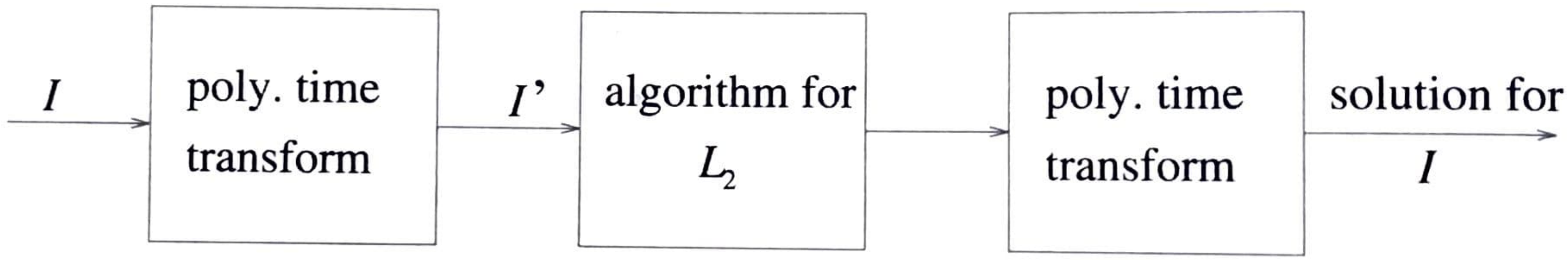

Figure 11.3 Reduction of L_1 to L_2

11.5 $\mathcal{NP}$-HARD CODE GENERATION PROBLEMS

The function of a compiler is to translate programs written in some source language into an equivalent assembly language or machine language program. Thus, the C++ compiler on the Sparc 10 translates C++ programs into the machine language of this machine. We look at the problem of translating arithmetic expressions in a language such as C++ into assembly language code. The translation clearly depends on the particular assembly language (and hence machine) being used. To begin, we assume a very simple machine model. We call this model machine A. This machine has only one register called the *accumulator*. All arithmetic has to be performed in this register. If $\odot$ represents a binary operator such as $+, -, *$, and $/$, then the left operand of $\odot$ must be in the accumulator. For simplicity, we restrict ourselves to these four operators. The discussion easily generalizes to other operators. The relevant assembly language instructions are:

LOAD X	load accumulator with contents of memory location X.
STORE X	store contents of accumulator into memory location X.
OP X	OP may be ADD, SUB, MPY, or DIV.

The instruction OP X computes the operator OP using the contents of the accumulator as the left operand and that of memory location X as the right operand. As an example, consider the arithmetic expression $(a +$

$b)/(c+d)$. Two possible assembly language versions of this expression are given in Figure 11.19. $T1$ and $T2$ are temporary storage areas in memory. In both cases the result is left in the accumulator. Code (a) is two instructions longer than code (b). If each instruction takes the same amount of time, then code (b) will take 25% less time than code (a). For the expressions $(a+b)/(c+d)$ and the given machine A, it is easy to see that code (b) is optimal.

```
LOAD    a        LOAD    c
ADD     b        ADD     d
STORE   T1       STORE   T1
LOAD    c        LOAD    a
ADD     d        ADD     b
STORE   T2       DIV     T1
LOAD    T1
DIV     T2

    (a)              (b)
```

Figure 11.19 Two possible codes for $(a+b)/(c+d)$

Definition 11.7 A *translation* of an expression E into the machine or assembly language of a given machine is *optimal* if and only if it has a minimum number of instructions. □

Definition 11.8 A binary operator $\odot$ is *commutative* in the domain D iff $a \odot b = b \odot a$ for all a and b in D. □

Machine A can be generalized to another machine B. Machine B has $N \geq 1$ registers in which arithmetic can be performed. There are four types of machine instructions for B:

1. LOAD M, R
2. STORE M, R
3. OP $R1, M, R2$
4. OP $R1, R2, R3$

These four instruction types perform the following functions:

1. LOAD M, R places the contents of memory location M into register $R, 1 \leq R \leq N$.

The problem of generating optimal code for level-one dags is $\mathcal{NP}$-hard even when the machine for which code is being generated has only one register. Determining the minimum number of registers needed to evaluate a dag with no STOREs is also $\mathcal{NP}$-hard.

Example 11.22 The optimal codes for the dag of Figure 11.21(b) for one- and two-register machines is given in Figure 11.22.

The minimum number of registers needed to evaluate this dag without any STOREs is two. □

```
LOAD   a,R1              LOAD  a,R1
ADD    R1,b,R1           ADD   R1,b,R1
STORE  T1,R1             ADD   R1,c,R2
ADD    R1,c,R1           MUL   R1,R2,R1
STORE  T2,R1
LOAD   T1,R1
MUL    R1,T2, R1

         (a)                   (b)
```

Figure 11.22 Optimal codes for one- and two-register machines

To prove the above statements, we use the feedback node set (FNS) problem that is shown to be $\mathcal{NP}$-hard in Exercise 5 (Section 11.3).

FNS: Given a directed graph $G = (V, E)$ and an integer k, determine whether there exists a subset V' of vertices $V' \subseteq V$ and $|V'| \leq k$ such that the graph $H = (V - V', E - \{\langle u, v\rangle | u \in V' \text{ or } v \in V'\})$ obtained from G by deleting all vertices in V' and all edges incident to a vertex in V' contains no directed cycles.

We explicitly prove only that generating optimal code is $\mathcal{NP}$-hard. Using the construction of this proof, we can also show that determining the minimum number of registers needed to evaluate a dag with no STOREs is $\mathcal{NP}$-hard as well. The proof assumes that expressions can contain commutative operators and that shared nodes may be computed only once. It is easily extended to allow recomputation of shared nodes. Using an idea due to R. Sethi, the proof is easily extended to the case in which only noncommutative operations are allowed (see Exercise 1).

Theorem 11.14 FNS $\propto$ the optimal code generation for level-one dags on a one-register machine.

11.5.2 Implementing Parallel Assignment Instructions

A *parallel assignment instruction* has the format $(v_1, v_2, \ldots, v_n) = (e_1, e_2, \ldots, e_n)$ where the v_i's are distinct variable names and the e_i's are expressions. The semantics of this statement is that the value of v_i is updated to be the value of the expression e_i, $1 \leq i \leq n$. The value of the expression e_i is to be computed using the values the variables in e_i have before this instruction is executed.

Example 11.24 1. $(A, B) = (B, C)$; is equivalent to $A = B;\ B = C;$.

2. $(A, B) = (B, A)$; is equivalent to $T = A;\ A = B;\ B = T;$.

3. $(A, B) = (A+B, A-B)$; is equivalent to $T1 = A;\ T2 = B;\ A = T1+T2;\ B = T1 - T2;$ and also to $T1 = A; A = A + B;\ B = T1 - B;$. □

As the above example indicates, it may be necessary to store some of the v_i's in temporary locations when executing a parallel assignment. These stores are needed only when some of the v_i's appear in the expressions e_j, $1 \leq j \leq n$. A variable v_i is *referenced* by expression e_i if and only if v_i appears in e_j. It should be clear that only referenced variables need to be copied into temporary locations. Further, parts (2) and (3) of Example 11.24 show that not all referenced variables need to copied.

An implementation of a parallel assignment statement is a sequence of instructions of types $T_j = v_i$ and $v_i = e'_i$, where e'_i is obtained from e_i by replacing all occurrences of a v_i that have already been updated with references to the temporary locations in which the old values of v_i has been saved. Let $R = (\tau(1), \ldots, \tau(n))$ be a permutation of $(1, 2, \ldots, n)$. Then R

is a *realization* of an assignment statement. It specifies the order in which statements of type $v_i = e'_i$ appear in an implementation of a parallel assignment statement. The order is $v_{\tau(1)} = e'_{\tau(1)}, v_{\tau(2)} = e'_{\tau(2)}$, and so on. The implementation also has statements of type $T_j = v_i$ interspersed. Without loss of generality we can assume that the statement $T_j = v_i$ (if it appears in the implementation) immediately precedes the statement $v_i = e'_i$. Hence, a realization completely characterizes an implementation. The minimum number of instructions of type $T_j = v_i$ for any given realization is easy to determine. This number is the cost of the realization. The *cost* $C(R)$ of a realization R is the number of v_i that are referenced by an e_j that corresponds to an instruction $v_j = e'_j$ that appears after the instruction $v_i = e'_i$.

Example 11.25 Consider the statement $(A, B, C) = (D, A + B, A - B)$. The 3! = 6 different realizations and their costs are given in Figure 11.24. The realization 3, 2, 1 corresponding to the implementation $C = A - B; B = A + B; A = D;$ needs no temporary stores ($C(R) = 0$). □

R	$C(R)$
1, 2, 3	2
1, 3, 2	2
2, 1 3	2
2, 3, 1	1
3, 1, 2	1
3, 2, 1	0

Figure 11.24 Realization for Example 11.25

An optimal realization for a parallel assignment statement is one with minimum cost. When the expressions e_i are all variable names or constants, an optimal realization can be found in linear time ($O(n)$). When the e_i are allowed to be expressions with operators then finding an optimal realization is $\mathcal{NP}$-Hard. We prove this statement using the feedback node set problem.

Theorem 11.15 FNS $\propto$ the minimum-cost realization.

Proof: Let $G = (V, E)$ be any n-vertex directed graph. Construct the parallel assignment statement $P : (v_1, v_2, \ldots, v_n) = (e_1, e_2, \ldots, e_n)$, where the v_i's correspond to the n vertices in V and e_i is the expression $v_{i_1} + v_{i_2} + \cdots + v_{i_j}$. The set $\{v_{i_1}, v_{i_2}, \ldots, v_{i_j}\}$ is the set of vertices adjacent from v_i (that is, $\langle v_i, v_{i_l}\rangle \in E(G)$, $1 \le l \le j$). This construction requires at most $O(n^2)$ time.