COMPLETE REGISTER ALLOCATION PROBLEMS

Ravi Sethi
The Pennsylvania State University

Abstract

The search for efficient algorithms for register allocation dates back to the time of the first Fortran compiler for the IBM 704. Since then, many variants of the problem have been considered; depending on two factors: (1) the particular model for registers, and (2) the definition of the term "computation of a program" e.g. whether values may be computed more than once. We will show that several variants of the register allocation problem for straight line programs are polynomial complete. In particular we consider, (1) the case when each value is computed exactly once, and (2) the case when values may be recomputed as necessary. The completeness of the third problem considered is surprising. A straight line program starts with a set of initial values, and computes intermediate and final values. Suppose, for each value, the register that value must be computed into is preassigned. Then, (3) the problem of determining if there is a computation of the straight line program, that computes values into the assigned registers, is polynomial complete.

Keywords and phrases: **register** allocation, program optimization, polynomial complete, straight line program, dag.

1. Introduction

That register allocation is of interest is evident from the number of studies that have considered variants of the problem. While the primary motivation has been the desire to produce decent object code [1-15]; the problem has also been found to occur during the removal of recursion from programs [16]. The relation between aspects of register allocation and aspects of memory allocation has also been noted [17]. Register allocation therefore seems to be an instance of a more general allocation problem.

Since flow of control within a program introduces a level of uncertainty, many of the studies cited have dealt only with straight line programs [3-6,8,9,11-15]. For the class of straight line programs that have no common subexpressions, linear time, optimal allocation algorithms are available [12-14]. We will study register allocation for straight line programs in the context of a set of graph games defined by Walker [15].

Graphical representations of straight line programs are intuitive, and fairly straightforward. Arithmetic expressions have traditionally been represented by trees. If common subexpressions are merged, a tree becomes a directed acyclic graph (dag) [18].

EXAMPLE 1.1: Consider the evaluation of the polynomial $a + bx + cx^2$ using the expression (c*x+b) *x+a (Horner's rule). The dag corresponding to this expression is given by figure 1.2.

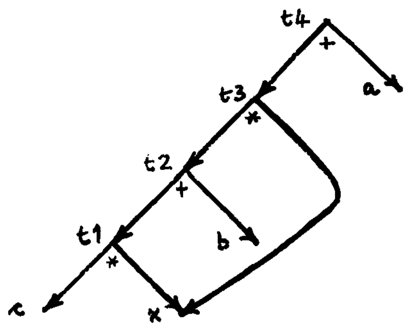


Figure 1.2

Suppose all computation is done in registers. Using three registers, the above dag might be computed as follows:

```
reg 1 ← c
reg 2 ← x
reg 1 ← reg 1 * reg 2
reg 3 ← b
reg 1 ← reg 1 + reg 3
reg 1 ← reg 1 * reg 2
reg 2 ← a
reg 3 ← reg 1 + reg 2
```

However, if only two registers are available, then the following program might be used:

```
reg 1 ← c
reg 2 ← x
reg 1 ← reg 1 * reg 2
reg 2 ← b
reg 1 ← reg 1 + reg 2
reg 2 ← x
reg 1 ← reg 1 * reg 2
reg 2 ← a
reg 1 ← reg 1 + reg 2
```

In the latter program, node x is computed twice. □

For a formal definition of dag, and an algorithm to construct a dag from a straight line program, see [18].

Several problems, like that of determining the chromatic number of a graph, have long defied a nonenumerative solution. A class of such problems, referred to as the class of (polynomial) complete problems has been defined in [19,21]. The class has the important property that if any member of the class can be solved in time a polynomial in some characteristic of the problem, then all members of the class can be solved in polynomial time. More importantly, if any complete problem can be solved in polynomial time, then all languages accepted by nondeterministic Turing machines in polynomial time can be accepted by deterministic Turing machines in polynomial time [19,21]. Background results relating to the term "polynomial complete" may be found in section 2.

Informally, the computation of a dag will be viewed as a game played on the dag. The game assumes that there is an infinite supply of labeled stones, where a stone represents a register. Placing a stone on a node corresponds to computing the node. Thus, a stone may be placed on a non-leaf node x only when there are stones on all directed descendants of x. It will be assumed in section 3 that a node in a dag is computed exactly once. The rules will be generalized in section 4 to permit a node to be recomputed. In each case, it will be shown that given an integer k, the problem of determining if a dag can be computed using no more than k registers, is polynomial complete.

When no node in a dag is recomputed, an allocation for the dag may be viewed as a function from nodes to registers. It should be clear that not all functions from nodes to registers are allocations. For example, it would not do to assign all nodes to the same register. A somewhat less trivial example is given by Figure 1.3. It will be shown in section 5 that the problem of determining if a function from nodes to registers is an allocation, is polynomial complete.

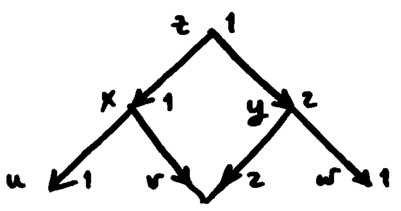


Figure 1.3

In order to appreciate why the last result mentioned is surprising, consider the colouring problem, which may be stated as follows: Given an undirected graph G, a <u>colouring</u> of G is a function from nodes in G to colours, such that no two nodes, joined by an edge in G, may have the same colour. Given an integer k, determine if there is a colouring of G that uses no more than k colours.

While it is known [21], that the colouring problem is polynomial complete, given a function from nodes to colours, it is easy to check if the function is a colouring of the graph. In contrast to checking for an allocation, all that needs to be done is to check that no two nodes joined by an edge are assigned the same colour.

Section 6 considers the implications of the results in sections 3 and 4 to other register allocation problems.

2. Polynomial Completeness

In order to define the class of "polynomial complete problems", a number of concepts that are basic to any discussion in language theory are required. We will not define such terms as <u>deterministic Turing machines</u>, <u>nondeterministic Turing machines</u>, <u>moves</u> made by such machines, and <u>languages accepted</u> by such machines. These definitions may be found, for example, in [20].

Let Σ be some alphabet. Let P be the class of languages over Σ, accepted by polynomial time bounded deterministic Turing machines, and let NP be the class of languages accepted by polynomial time bounded nondeterministic Turing machines. P is clearly a subset of NP. It is not known if P = NP.

Just as languages accepted by Turing machines are defined, it is possible to define the "function computed" by a Turing machine. See [21], for instance.

Let Π be the class of functions from Σ^* into Σ^* computed by polynomial time bounded deterministic Turing machines. Let L and M be languages. L is said to be <u>reducible</u> to M, if there exists a function $f \in \Pi$, such that f(x) is in M if and only if x is in L. L is called <u>(polynomial) complete</u> if L is in NP, and every language in NP is reducible to L. Either all complete languages are in P, or none of them is. The former alternative holds if and only if P = NP [21]."

Demonstrating that all languages in NP are reducible to a given language L is facilitated by a theorem due to Cook [19]. Informally, Cook showed that acceptance of a string in any language in NP is reducible to determining if a formula in the propositional calculus is satisfiable. We will have occasion to deal with this problem in some detail.

DEFINITION: There is a set $\{x_1, x_2, \ldots, x_n\}$

of variables. If x is a variable, then the symbols "x" and "$\bar{x}$" are called literals. x is called a complement of $\bar{x}$, and $\bar{x}$ is called a complement of x. A clause is a subset of the set of literals. A clause $C = \{y_1,y_2,\ldots y_m\}$ will often be represented by "$C = y_1 \vee y_2 \vee \ldots \vee y_m$."

If $C_1,C_2,\ldots,C_m$ are clauses, then C, the conjunction of the clauses, will be represented by "$C_1 \wedge C_2 \wedge \ldots \wedge C_m$". C will also be referred to as an m-clause satisfiability problem over n variables.

C is said to be satisfiable, if there exists a set S which is a subset of the set of literals, such that:

1. S does not contain a pair of complementary literals.
2. $S \cap C_i \neq \phi$, for $i = 1,2,\ldots,m$.

If the set S exists, then a literal y in S will be said to be true, or have value 1, and the complement of the literal will be said to be false, or have value 0. If a literal in a class is true, the clause will be said to be true. □

It is easy to associate a language with the set of satisfiability problems. Following Cook [19], each variable can be represented by some element in Σ, followed by a number in binary notation. Note that there may be an arbitrarily large number of variables. The complement of a variable can be represented, say, by the symbol "$\neg$" followed by the representation of the variable. The other connectives are "$\vee$" and "$\wedge$". When no confusion can occur, the term "satisfiability problem" will be used to refer to the corresponding string, generated as outlined in this paragraph.

THEOREM (Cook): If a language L is in NP, then L is reducible to the set of satisfiability problems.

PROOF: See [19]. □

Just as satisfiability problems were defined, it is possible to define satisfiability problems in with each clause has exactly three literals.

THEOREM (Cook): If a language L is in NP, then L is reducible to the set of satisfiability problems with exactly three literals per clause.

PROOF: Immediate from the result for satisfiability problems with at most three literals per clause [19]. □

The approach in the following sections will be to show that the problem on hand can be associated with a language L in NP, and that the set of satisfiability problems with exactly three literals per clause is reducible to L.

3. Satisfiability to Minimal Allocation

Following Walker [15], the "computation" of a dag will be viewed as a game played on the dag.

GAME 1: Let there be an infinite supply of labelled stones, where the stones may be thought of as registers.

A move in game 1 is one of the following:

1. place a stone on a leaf
2. pick up a stone from a node

if there are stones on every direct descendant of a node x, then:

3. place a stone on x, or
4. move a stone to x from one of the direct descendants of x.

□

DEFINITION 3.1: A computation of a dag is a sequence of moves in game 1, that starts with no stones on any node in the dag, place a stone at most once on any node, and ends with stones on all roots in the dag. □

PROBLEM 1: Given an integer k, does there exist a computation of a dag that uses no more than k registers (nodes may not be recomputed). □

The polynomial completeness of problem 1 will be demonstrated as follows: Given an m-clause satisfiability problem over n variables, with exactly 3 literals per clause, a dag D will be constructed. If the problem is satisfiable, it will be possible to compute D using some number, say k, of registers. If the problem is not satisfiable, then at least $k + 1$ registers will be required to compute D.

D will have $2n$ nodes $x_1,\bar{x}_1,\ldots,x_n,\bar{x}_n$, that correspond to the literals, and m nodes $c_1,c_2,\ldots c_m$, that correspond to the clauses. The first stage of the computation of D will be to compute exactly one of x_k and $\bar{x}_k$, for all k, $1 \leq k \leq n$. This stage may be thought of as "assigning" values to the literals.

EXAMPLE 3.2: Consider the schematic diagram of a dag in Figure 3.3. Circles at nodes mean that once placed, a stone may not be picked up from these nodes. This effect can be achieved by defining a new node called the final node, and making a all circled nodes direct descendants of the final node. Since no nodes may be recomputed, a stone placed on a circled node must remain there until the final node is computed.

Triangles at some of the leaves mean that the computation begins by placing stones on these leaves. This feature can be implemented by defining a new node called the initial node, and making, (a) all leaves with triangles direct descendants of the initial node, and (b) all other nonleaf nodes ancestors of the initial node.

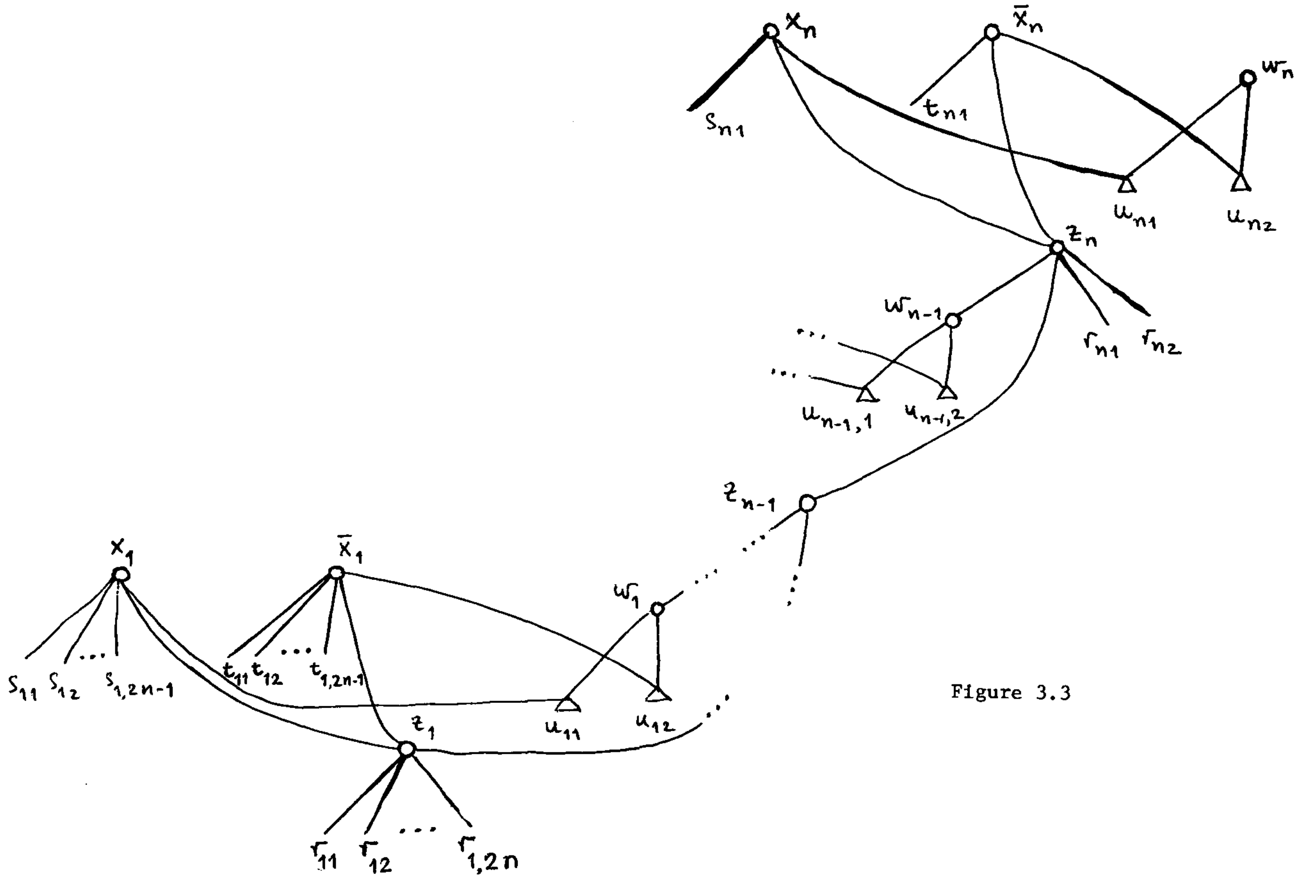


Figure 3.3

Suppose that in addition to the stones on leaves with triangles, there are 2n stones in hand. If any node other than z_1 is now computed we will never be able to compute z_1. Once computed, z_1 will hold a stone, and 2n-1 stones will then be available.

We now have a choice. We may either compute w_1, or one of x_1 and $\bar{x}_1$. Suppose x_1 is computed. One stone will then be held at x_1, leaving 2n-2 stones free. Note that at u_{11} is now free to go to w_1. The 2n-2 free stones cannot be used to compute $\bar{x}_1$, but z_2 can easily be computed. It is easy to see that for all i, $1 \leq i \leq n$, exactlv one of x_i and $\bar{x}_i$ can be computed. Note also that had w_1 not been present, it would have been possible to compute both x_2 and $\bar{x}_2$, by skipping the computation of x_1 or $\bar{x}_1$. □

The second stage computes the nodes $c_1, c_2, \ldots c_m$. If the "assignment" of values to literals is such that each clause is true, then no extra registers will be required. Otherwise an extra register will have to be used. The computation of these m nodes releases enough stones to compute any nodes that are left to be computed.

EXAMPLE 3.4: Consider a clause with two literals y_1 and y_2. (The generalization to three literals is immediate). Figure 3.5 depicts a portion of a day. Nodes y_1, $\bar{y}_1$ and y_2 are in the part of the dag that is not shown. y_1 is

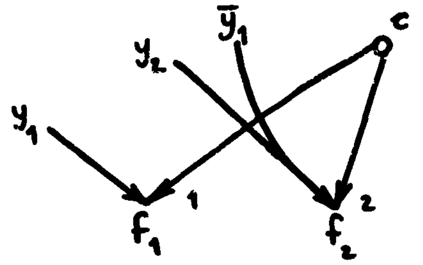


Figure 3.5

a direct ancestor of f_1, $\bar{y}_1$ and y_2 are direct ancestors of f_2. Nodes y_1 and $\bar{y}_1$ are such that y_1 can be computed if and only if $\bar{y}_1$ is not computed. Let stone 1 be on f_1 and stone 2 on f_2.

Since the reduction of the satisfiability problem to problem 1 will be sensitive to the number of stones available, we want to ensure that (1) if either y_1 or y_2 is computed, a stone can be moved to c, and (2) even if both y_1 and y_2 are computed, neither of the stones can be picked up (node c will hold a stone).

Since y_1 is computed if and only if $\overline{y}_1$ is not, it should be evident that the construction in figure 3.5 satisfies both the above criteria. □

The use of the letters c, f, r-u, w-z will be consistent with their use in figures 3.3 and 3.5.

REDUCTION 1: Given an m-clause satisfiability problem over the n variables $x_1,x_2,\ldots,x_n$, where for all i, $1 \leq i \leq m$, clause i has exactly three literals, y_{i1},y_{i2} and y_{i3}, construct a dag D as follows:

Nodes: $A \cup B \cup C \cup F \cup M \cup RST \cup U \cup W \cup X \cup Z$

$A = \{a_j \mid 1 \leq j \leq 2n + 1\}$

$B = \{b_j \mid 1 \leq j \leq 2n - m\}$

$C = \{c_i \mid 1 \leq i \leq m\}$

$F = \{f_{ij} \mid 1 \leq i \leq m, 1 \leq j \leq 3\}$

$M = \{\text{initial, d, final}\}$

$RST = \{r_{kj} \mid 1 \leq k \leq n, 1 \leq j \leq 2n\text{-}2k\text{+}2\} \cup \{s_{kj},t_{kj} \mid 1 \leq k \leq n, 1 \leq j \leq 2n\text{-}2k\text{+}1\}$

$U = \{u_{kj} \mid 1 \leq k \leq n, 1 \leq j \leq 2\}$

$W = \{w_k \mid 1 \leq k \leq n\}$

$X = \{x_k, \overline{x}_k \mid 1 \leq k \leq n\}$

$Z = \{z_k \mid 1 \leq k \leq n\}$

The nodes in the set A,B,F and U will be leaves that are direct descendants of the initial node. Stones will first be placed on these nodes. Nodes in A will have only one direct ancestor - the initial node. The first step will be to move a stone from a node in A to the initial node. The initial node, being a direct descendant of the final node will hold one stone. The 2n stones on the remaining nodes in A can be picked up.

These 2n stones will be used to "assign" values to the literals, as in figure 3.3. Once values have been "assigned" to all the literals, i.e. w_n and z_n have been computed, there will be no stones free. As in figure 3.5, nodes in C will be computed without using any stones in addition to those already on nodes in F, if and only if at least one literal in each clause is true. Node d is computed when all the nodes in C have been computed, releasing the stones on the nodes in B. The set B is used to ensure that, regardless of the value of m, at least 2n-1 stones will be free after d is computed, so that the remaining nodes in X can be computed.

Edges $E1 \cup E2 \cup \ldots \cup E10$

$E1 = \{(\text{initial},g) \mid g \in A \cup B \cup F \cup U\}$

$E2 = \{(g,\text{initial}) \mid g \in C \cup RST \cup W\}$

$E3 = \{(\text{final},g) \mid g \in W \cup X \cup Z \cup \{\text{initial,d}\}\}$

$E4 = \{(x_k,z_k), (\overline{x}_k,z_k), (x_k,u_{k1}) (\overline{x}_k,u_{k2}) \mid 1 \leq k \leq n\}$

$E5 = \{(w_k,u_{kj}) \mid 1 \leq k \leq n, 1 \leq j \leq 2\}$

$E6 = \{(x_k,s_{kj}), (\overline{x}_k,t_{kj}) \mid 1 \leq k \leq n, 1 \leq j \leq 2n\text{-}2k\text{+}1\} \cup \{(z_k,r_{kj}) \; 1 \leq k \leq n, 1 \leq j \leq 2n\text{-}2k\text{+}2\}$

$E7 = \{(z_k,w_{k-1}), (z_k,z_{k-1}) \mid 2 \leq k \leq n\} \cup \{(c_i,w_n), (c_i,z_n) \mid 1 \leq i \leq m\}$

$E8 = \{(c_i,f_{ij}) \mid 1 \leq i \leq m, 1 \leq j \leq 3\}$

$E9 = \{(d,g) \mid g \in B \cup C\}$

For all i, $1 \leq i \leq m$, clause i consists of the literals y_{i1},y_{i2} and y_{i3}. For all j, $1 \leq j \leq 3$, since y_{ij} is a literal, there exists a k, $1 \leq k \leq n$, such that y_{ij} is either x_k or $\overline{x}_k$. For the definition of the set E10, if $y_{ij} = x_k$, we use the symbol "y_{ij}" to refer to node x_k, and "$\overline{y}_{ij}$" to refer to node $\overline{x}_k$. Otherwise, if $y_{ij} = \overline{x}_k$, we use the symbol "y_{ij}" to refer to $\overline{x}_k$ and "$\overline{y}_{ij}$" to refer to x_k.

$E10 = \{(y_{ij},f_{ij}), (\overline{y}_{ij},f_{ik}) \mid 1 \leq i \leq m, 1 \leq j \leq 3, j + 1 \leq k \leq 3\}$ □

DEFINITION: Given a set S, let #S give the number of elements in S.

DEFINITION: Let the term <u>(m-3,n) satisfiability problem</u> be used to refer to an m-clause satisfiability problem over n variables, with exactly three literals per clause.

LEMMA 3.6: Let D be the dag created by reduction 1 for an (m-3,n) satisfiability problem. If the problem is satisfiable, then D can be computed using 3m + 4n + 1 + #B registers.

PROOF: Let q = 3m + 4n + 1 + #B. We will give an algorithm to compute D using q stones. Let the stones be numbered 1,2,...,q.

1. D has q leaves, given by the sets A,B,F, and U. Place q stones on the leaves of D as follows: stones 1,2,...3m on nodes in F; stones 3m + 1,...,3m + #B on the elements of B; stones 3m + #B + 1,...,3m + #B + 2n on the elements of U; the remaining 2n + 1 stones on the elements of A.

2. Let p = 3m + 2n + 1 + #B. Note that stone p is on an element of A, and that the initial node is the only direct ancestor of the nodes in A. Move stone p up to the initial node, and pick up stones p+1,p+2,...,p+2n.

3. Do 4 for k = 1,2,...,n. (See figure 3.3).

4. Place stone $p+2k-1,\dots,p+2n$ on nodes r_{kj}, $1 \le j \le 2n-2k+2$. Move stone $p+2k-1$ up to z_k, and pick up stones $p+2k,\dots,p+2n$. If the literal x_k is true for the problem to be satisfiable, then place stone $p+2k,\dots,p+2n$ on nodes s_{kj}, $1 \le j \le 2n - 2k+1$. Move stone $p + 2k$ up to node x_k, and pick up the stones $p+2k+1,\dots,p+2n$. Move the stone at u_{k1} to w_k.

If the literal x_k is false, then place stones $p+2k,\dots,p+2n$ on nodes t_{kj}, $1 \le j \le 2n - 2k + 1$. Move stone $p + 2k$ up to node $\bar{x}_k$, and pick up stones $p+2k+1,\dots,p+2n$. Move the stone at u_{k2} to w_k.

If the value of x_k is undefined, then proceed as if x_k were true.

5. Do 6 for $i = 1,2,\dots,m$. (see figure 3.5)

6. For clause i, $y_{i1} \vee y_{i2} \vee y_{i3}$, if y_{i1} is true, then move the stone which is at f_{i1}, up c_i. Otherwise, if y_{i2} is true, then move the stone at f_{i2} to c_i. If y_{i2} is also false, then the conjunction of the clauses being satisfiable, y_{i3} must be true. So move the stone at f_{i3} to c_i.

7. Move the stone at c_1 up to d. Pick up the stones at $c_2,c_3,\dots c_m$, and the stones on the elements of B. By construction, $m - 1 + \#B \ge 2n - 1$.

8. Use the $2n-1$ stones in hand to compute the nodes that remain to be computed.

It is easy to verify that the above algorithm does indeed compute D.

□

LEMMA 3.7: Let D be the dag constructed by reduction 1 for an $(m-3,n)$ satisfiability problem. Let D be computed using $3m + 4n + 1 + \#B$ registers. Then, for all k, $1 \le k \le n$, just after z_k is computed,

(a) for all j, $1 \le j \le k - 1$, at most one of x_j and $\bar{x}_j$ has been computed.

(b) $2n - 2k + 1$ stones are free.

PROOF: Since the initial node is a direct ancestor of all the leaves, and a descendant of all other nonleaves, the first moves in the computation must be to place stones on all the leaves.

Since all elements of the set C are direct ancestors of z_n, just after z_n is computed, none of the elements of C can have been computed. Therefore stones on nodes in the set F cannot be free. Node d, being an ancestor of the nodes in C cannot have been computed, so stones on nodes in the set B cannot be free. Moreover, the initial node, being a direct descendant of the final node will hold a stone. Therefore $3m + 1 + \#B$ stones cannot be free.

basis: $k = 1$. Node z_1 has $2n$ direct descendants, $r_{11},r_{12},\dots,r_{1,2n}$. Since z_1 has just been computed, at least $2n - 1$ stones are free.

Since all nodes in the set X are ancestors of z_1, all nodes in X have yet to be computed. Hence, stones on nodes in U cannot be free. That leaves $2n$ stones, one of which is at z_1. Hence, exactly $2n - 1$ stones are free.

inductive step: Assume the lemma is true for all smaller values, and consider z_{k+1}.

From the inductive hypothesis, $2n - 2k + 1$ stones are free just after z_k is computed. Note that since $2n - 2k + 1$ stones are free, none of the elements of the set $\{x_1,\bar{x}_1,\dots,x_{k-1},\bar{x}_{k-1}\}$ that remain to be computed can be touched. $2n - 2j + 1$ stones are required to compute either x_j or $\bar{x}_j$. For $j < k$, $2n - 2j + 1 > 2n - 2k + 1$.

By accounting for the number of stones held, it can be seen that w_k cannot have a stone on it. Thus, the next node computed is either one of x_k and $\bar{x}_k$, or w_k. Suppose one of x_k and $\bar{x}_k$ is computed. Then a stone can be moved from one of u_{k1} and u_{k2}, as appropriate, to w_k. Moreover, once one of x_k and $\bar{x}_k$ is computed, $2n - 2k$ stones will be free, and the other element of $\{x_k,\bar{x}_k\}$ cannot be computed.

If neither x_k nor $\bar{x}_k$ is computed, w_k must still be computed before z_{k+1}. Therefore, again, $2n - 2k$ stones will be free. Since z_{k+1} requires $2n - 2k$ stones, the free stones must be saved for the direct descendants of z_{k+1}.

□

LEMMA 3.8: Let D be the dag constructed by reduction 1 for an $(m-3,n)$ satisfiability problem. For all moves between (and not including) the moves at which w_n and d are computed, there are no stones free.

PROOF: From lemma 3.7, 1 stone is free when z_n is computed. It is easy to see that just after w_n is computed, ther are no stones free.

Suppose the lemma is false. Then at some move between the moves at which w_n and d are computed, there is at least one stone free. Let move $k + 1$ be the first such move at which there is a free stone.

Consider move k. By hypothesis, there are no free stones at move k. Thus move k computes a node, say v, by taking a stone from a direct descendant of v, to v. Since we are interested only in moves between the moves that compute w_n and d, v can only be an element of the set C. Since nodes in C have d as a direct ancestor, the stone at node v cannot be free at the next move. Hence the stone that is freed must be on a direct descendant of v.

Since v is an element of C, let v be c_i, for some i, $1 \leq i \leq m$. By construction, c_i has f_{i1}, f_{i2} and f_{i3} as direct descendants. Let clause i be $y_{i1} \vee y_{i2} \vee y_{i3}$. For all j, $1 \leq j \leq 3$, let y_{ij} and $\overline{y}_{ij}$ refer to elements of the set X, as given in the specificication of the set E10 of edges.

<u>Case 1</u>: A stone is moved from f_{i1} to c_i. Then node y_{i1}, a direct ancestor of f_{i1} must have already been computed. From lemma 3.7, $\overline{y}_{i1}$ has not yet been computed. Thus f_{i2} and f_{i3} have a direct ancestor, $\overline{y}_{i1}$, that has yet to be computed. Hence, the stones at f_{i2} and f_{i3} cannot be free, contradicting the supposition that there is a free stone at move k + 1.

The remaining cases follow quite simply.

□

LEMMA 3.9: Let D be the dag constructed by reduction 1 for an (m-3,n) satisfiability problem. If D is computed using 3m + 4n + 1 3m + 4n + 1 + #B registers, then the conjunction of the clauses is satisfiable.

PROOF: From lemma 3.7, for all k, $1 \leq k \leq n$, at most one of x_k and $\overline{x}_k$ has been computed, just after the computation of z_n. If x_k has been computed, assign the value true to the literal x_k, and false to $\overline{x}_k$. Otherwise assign the value false to x_k, and true to $\overline{x}_k$. Suppose this assignment of values is such that the conjunction of the clauses is not satisfied. Then we will show that a contradiction must occur.

If the conjunction is not satisfied, there must be at least one clause, say i, $1 \leq i \leq m$, such that all literals in caluse i are false. Let these literals be y_{i1}, y_{i2} and y_{i3}. Since y_{i1} is a literal, for some k, $1 \leq k \leq n$, y_{i1} is either x_k or $\overline{x}_k$. If y_{i1} is x_k, then from the assignment of values to literals, y_{i1} being false, x_k has not yet been computed. Similarly if y_{i1} is $\overline{x}_k$, then $\overline{x}_k$ has not yet been computed. Evidently, f_{i1} will have a direct ancestor that has not yet been computed. Similarly, f_{i2} and f_{i3} will also have direct ancestors that have not been computed. Thus the stones at f_{i1}, f_{i2} and f_{i3} will be held there.

From lemma 3.8, there are no free stones when c_i is computed. Hence c_i cannot be computed without using an extra stone. But then D cannot be computed using 3m + 4n + 1 + #B stones. Contradiction

□

LEMMA 3.10: Problem 1 is in NP.

PROOF: If $k \leq n$, the number of nodes in a dag D, then the dag can be computed using no more than k registers. Therefore, suppose k<n.

Given the dag D, and the integer k, let T be a multitape nondeterministic Turing machine that generates a sequence of n pairs, $(i_1,x_1),(i_2,x_2),\ldots,(i_n,x_n)$, $1 \leq i_j \leq k$, and x_j is a node in D. The integers can be represented in binary notation, and the nodes by the symbol "x" followed by an integer in binary notation. The length of the sequence will be O(n log n).

Intuitively, the pair (i,x) may be thought of as specifying that the stone i is placed on node x.

The Turing machine then scans the sequence generated to see if there is a computation of D that corresponds to the sequence. The time taken by T is clearly polynomial in n, and independent of k. From [20], there is a one-tape nondeterministic Turing machine that accepts problem 1 in polynomial time.

□

THEOREM 3.11: Given an integer k, the problem of determining if there is a computation of a dag that uses no more than k registers, is polynomial complete (no recomputation of nodes).

PROOF: Note that the statement of the theorem refers to problem 1.

From [19,21], the satisfiability problem with exactly three literals per clause is polynomial complete. Therefore, given lemmas 3.6-3.10, all we need to show is that the dag D constructed by reduction 1 for an (m-3,n) satisfiability is constructed deterministically in polynomial time.

Note that the number of nodes in D depends only on n, the number of variables, and m, the number of clauses. Moreover, the number of nodes in D is $O(n^2+m)$. Of the sets E1-E10 of edges, E1-E9, depend only on n and m. A list of ordered pairs of nodes, specifying the edges in D can be constructed in one pass over the satisfiability problem.

□

4. Permitting Recomputation

The reduction of the last section relied heavily on the ability to hold stones at designated nodes. If recomputation is permitted, the constructions of the last section are no longer adequate. However, we will show how minor modifications of the construction in the last section permit the reduction to work.

DEFINITION 4.1: The computation of a dag is a sequence of moves in game 1, that starts with no stones on any node in the dag, and ends with stones on all the roots in the dag.

□

EXAMPLE 4.2: Consider the dag in figure 4.3. In order to compute node b, m stones must be placed on the nodes $a_1,a_2,\dots,a_m$. In order to compute node d, stones must be placed on nodes in the set

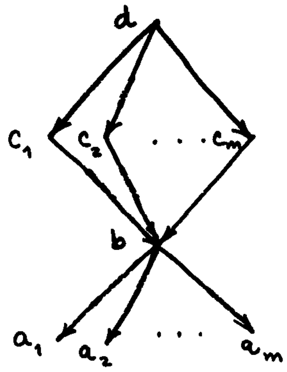


Figure 4.3

$C = \{c_1,c_2,\dots,c_m\}$. If there are exactly m stones that may be used, once stones are placed on all nodes in the set C, none of the nodes in C may be recomputed. In order to recompute a node in C, there must be a stone on b. Since it takes m stones to compute b, recomputing a node in C is tantamount to starting afresh.

Clearly, treating d as the "initial" node, and elements of C as "leaves", we can ensure that no "leaves" are recomputed.

□

EXAMPLE 4.4: Consider the dag in figure 4.5. As in example 3.2, let a circle at a node mean that the node is a direct descendant of a final node that is the last node to be computed. Triangles at leaves h_1 and h_2 mean that the computation starts by placing stones on these leaves. Moreover, assume that h_1 and h_2 may not be recomputed. From example 4.2, such an assumption is enforceable. We will show how a stone may be held at node z.

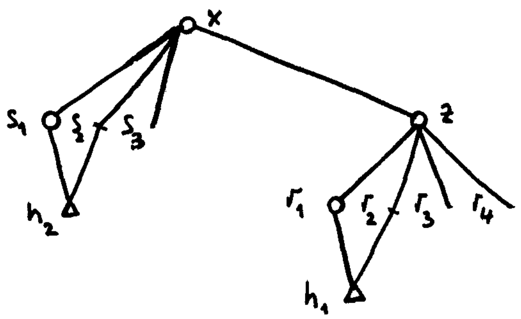


Figure 4.5

Suppose that in addition to the 2 stones on h_1 and h_2, there are 3 stones available. Since z must be computed before x, there is little point in placing one of these three stones on s_1-s_3. In order to compute z, there must be stones on r_1-r_4. Since h_2 may not be recomputed, the remaining four stones must be placed on r_1-r_4. The important point is that a stone may not be held at h_1. Since h_1 may not be recomputed, neither r_1 or r_2 may be recomputed. Node r_1, being a direct descendant of the "final" node, will therefore hold a stone.

Consider node z. Since z is a direct descendant of the "final" node, there must be a stone on z when the "final" node is computed. In order to compute node x, there must be stones on s_1,s_2,s_3 and z. Since at most five stones may be used, and one is held at r_1, stones may not be held at either h_2 or r_2. Since r_2 may not be recomputed, the stone at z must remain there until the "final" node is computed.

□

REDUCTION 2: Let the terms used here be as in reduction 1. Let sets $C,F,U,W,X,$ and Z be as in reduction 1.

Nodes: $A \cup B \cup C \cup F \cup H \cup L \cup M \cup RSTV \cup U \cup W \cup X \cup Z$

$A = \{a_j \mid 1 \le j \le 2n + 2\}$

$B = \{b_j \mid 1 \le j \le 2n - m + 1\}$

$H = \{h_j \mid 1 \le j \le 4n\}$

$L = \{\ell_j \mid 1 \le j \le 3m + 8n + 2 + \#B\}$

$M = \{\text{pivot, initial, d, final}\}$

$RSTV = \{r_{kj} \mid 1 \le k \le n,\ 1 \le j \le 2n - 2k + 4\} \cup \{s_{kj},t_{kj},v_{kj} \mid 1 \le k \le n,\ 1 \le j \le 2n - 2k+3\}$

Edges: $E0 \cup E1 \cup \dots \cup E10$

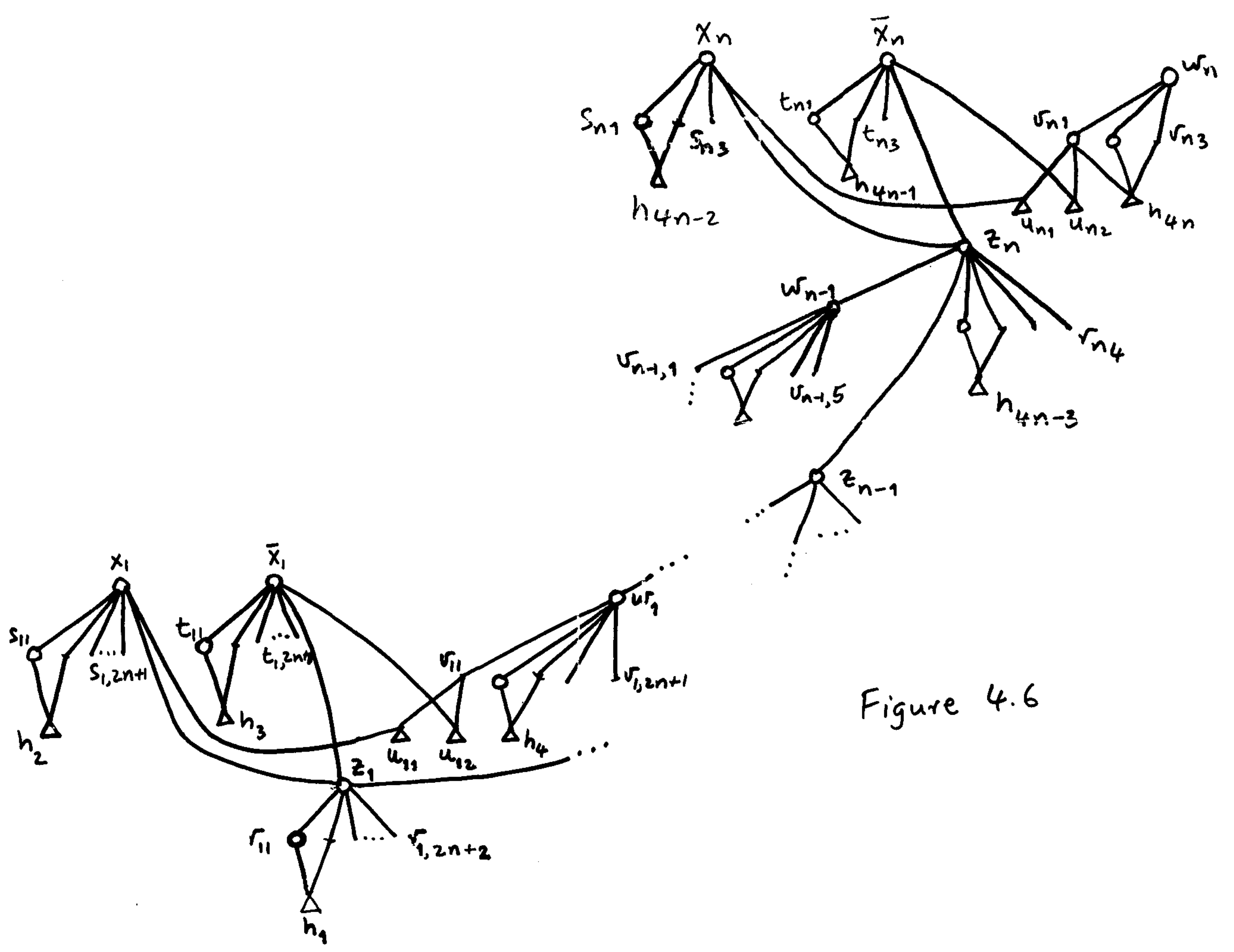


Figure 4.6

$E0 = \{(\text{pivot},g)|g\in L\}\cup\{(g,\text{pivot})|g\in A\cup B\cup F\cup H\cup U\}$

$E1 = \{(\text{initial},g)|g\in A\cup B\cup F\cup H\cup U\}$

$E2 = \{(g,\text{initial})|g\in C\cup RSTV\}$

$E3 = \{(\text{final},g)|g\in W\cup X\cup Z\cup\{r_{k1},s_{k1},t_{k1},v_{k2}|1\le k\le n\}$
$\quad \cup \{\text{initial},d,v_{n1}\}\}$

$E4 = \{(x_k,z_k),\ (\bar{x}_k,z_k),(x_k,u_{k1}),(\bar{x}_k,u_{k2})\ 1\le k\le n\}$

$E5 = \{(v_{k1},u_{kj})|1\le k\le n,\ 1\le j\le 2\}\cup\{(v_{n1},h_{4n})\}$

$E6 = \{(x_k,s_{kj}),(\bar{x}_k,t_{kj}),(w_k,v_{kj})|1\le k\le n,$
$\quad 1\le j\le 2n-2k+3\}\ \cup$
$\quad \{(z_k,r_{kj})|1\le k\le n,\ 1\le j\le 2n-2k+4\}$

$E6' = \{(r_{kj},h_{4k-3}),(s_{kj},h_{4k-2}),(t_{kj},h_{4k-1}),$
$\quad (v_{k,j+1},h_{4k})|1\le k\le n,\ 1\le j\le 2\}$

sets E7-E10 are as for reduction 1. □

THEOREM 4.7: Given an integer k, the problem of determining if there is a computation of a dag that uses no more than k registers is polynomial complete.

PROOF: Similar to the proofs in section 3. For further details see [22]. □

5. Validating Register Allocations

In this section we consider a seemingly simpler problem in register allocation. Suppose no value is computed more than once. Then, for any computation, a register can be associated with each node. In other words, each computation defines a function from nodes to registers. From figure 1.3, the converse - for each function from nodes to registers, there exists a computation - is not true. Here we examine, if given a function from nodes to registers, there exists a computation that computes nodes into those registers.

DEFINITION: Let $Q = x_1x_2\ldots x_n$ be a sequence of nodes in a dag. Node u is said to *appear before* node v in Q, if for some i, j, $1 \le i < j \le n$, u is x_i and v is x_j. Node v is said to *appear after* node u in Q. If a node u appears before node v, and v appears before node w in Q, then v is said to *appear between* u and w in Q. The term *occur* may sometimes be substituted for "appear". □

DEFINITION: Let $Q = x_1 x_2 \ldots x_n$ be a sequence of nodes in a dag D. Q is called a complete sequence of nodes in D if every node in D appears exactly once in Q.

DEFINITION: Given a dag D, let L be a function from nodes in D into the set of names. L will be called an allocation for D. The pair (D,L) will often be referred to as a program dag, or just program. □

Suppose a value is in a specified register. It should be retained in the register, at least as long as it is needed. It will be needed until all its direct ancestors have been computed.

DEFINITION: Given a program (D,L), Let Q be a sequence of nodes in D. (Q,L) is said to be consistent, if for all nodes u, v and w in D, if

(1) u is a direct descendant of w, and
(2) v appears between u and w,

then $L(u) \neq L(v)$.

DEFINITION: Let Q be a complete sequence of nodes in a dag D. Q is called a realization of a program (D,L), if (W,L) is consistent, and for all nodes u and v in D, if u appears before v in D, then v is not a descendant of u. □

Completeness of a sequence ensures that an attempt will be made to compute every node. A realization also forces descendants to be computed before their ancestors. Consistency ensures that the value of a node will be retained in the appropriate register, as long as it is needed.

EXAMPLE 5.1: Example 1.1 gave a program to compute the dag in figure 1.2 using three registers. The realization corresponding to that program is given by figure 5.2.

node	c	x	t1	b	t2	t3	a	t4
stone	1	2	1	3	1	1	2	3

Figure 5.2

$$\begin{tikzcd} c & x & b & a & t1 \arrow[lll, bend right] & t2 & t3 & t4 \\ 1 & 2 & 3 & 2 & 1 & 1 & 1 & 3 \end{tikzcd}$$

Figure 5.3

Consider the sequence Q in figure 5.3. Q is complete, since it contains all the nodes in D. Moreover descendants appear before their ancestors in Q. However Q is not a realization of D, since (Q,L) is not consistent -- the value of node x is needed to compute t1, but placing a into register 2 destroys the value of x before it can be used. □

PROBLEM 3: Given a program dag (D,L), does (D,L) have a realization? □

REDUCTION 3: Given an m-clause satisfiability problem over n variables $x_1, x_2, \ldots, x_n$, where for all i, $1 \leq i \leq m$, clause i has exactly 3 literals, y_{i1}, y_{i2} and y_{i3}, construct a program dag (D,L) as follows:

1. For all k, $1 \leq k \leq n$, construct two leaves s_k and $\bar{s}_k$, corresponding to the literals x_k $\bar{x}_k$. Let $L(s_k) = S_k$.

2. For all i, j, $1 \leq i \leq m$, $1 \leq j \leq 3$, construct nodes p_{ij}, q_{ij}, r_{ij} and $\bar{r}_{ij}$, as in figure 5.4, and edges (q_{ij}, p_{ij}), (p_{ij}, r_{ij}). Let $L(p_{ij}) = P_{ij}$, $L(q_{ij}) = Q_{ij}$ and $L(r_{ij}) = L(\bar{r}_{ij}) = R_{ij}$. Also construct the edges $(q_{i1}, \bar{r}_{i2})$, (q_{i2}, r_{i3}) and $(q_{i3}, \bar{r}_{i1})$.

The subdag created in figure 5.4 corresponds to clause i. The nodes r_{ij} and $\bar{r}_{ij}$ correspond to the literals y_{ij} and $\bar{y}_{ij}$.

3. For all i, $1 \leq i \leq m$, clause i consists of the literals y_{i1}, y_{i2} and y_{i3}. For all j, $1 \leq j \leq 3$, since y_{ij} is a literal, there exists a k, $1 \leq k \leq n$, such that y_{ij} is either x_k or $\bar{x}_k$. If $y_{ij} = x_k$, we use the symbol "y_{ij}" to refer to node s_k, and "$\bar{y}_{ij}$" to refer to node $\bar{s}_k$. Otherwise, if $y_{ij} = \bar{x}_k$, we use the symbol "y_{ij}" to refer to node $\bar{s}_k$ and "$\bar{y}_{ij}$" to refer to node s_k.

For all i, j, $1 \leq i \leq m$, $1 \leq j \leq 3$, construct the edges (r_{ij}, y_{ij}) and $(\bar{r}_{ij}, \bar{y}_{ij})$. □

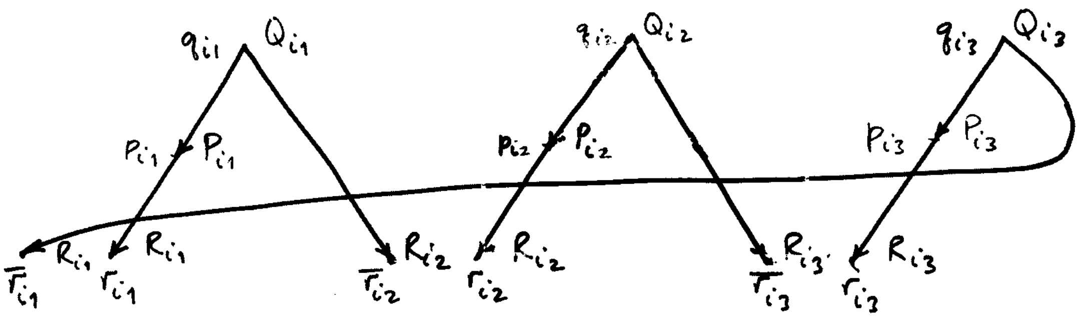
LEMMA 5.5: Let (D,L) be the program constructed by reduction 3 for an (m-3,n) satisfiability problem. If the conjunction of clauses is satisfiable, then (D,L) has a realization.

PROOF: We will construct a realization for (D,L).

1. Initially the sequence Q is empty. As a convention, nodes may be added to Q on the right only. Do 2 for $k = 1, 2, \ldots, n$.

2. If the literal x_k is true for the conjunction of clauses to be satisfiable, then add s_k, all direct ancestors of s_k, and $\bar{s}_k$ to Q. Otherwise, add $\bar{s}_k$, all direct ancestors of $\bar{s}_k$ and s_k to Q.

Note that all direct ancestors of s_k and $\bar{s}_k$ are elements of the set $\{r_{ij}, \bar{r}_{ij}\ 1 \leq i \leq m, 1 \leq j \leq 3\}$. Each element of



q_{i1} Q_{i1}
p_{i1} P_{i1}
$\bar{r}_{i1}$ R_{i1} r_{i1} R_{i1}
q_{i2} Q_{i2}
p_{i2} P_{i2}
$\bar{r}_{i2}$ R_{i2} r_{i2} R_{i2}
q_{i3} Q_{i3}
p_{i3} P_{i3}
$\bar{r}_{i3}$ R_{i3} r_{i3} R_{i3}

Figure 5.4

this set has only one direct descendant. Moreover, by construction, for all i, j, $1 \leq i \leq m$, $1 \leq j \leq 3$, both r_{ij} and $\bar{r}_{ij}$ cannot both be direct ancestors of the same node. Therefore only one of r_{ij} and $\bar{r}_{ij}$ has been added to Q. Since $L(r_{ij})$ is equal only to $L(\bar{r}_{ij})$, (Q,L) must so far be consistent.

3. For all i, j, $1 \leq i \leq m$, $1 \leq j \leq 3$, if r_{ij} has been added to Q, then add p_{ij} and $\bar{r}_{ij}$ to Q. Note that p_{ij} is the unique direct ancestor of r_{ij}, and that for all nodes x in D, if $x \neq p_{ij}$, then $L(x) \neq L(p_{ij})$.

4. At this stage, note that for all i, j, $1 \leq i \leq m$, $1 \leq j \leq 3$, $\bar{r}_{ij}$ has been added to the list, but that r_{ij} may not have. For all i, $1 \leq i \leq m$, do 5.

5. Consider clause i, given by $y_{i1} \vee y_{i2} \vee y_{i3}$. Since the conjunction of clauses is satisfiable, clause i must be true. Without loss of generality let y_{i1} be true. We will show that r_{i1} appears in Q before $\bar{r}_{i1}$.

Since y_{i1} is a literal, for some k, $1 \leq k \leq n$, y_{i1} is either x_k or $\bar{x}_k$. If y_{i1} is x_k, then, by construction, r_{i1} is a direct ancestor of s_k. Since y_{i1} is true x_k must be true, so from part 2, above, s_k and r_{i1} are added to Q.

If, on the other hand, y_{i1} is $\bar{x}_k$, then, by construction, r_{i1} is a direct ancestor of $\bar{s}_k$. Since y_{i1} is true, $\bar{x}_k$ is true, and $\bar{s}_k$ and r_{i1} are added to Q by part 2.

Since both p_{i1} and $\bar{r}_{i2}$ have been added to Q, node q_{i1} can now be added to the sequence Q. Once q_{i1} is added to Q, $L(\bar{r}_{i2})$, which $\bar{r}_{i2}$ shares with r_{i2}, can be used for r_{i2}, unless of course, r_{i2} is already in Q. In either case, q_{i2} can now be computed, and similarly q_{i3}.

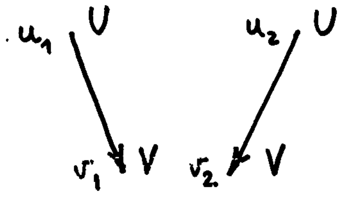


Figure 5.6

LEMMA 5.7: Let u_1, u_2, v_1 and v_2 be nodes in a dag D such that for $i = 1,2$, u_i is a direct ancestor of v_i. Let $L(u_1) = L(u_2)$ and $L(v_1) = L(v_2)$. Let Q be a realization of (D,L). Then v_1 appears before v_2 in Q if and only if u_1 appears before u_2.

PROOF: Suppose u_1 appears before u_2, but that v_2 appears before v_1. We will show that a contradiction must occur.

Since Q is a realization of (D,L), descendants must appear before their ancestors. Since v_1 is a descendant of u_1, it follows that the nodes appear in the order v_2, v_1, u_1, u_2. Since u_2 is a direct ancestor of v_2, and $L(v_1)=L(v_2)$, (Q,L) cannot be consistent. Hence Q cannot be a realization of (D,L) contradiction.

The converse follows similarly.

□

LEMMA 5.8: Let (D,L) be the program constructed by reduction 3 for an (m-3,n) satisfiability problem. Let Q be a realization of (D,L). Then for all i, $1 \leq i \leq m$ there exists a j, $1 \leq j \leq 3$, such that r_{ij} appears before $\bar{r}_{ij}$ in Q.

PROOF: Suppose the lemma is false. Then there exists an i, $1 \leq i \leq m$, such that for all j, $1 \leq j \leq 3$, $\bar{r}_{ij}$ appears before r_{ij} in Q. We will show that a contradiction must occur.

(In figure 5.4) note that $L(\bar{r}_{ij}) = L(r_{ij})$. Thus, for (Q,L) to be consistent, any direct ancestors of $\bar{r}_{ij}$ must appear before r_{ij} in Q. Note also that r_{ij}, being a descendant of q_{ij}, must appear before q_{ij} in Q. We therefore conclude that:

q_{i1}	before	r_{i2}
r_{i2}	before	q_{i2}
q_{i2}	before	r_{i3}
r_{i3}	before	q_{i3}
q_{i3}	before	r_{i1}
r_{i1}	before	q_{i1}

Thus Q cannot be a realization of (D,L) contradiction.

□

LEMMA 5.9: Let (D,L) be the program constructed by reduction 3 for an (m-3,n) satisfiability problem. (D,L) has a realization if and only if the conjunction of clauses is satisfiable.

PROOF: The if part is provided by lemma 5.5. So we only need to show that if (D,L) has a realization then the conjunction of clauses is satisfiable.

Let Q be a realization for (D,L). For all k, $1 \leq k \leq n$, if s_k is computed before $\bar{s}_k$, assign the value true to x_k, otherwise assign the value false to x_k.

Suppose this assignment of values is such that the conjunction of clauses is not satisfied. Then we will show that a contradiction must occur.

Since the conjunction of clauses is not satisfied, there must be at least one clause such that all the literals in the clause are false. Let clause i be such a clause.

From lemma 5.8, there exists a j, $1 \leq j \leq 3$, such that r_{ij} appears before $\bar{r}_{ij}$. Without loss of generality, let j be 1. For some k, $1 \leq k \leq n$, r_{i1} either has s_k or $\bar{s}_k$ as direct descendant.

case 1: s_k is a direct descendant of r_{i1}. Since r_{i1} appears before $\bar{r}_{i1}$, from lemma 5.7, s_k appears before $\bar{s}_k$. Thus x_k is assigned the value true. By construction, the literal y_{i1} must be x_k. Hence y_{i1} must be true. Contradiction.

The other case follows similarly.

□

LEMMA 5.10: Problem 3 is in NP.

PROOF: Straightforward. □

THEOREM 5.11: Given a program (D,L) the problem of determining if (D,L) has a realization is polynomial complete.

PROOF: The construction of reduction 3 can clearly be performed in polynomial time. The theorem follows from lemma 5.5, 5.7-5.10.

□

6. Other Problems

The problems in sections 3 and 4 were concerned with the number of registers used. When recomputation is permitted, another concern might be the length of the computation. Clearly, the lower bound on the length of the computation is given by the case in which no nodes are recomputed.+ This bound can be achieved by placing a fresh stone on every node. Therefore it only makes sense to talk of limiting the length of a computation if there is also a bound on the number of registers.

In addition to the game that has been considered so far, Walker [15], considers a number of other games. For example, we may have a game that models two levels of storage -- registers and core memory.

GAME 2: Let there be a finite set of r red stones (registers) $R_1, R_2, \ldots, R_r$, and an infinite set of black stones (core locations) $L_1, L_2, \ldots$.

A move in game 2 is one of the following:

1. place a black stone on a leaf
2. exchange a red stone for a black stone
3. exchange a black stone for a red stone

if there are red stones on every direct descendant of a node x, then:

4. place a red stone on x, or
5. move a red stone to x from one of the direct descendants of x

□

+We should really exclude moves that pick up stones when determining the length of a computation, since such a move is not necessary in practice.

Given a fixed value of r, there are $O(n^r)$ ways of arranging red stones on nodes in the dag. Note that the black stones can be treated as indistinguishable. Thus it is expected that problems based on game 2 will be in NP. Moreover, problems based on game 2 easily encompass problems based on game 1.

Another possible game arises from the incorporation of "parallel assignment" instructions. Note that the dag in figure 7.1 would require three

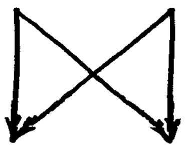


Figure 6.1

stones if computed using the definition of computation in sections 3 and 4. But, if instead of allowing a stone to be placed on only one node at a time, stone could be placed on all direct ancestors of a set of nodes, then the dag in figure 6.1 could be computed using two stones.

GAME 3: Let there be an infinite supply of labelled stones. A move in game 3 is one of the following:

1. place a stone on a leaf
2. pick up a stone from a node

if there are stones on all direct descendants of a set of nodes $X = \{x_1, x_2, \ldots, x_n\}$, then:

3. places stones on a subset of X, or
4. for some subset W of X, move stones to nodes in W as follows:

For all $w \in W$, move a stone to w from a direct descendant of w.

Note that it is not necessary to consider combinations of 3 and 4 above.

□

It is expected that the reductions in sections 3 and 4 would also work for problems based on game 3.

7. Practical Significance

Fortunately, the dags used in the reductions in this paper tend not to occur in practice. In most programs straight line sections tend to be fairly small. The practical significance of the results that have been presented is twofold: (1) Since the dags that occur in practice tend to be simple, it would be worthwhile to study register allocation for restricted classes of dags; (2) If the dags are small enough, then efficient enumerative techniques might be worth considering.

Acknowledgments: Interest, suggestions and criticism by John Bruno, Neil Jones, Ray Strong and Jeff Ullman are greatly appreciated.

References

1. J. W. Backus, et al, The FORTRAN automatic coding system. Proc. WJCC 11 (1957) 188-198, also in S. Rosen (ed) Programming systems and languages, McGraw-Hill, New York. (1967) 29-47.

2. J. C. Beatty, A global register assignment algorithm, in R. Rustin (ed) Design and optimization of compilers, Prentice-Hall, Englewood Cliffs, N. J. (1972) 65-68.

3. J. C. Beatty, An axiomatic approach to code optimization for expressions. JACM 19,4 (Oct. 1972), 613-640.

4. M. A. Breuer, Generation of optimal code for arithmetic expressions via factorization, CACM 12,6 (June 1969) 333-340.

5. V. A. Busam, D. W. Englund, Optimization of expressions in FORTRAN, CACM 12,12 (Dec. 1969), 666-674.

6. G. Chroust, Scope conserving expression evaluation, IFIP '71, TA-3, 178-182.

7. W. H. E. Day, Compiler assignment of data items to registers, IBM Sys. J. 9,4 (1970) 281-317.

8. L. P. Horwitz, R. M. Karp, R. E. Miller, S. Winograd, Index register allocation, JACM 13,. (Jan. 1966) 43-61.

9. K. Kennedy, Index register allocation in straight line code and simple loops. In R. Rustin (ed) Design and optimization of compilers, Prentice-Hall, Englewood Cliffs, N. J., (1972) 51-64.

10. E. S. Lowry, C. W. Medlock, Object code optimization CACM 12,1 (Jan. 1969) 13-22.

11. T. Marill, Computational chains and the simplification of computer programs, IRE Trans. on Elec. Comp. EC-11,2 (April 1962) 173-180.

12. Ikuo Nakata, On compiling algorithms for arithmetic expressions, CACM 10,8 (Aug. 1967) 492-494.

13. R. R. Redziejowski, On arithmetic expressions and trees, CACM 12,2 (Feb. 1969) 81-84.

14. R. Sethi, J. D. Ullman, The generation of optimal code for arithmetic expressions, JACM 17, 4 (Oct. 1970) 715-728.

15. S. A. Walker, Some graph games related to the efficient calculation of expressions, IBM Res. Rep. RC-3628 (Nov. 1971) 17 p.

16. S. A. Walker, H. R. Strong, Characterizations of flow chartable recursions, 4th Ann. Sym. on theory of computing, Denver, Colo. (May 1972) 18-34.

17. L. A. Belady, A study of replacement algorithms for a virtual storage computer, IBM Sys. J. 5,2 (1966) 78-101.

18. A. V. Aho, J. D. Ullman, Optimization of straight line programs, SIAM J. Computing 1,1 (Mar. 1972) 1-19.

19. S. A. Cook, The complexity of theorem-proving procedures, 3rd. Ann. Sym. on Th. of Computing, Shaker Heights, Ohio (May 1971), 151-158.

20. J. E. Hopcroft, J. D. Ullman, Formal languages and their relation to automata, Addison Wesley, Reading, Mass. (1969).

21. R. M. Karp, Reducibility among combinatorial problems, Tech. Rep. No. 3, Computer Science Dept., Univ. of Cal., Berkeley (April 1972).

22. R. Sethi, Complete register allocation problems, Tech. Rep. No. 134, Computer Science Dept., Penn State Univ., Univ. Park, Pa. (Jan. 73).