On the Sequential Nature of Unification

Cynthia Dwork[1] *MIT*
Paris C. Kanellakis[2] *Brown Univ.*
John C. Mitchell[3] *MIT*

November 1983

Abstract

The problem of unification of terms is log-space complete for P. In deriving this lower bound no use is made of the potentially concise representation of terms by directed acyclic graphs. In addition, the problem remains complete even if infinite substitutions are allowed. A consequence of this result is that parallelism cannot significantly improve on the best sequential solutions for unification. The "dual" problem of computing the congruence closure of an equivalence relation is also log-space complete for P. However, we show that for the problem of term matching, an important subcase of unification, there is a good parallel algorithm using $O(\log^2 n)$ time and $n^{O(1)}$ processors on a PRAM. For the $O(\log^2 n)$ parallel time upper bound we assume that the terms are represented by directed acyclic graphs; if the longer string representation is used we obtain an $O(\log n)$ parallel time bound.

1. Introduction

Unification is an important step in resolution theorem proving [R] with applications to a variety of symbolic computation problems. In particular, unification is used in PROLOG interpreters [CM], type inference algorithms [M], and term rewriting systems [GKM]. Many symbol manipulation problems are inherently difficult and thus do not have efficient solutions. Theorem provers and PROLOG interpreters do not always give us the answers we want fast enough. One way to combat the difficulty of these problems is by coordinating many processors to solve a single problem instance by working on several subproblems in parallel. Although there are a number of ways to introduce parallelism into interpreters [S] and theorem provers, unification is a prime target since it is the most commonly repeated operation in these tasks. However, our analysis suggests that parallel unification algorithms will not perform significantly faster than the best sequential algorithms known (e.g., [PW] runs in linear time). We show that, unless $P \subseteq NC$, an unlikely twist of complexity theory [C], no parallel algorithm for unification will run in time bounded by a polynomial in the logarithm of the input size, and using a number of processors bounded by a polynomial in the size of the input. We use the PRAM of [FW] as our model of parallel computation, although we could, just as well, have used any other "reasonable parallel model" [J].

---

[1] Supported by a Bantrell Fellowship. [2] Supported partly by NSF grant MCS-8210830 and partly by ONR-DARPA grant N00014-83-K-0146. [3] Supported by an IBM Fellowship.

Informally, two symbolic terms s and t are unifiable if there is some way of substituting additional terms for variables in s and t so that both become the same term. All occurences of a variable x in both s and t must be replaced by the same term. For example, the terms f(x, x) and f(g(y), g(g(z))) may be unified by substituting g(z) for y and g(g(z)) for x. A unification problem like "unify f(t_1, t_2) and f(t_3, t_4)" may be decomposed into two subproblems "unify t_1 and t_3" and "unify t_2 and t_4". However, these two problems cannot be solved entirely separately in parallel. If some variable x occurs in both t_1 and t_4, for example, then the solutions to the subproblems must be coordinated so that both substitute the same term for x.

There are several variations of the unification problem. For example, a type inference algorithm may construct labeled graphs which represent terms that must be unified. An acceptable result of unification, in this case, may be a labeled graph with a cycle. Labeled graphs with cycles represent types defined by recursion [MPS], or, if interpreted as terms, represent "infinite terms". Thus one natural, unrestricted version of unification is to allow "infinite terms" to be substituted for variables. Using the "infinite term" f(f(f...)), we can unify x and f(x), something we could not do otherwise. Unrestricted unification also appears in many PROLOG interpreters; those omitting the *occur* test [CM]. Another variation on unification is the special case in which the labeled graphs are from a class of tree-like directed acyclic graphs (which we call *simple dags*). The complexity of unification on simple dags is precisely the complexity of unification on symbolic (string) representations of terms, as opposed to the complexity as a function of the size of more concise graph representations. For this case it was known that unification, without "infinite terms", is co-NLOGSPACE-hard [LS]. This did not exclude the possibility of parallel algorithms, moreover no lower bound was known for unrestricted unification.

We show that all of the above variants of unification are log-space complete for P [C, G1, G2], and hence unlikely to have nice parallel solutions. The nondeterministic log-space test for ununifiability in [LS], which could have led to a $O(\log^2 n)$ parallel time solution, is sufficient, but unfortunately not necessary (see Figure 3b for a counterexample to this test). In addition, we show that the related problem of congruence closure [DST] is complete for P.

One important special case of unification can be solved quickly in parallel. This problem called term matching, arises in term rewriting. A term s matches a term t if t is a substitution instance of s. The rewrite rule $l \rightarrow r$ may be used to rewrite a term t whenever l matches t [GKM]. We show that matching can be accomplished in $\log^2$-time on a PRAM, using a polynomial number of processors. Our algorithm combines parallel transitive closure of a directed acyclic graph, with parallel computation of connected components of an undirected graph [HCS, Ch]. Also, matching is in NLOGSPACE, and for simple dags it is in DLOGSPACE.

Following the definitions presented in Section 2, we will discuss labeled graph unification in Section 3, unification for simple dags and congruence closure in Section 4, and term matching in Section 5.

2. Definitions

2.1 Terms and Dags

Let *V* be an infinite set of *variables* x,y,z,x_1.... and *F* an infinite set of *function symbols* f,g,h,f_1.... We assume that *V* and *F* are disjoint. Each function symbol f has a fixed *arity*, a nonnegative integer a(f). A function symbol g∈*F* with a(g)=0 is called a *constant*. The set *T* of *terms* is defined inductively by:

a variable x∈*V* or constant g∈*F* is a term, and

if f∈*F* and $t_1,\ldots,t_{a(f)}$ are terms, then $f(t_1,\ldots,t_{a(f)})$ is a term.

Terms may be represented using directed acyclic graphs with labeled nodes and, possibly, multiple labelled arcs. A *labeled directed graph* is a finite directed graph G, such that:

(1) every node v of G has a unique label, denoted label(v), with label(v)∈$V \dot{\cup} F$,

(2) for each x∈*V*, there is at most one node v with label(v)=x, and it has outdegree 0,

(3) if a node v has label f∈*F*, with arity a(f)≥0, then it has outdegree a(f), and the arcs leaving it are labeled 1,2,...,a(f).

If there is an arc labeled i from node u to node v, then we say that v is the i-th *son* of u. A *labeled dag* G is a labeled directed acyclic graph. The *leaves* of G are the nodes of outdegree 0; note that a node v is a leaf iff label(v) is either a variable or a constant. The *height* of a node v of a dag G is the length of the longest path from v to a leaf. A *root* of a dag is a node of indegree 0.

If G is a labeled dag, we can associate a term t_v with any node v of G. We say that v *represents* t_v. The term t_v is defined by induction on the height of v:

if v is a leaf, then t_v=label(v),

if v has sons $v_1,\ldots,v_k$, and label(v)=f, then $t_v=f(t_{v_1},\ldots,t_{v_k})$.

The definition of labeled dags above ensures that t_v is always a well-formed term. If G is a labeled directed graph, then we can associate an *infinite term* t_v with each node v of G by a similar definition. Since we only consider finite graphs, all terms represented by nodes of a labeled graph G are finite iff G is acyclic. If G is a labeled dag with only one root r, then we say that G represents the term t_r.

The representation of terms by labeled dags is illustrated in Figure 1. The terms g(x) and x are represented by the two nodes of the labeled dag in Figure 1a. Both roots in Figures 1b, 1c represent f(f(x, x), f(x, x)). The terms h(x, x, y, z) and h(g(y), g(g(z)), g(g(g_1)), g(g_2)) are represented by the roots of Figure 1d. In Figure 1, we assume that a(f)=2, a(h)=4, a(g)=1, and a(g_1)=a(g_2)=0.

Although each node of a labeled dag determines a single term, the converse is not true. A term t can be represented by several different dags. In particular, if t is a term with several occurrences of a subterm t_1, then we may use a separate subdag for each occurrence of t_1 in t, or use one subdag for all occurrences; cf. Figures 1b and 1c. Since a repeated subterm need be represented only once, it is possible to represent some very long terms with relatively small labeled dags. For example, the dag in Figure 1e with n nodes represents a term with $O(2^n)$ symbols. We define a class of labeled dags which are no more concise that terms.

A *simple dag* is a labeled dag G such that the only nodes of G with indegree greater than 1 are leaves. Thus every node of a simple dag that is not a leaf or a root must have indegree 1. Given a term t (in the form of a string of symbols), we can construct a simple dag representing t in linear time, using only logarithmic space. Similarly, given a simple dag G with a single root, we can write out the term represented by G in linear time and logarithmic space. Moreover, the size of a simple dag, measured in number of nodes and arcs, is within a constant multiplicative factor of the length of the term it represents.

2.2 Unification and Term Matching

Unification and term matching are both problems that are solved by computing substitutions. A *substitution* σ is a mapping from variables to terms such that $\sigma(x)=x$ for all but finitely many $x \in V$. The action of a substitution σ on a term t, written $\sigma(t)$, is the result of replacing each variable x in t by $\sigma(x)$. Thus $\sigma(f(t_1,...,t_k))=f(\sigma(t_1),...,\sigma(t_k))$. In particular, any substitution σ maps every function symbol to itself. We use $=$ to denote syntactic equality of strings.

Two terms s and t are *unifiable* if there exists a substitution σ such that $\sigma(s)=\sigma(t)$. A term s *matches* term t if there exists a substitution σ with $\sigma(s)=t$.

In some instances we may wish to allow substitutions to map variables to infinite terms. If we allow these more general substitutions, then we have the *unrestricted unification* and *unrestricted matching* problems. Unrestricted unification differs from unification (e.g., in Figure 1a x and g(x) are ununifiable but unrestricted unifiable with $\sigma(x)=g(g(...))$ an infinite term). Unrestricted matching and matching are the same; note that we only consider substitutions that involve infinite terms, not unification of infinite terms s and t.

If $\sigma(s)=\sigma(t)$, then σ is called a *unifier* for s and t. A substitution σ is *more general* than a substitution τ if there exists a substitution ρ with $\tau=\rho\circ\sigma$. In [R], it is shown that whenever terms s and t are unifiable, there is a unifier σ for s and t, which is more general than any other unifier. This is called the *most general unifier* (mgu) for s and t. The mgu is unique up to renaming of variables. For example, consider the terms $s=f(x, y)$ and $t=f(g(y), g(z))$ represented in Figure 1f. These terms are unifiable, with mgu $\sigma(x)=g(g(z))$, $\sigma(y)=g(z)$, and $\sigma(z)=z$; then $\sigma(s)=\sigma(t)=f(g(g(z)), g(z))$.

Two terms s and t are unifiable if a certain kind of relation, can be constructed on the nodes of a labeled dag representing s and t. If u and v are two nodes of a labeled dag and if u_i is the i-th son of u and v_i the i-th son of v, for some i, then u_i,v_i are *corresponding* sons of u,v. A relation R on the nodes of a labeled dag is a *correspondence* relation if, for all u,v,u_i,v_i:

$$uRv \rightarrow u_iRv_i \text{ whenever } u_i,v_i \text{ are corresponding sons of } u,v.$$

A correspondence relation that is also an equivalence relation will be called a *c-e relation.* A relation R is *homogeneous* if label(u) and label(v) are not different symbols whenever uRv. An equivalence relation R on nodes of a labeled directed graph G is *acyclic* if the R-equivalence classes are partially ordered by the arcs of G. In [PW], acyclic, homogeneous c-e relations are called valid equivalence relations. These relations characterize unifiability.

Proposition 1: [PW] Let u and v be nodes in a labeled dag G. Then t_u and t_v are unrestricted unifiable iff there is a homogeneous c-e relation R, with uRv. Similarly, t_u and t_v are unifiable iff there is an acyclic, homogeneous c-e relation R, with uRv. □

If R is an acyclic, homogeneous c-e relation on a labeled dag G, then the reduced graph formed by treating each equivalence class as a single node is again a labeled dag. If u and v are the only two roots of G, and uRv, then this reduced graph with a single root represents a term s that is a substitution instance of both t_u and t_v. If R is the *minimal* c-e relation with uRv, then $s=\sigma(t_u)=\sigma(t_v)$, where σ is the mgu of t_u and t_v [PW]. We can extract σ from R by taking $\sigma(x)$ to be the term in the reduced graph that is represented by the node formed from the equivalence class of x. We can therefore consider the reduced labeled dag as a reasonable representation of a unifier for two terms. This representation of a unifier has the virtue of being compact; it is clear that the reduced graph is no larger than the original dag. However, if we were to write out each unifier explicitly, we might end up writing out terms that were much longer than the terms represented by the input dag. An example in [PW] shows that the length of the substitution may be an exponential function of the length of the input terms.

As in [PW], we will represent equivalence relations on the nodes of labeled dags by adding undirected edges to the labeled dag data-structure.

Matching may be viewed as a special case of unification. Let σ_c be a substitution such that for each distinct variable x, in the terms we are examining, $\sigma_c(x)$ is c_x, a distinct constant symbol not appearing in these terms. It is easy to see that a term s matches a term t iff s and $\sigma_c(t)$ are unifiable. Another, degenerate case of unification is to determine whether two terms are syntactically identical. Of course, this is a trivial operation on strings, but it is not quite so trivial an operation when terms are represented by labeled dags. Clearly, s and t are syntactically equal iff $\sigma_c(s)$ and $\sigma_c(t)$ are unifiable.

In summary, using the labeled dag data structure, we have the following problems:

UNIFY(G,u,v)
Input: A labeled dag G with distinguished nodes u and v.
Output: Are t_u and t_v unifiable?
If yes, then produce a labeled dag representing the mgu.

MATCH(G,u,v): This is UNIFY(G,u,v) with $\sigma_c(t_v)$ instead of t_v.

EQUAL(G,u,v)): This is UNIFY(G,u,v) with $\sigma_c(t_u)$, $\sigma_c(t_v)$ instead of t_u, t_v.

Of course, there is also unrestricted unification $\mathrm{UNIFY}^{\infty}(G,u,v)$. We have a special case of each of the above problems when G is a simple dag.

2.3 Parallelism, NC and P

For sequential computation we use the standard definitions for time, space, space-bounded reductions and complexity classes such as P, DLOGSPACE, NLOGSPACE, T(n)-DSPACE, on a Random Access Machine (RAM) [C]. We denote log-space reducibility by $\leq_{\log}$. As usual, P is the class of languages recognizable in deterministic polynomial time. The problems UNIFY, MATCH and EQUAL all belong to P [PW, MM]. Some may be solved in $\log^{O(1)} n$ space, while others, those log-space complete for P, most probably cannot.

For parallel computation we use the Parallel RAM (PRAM) of [FW] as our model, with parallel time and number of processors as the critical resources. We make use of the *parallel computation thesis*, relating parallel time and sequential space, and its proof for PRAM's [FW]:

$$\cup_{k\geq 0} \log^k(n)\text{-parallel time-PRAM} = \cup_{k\geq 0} \log^k(n)\text{-DSPACE}.$$

We take NC to be the class of problems solvable on a PRAM using $\log^{O(1)} n$ parallel time, and $n^{O(1)}$ processors. We try to determine whether a problem in P is "parallelizable" (i.e., in NC) or "most probably not parallelizable" (i.e., log-space complete for P); [C, J] review related results.

One problem that is log-space complete for P is the circuit value problem for monotone circuits. A *monote circuit* β is a sequence $(\beta_0,...,\beta_n)$, where each β_i is either an input, an and-gate AND(j,k), or an or-gate OR(j,k); where for indices j,k we have i>j>k, and the 0,1 values of the inputs are given explicitly. In addition, monote circuits are assumed to have the following properties:

(1) if β_i is an input, then the index i appears at most once in β, (fan-out ≤ 1 for inputs),

(2) if β_i is a gate, then the index i appears at most twice in β, (fan-out ≤ 2 for gates),

(3) β_n is an or-gate with one output.

The *monotone circuit value problem* is:

$$\mathrm{MCV}=\{\beta \mid \beta \text{ is a monotone circuit with the output value of } \beta_n = 0\}.$$

From [G1, G2] we have:

Proposition 2: MCV is log-space complete for P. □

3. The Complexity of Unification

The general unification problem, encountered in theorem proving and elsewhere, is to find a simultaneous unifier for a set of terms. However, the general case is log-space and linear time reducible to the special case of unifying a single pair of terms [PW]. On a PRAM this reduction can be performed in O(log n) parallel time and with O(n) processors; it affects none of our results.

We first describe a naive unification algorithm based on the criterion of Proposition 1, and on the fact that the mgu is the minimal c-e relation [PW]. The input to the algorithm is a labeled dag G with two distinguished nodes u and v. We wish to solve UNIFY(G,u,v). A relation ® is constructed and maintained as undirected edges in G. The relation ® is by its representation symmetric and reflexive. In order to make ® a c-e relation, both "correspondence" and "equivalence" must be satisfied. Setting sons equivalent, when their fathers are equivalent, is known as *propagation*. For ® to be an equivalence relation we must also enforce *transitivity*. Having created the minimal c-e relation ® for which u®v, we then test for homogeneity. In the affirmative case a new labeled graph G' can be constructed by coalescing classes of nodes in G. Now we know that the input is at least unrestricted unifiable. If G' is acyclic it is unifiable.

```
proc naive-unification(G,u,v)
set u®v ;
while (® is not a c-e relation) do
propagation: while (u®v have corresponding sons $u_i,v_i$ not related by ®) do set $u_i$®$v_i$ od ;
transitivity: while (u®v and v®w, but u,w are not related by ®) do set u®w od
                            od ;
if ® not homogeneous then print UNUNIFIABLE
                      else {coalesce equivalence classes to produce labeled graph G'}
                        if G' has a cycle
                        then print UNUNIFIABLE BUT UNRESTRICTED UNIFIABLE
                        else print UNIFIABLE
                        fi
fi
proc {G' represents mgu}
```

In this algorithm all individual steps can be performed on a PRAM using $\log^{O(1)} n$ time and $n^{O(1)}$ processors. The difficulty arises in the outer loop, the body of which is executed if ® is either not a correspondence, or not an equivalence relation, i.e., if either condition inside an inner loop is satisfied. The problem is that on an input of size n the body of the main loop might be executed $\Omega(n)$ times. This behavior is illustrated in Figure 2. The example can easily be generalized to force the $\Omega(n)$ alternation between propagation and transitivity for any n.

Theorem 1: UNIFY(G,u,v) and UNIFY^{∞}(G,u,v) are log-space complete for P.

Proof: We show how to log-space reduce MCV to unifiability (for membership see [PW]). More specifically, if α is a monotone circuit $\{\alpha_0,\alpha_1,...,\alpha_n\}$, we construct $G(\alpha)$, $u(\alpha)$, and $v(\alpha)$ such that

$\alpha \in$ MCV iff UNIFY($G(\alpha),u(\alpha),v(\alpha)$) = UNIFIABLE.

This reduction directly applies to UNIFY^{∞} and is easily seen to use only log space.

The monotone circuit α can be represented as a diagram with wires, AND and OR gates of fan-in 2 and fan-out at most 2, a special OR output gate with one output wire, and with each input wire leading to one gate and having a 0 or a 1 value (see Figure 3a for an example). The input wire values combine to produce values for all other wires and the output wire in particular. The circuit has no feedback, i.e., if the wires are viewed as arcs and the inputs and gates as nodes we get a dag without multiple arcs.

(1) Introduce two nodes $u(\alpha)$, $v(\alpha)$ in $G(\alpha)$.

(2) If α_i is an AND gate include G_{and} from Figure 4a in $G(\alpha)$. If α_i is an OR gate include G_{or} from Figure 4b in $G(\alpha)$. These dags have two pairs of input nodes and one pair of output nodes each, i.e., $\{IN_{1i}, IN_{2i}\}$, $\{IN_{3i}, IN_{4i}\}$, and $\{OUT_{1i}, OUT_{2i}\}$. Corresponding sons are illustrated by the labels a, b on the arcs.

(3) If α_i is an input include in $G(\alpha)$ a pair of nodes $\{OUT_{1i}, OUT_{2i}\}$. If the value of the input is 1 then make OUT_{1i}, OUT_{2i} corresponding sons of $u(\alpha)$, $v(\alpha)$. If the value of the input is 0 then make OUT_{1i}, OUT_{2i} sons of $u(\alpha)$ and let $v(\alpha)$ have two sons that correspond to them and are two new leaves in $G(\alpha)$.

(4) If gate α_i is connected to α_j, α_k (i.e., in the wire diagram) then identify nodes $IN_{1i}=OUT_{1j}$, $IN_{2i}=OUT_{2j}$, $IN_{3i}=OUT_{1k}$, $IN_{4i}=OUT_{2k}$. When these subdags are concatenated nodes have outdegree ≤ 2, and the labels on the arcs can be made 1 and 2, so that the equalities of labels a, b in Figures 4a, 4b is preserved.

(5) In the dag constucted in steps 1-4 above assign labels to the nodes as follows:
label(u) = label(v) = h,
label(node of outdegree 1) = g,
label(node of outdegree 2) = f,
label(OUT_{1n}) = $g_1 \neq g_2$ = label(OUT_{2n}),
label(leaf other than OUT_{1n}, OUT_{2n}) = distinct variable.

We can easily see now that every wire w in the wire diagram can be associated to a pair of nodes OUT_w, $OUT_{w'}$. We require $u(\alpha)$®$v(\alpha)$. For such a minimal c-e relation ®, we claim that the value of wire w in α is 1 iff OUT_w® $OUT_{w'}$. This certainly holds for the inputs, because of the way we built corresponding sons of u and v. Also, it is trivial to check that G_{and} and G_{or} simulate the behavior of AND and OR gates. Therefore the value of α_n is 1 iff OUT_{1n}®OUT_{2n}. The graph $G(\alpha)$ is constructed in such a way that the only place homogeneity could be violated by ® is if OUT_{1n}®OUT_{2n}. As a result, if $\alpha=1$, the terms represented by $u(\alpha)$ and $v(\alpha)$ are not unrestricted unifiable, and if $\alpha=0$ they are unifiable (the acyclicity condition is also true). □

4. Simple Dags and Congruence Closure

In this Section we will make our lower bounds independent of the potentialy concise dag representation of terms, by extending them to simple dags. We will also investigate the related problem of computing the congruence closure of an equivalence relation.

Theorem 2: UNIFY(G,u,v) and UNIFY^{∞}(G,u,v) are log-space complete for P, even when G is a simple dag.

Proof: Given monotone circuit α we construct a simple dag $G(\alpha)$ with two roots $u(\alpha)$, and $v(\alpha)$ so that, if $\alpha_n = 0$ then the terms $t_{u(\alpha)}$, $t_{v(\alpha)}$ are unifiable else they are not unrestricted unifiable. This suffices for the completeness of both UNIFY and UNIFY^{∞}. Note that the proof of Theorem 1 no longer applies, because the G_{or} dags used in that reduction could introduce nodes with indegree 2, i.e., their output nodes, which were not leaves.

As in the proof of Theorem 1, we encode the input of α using a pair of nodes for each circuit input. The input-subgraph of the graph of Theorem 1 is actually a simple dag, so we use the same construction. However, we cannot attach "gates" directly to the input-subgraph since this will produce a dag which is not simple. Instead, each gate will be constructed separately using a pair of subgraphs. Any c-e relation $\circledR$ with $u(\alpha) \circledR v(\alpha)$ will relate the two parts of each gate. In addition, the input nodes of one gate will be "connected" to input-subgraph nodes or output nodes of other gates using a separate "patch board" subgraph. Recall that the gates of α are numbered so that if an output of gate α_i goes to an input of gate α_j, then $i<j$.

For each gate of α, we use four input nodes and four output nodes. For gate α_i, let us denote these nodes by $IN_{1i}, \ldots, IN_{4i}$ and $OUT_{1i}, \ldots, OUT_{4i}$. As in the proof of Theorem 1, the nodes of $G(\alpha)$ work in pairs. Inputs IN_{1i} and IN_{2i} represent the first input to α_i and IN_{3i} and IN_{4i} the second. Similarly, nodes OUT_{1i} and OUT_{2i} represent the first output of α_i and OUT_{3i} and OUT_{4i} the second. We also use nodes u_i, v_i which are the i-th sons of roots $u(\alpha)$ and $v(\alpha)$, respectively, and four or seven internal nodes which may remain anonymous.

If α_i is an OR-gate, then we construct a simple dag $GATE_i$ as in Figure 5a, with u_i, v_i corresponding sons of $u(\alpha)$, $v(\alpha)$. If $\circledR$ is a c-e relation with $u(\alpha) \circledR v(\alpha)$, it is easy to see that $OUT_{1i} \circledR OUT_{2i}$ and $OUT_{3i} \circledR OUT_{4i}$ if either $IN_{1i} \circledR IN_{2i}$ or $IN_{3i} \circledR IN_{4i}$. It will be clear from the construction of $G(\alpha)$ that if $\circledR$ is minimal, then these are the only cases in which the output nodes will be related by $\circledR$. If α_i is an AND-gate, exactly similar reasoning applies for the simple dag of Figure 5b, which simulates the logic of AND.

The remaining task is to "connect" the gates so that if, for example, the first output of α_i goes to the second input of α_j, then $IN_{3j} \circledR IN_{4j}$ whenever $OUT_{1i} \circledR OUT_{2i}$. We use an example connection between α_i and α_j to illustrate the construction of a "patch board" simple dag PATCH, which contains two new nodes u_p, v_p and IN and OUT nodes from the input-subgraph and gate subgraphs of $G(\alpha)$. Let u_p, v_p be corresponding sons of $u(\alpha)$, $v(\alpha)$, different from the sons used in the gate and input subgraphs. Now make IN_{3j} and OUT_{1i} corresponding sons of u_p and v_p; also make IN_{4j} and OUT_{2i} corresponding sons of u_p and v_p (see Figures 5c and 5d). When $u(\alpha)$®$v(\alpha)$, two input nodes of $GATE_j$ will be merged if the right two output nodes of $GATE_i$ are.

As in the proof of Theorem 1, we label the outputs of the final gate with different constant symbols. All other nodes have labels that depend on their arity, so that nodes with outdegree 2, say, have the same label. It is easy to verify by induction that in the minimal c-c relation ® with $u(\alpha)$®$v(\alpha)$, we have OUT_{1n}®OUT_{2n} and OUT_{3n}®OUT_{4n} iff the output of the last gate α_n is 1. This completes the proof of Theorem 2.□

Congruence closure is a practical problem that is in many ways a dual to unification. In unification, the equivalence classes of m and n are merged whenever *there exist* some equivalent r and s such that m and n are corresponding sons of r and s. In congruence closure, the equivalence classes of r and s are merged whenever, *for all* pairs of corresponding sons m, n, we have that m is equivalent to n. We consider a pure form of congruence closure in which the node labels are ignored, however, the arcs must still be labeled so that we can see which sons correspond. Efficient algorithms for congruence closure are contained in [DST].

An equivalence relation © on the nodes of a labeled graph is a *congruence* relation when: if u,v have same outdegree and for each pair of corresponding sons we have u_i©v_i, then u©v. Given any equivalence relation R, there is a unique minimal congruence relation that contains R, called the *congruence closure* of R. An equivalence relation R can be represented using undirected edges in a labeled dag. We can now pose the following language recognition problem: CONG={<G,u,v,R>| nodes u and v of labeled dag G are related by the congruence closure of R}.

By using a construction that resembles that of Theorem 1 "turned upside-down" and that exhibits an and/or duality between unification and congruence closure we can show that:

Theorem 3: The language CONG is complete for P.

Proof: Again we reduce MCV to CONG. The wires of the circuit diagram correspond to pairs of nodes, such that, the two nodes are related in © (the congruence closure) iff the value on the wire is 1. Given a monotone circuit α we construct a dag $G(\alpha)$, an equivalence relation $R(\alpha)$ on its nodes, and two roots of the dag $u(\alpha)$, $v(\alpha)$. We wish to test the two roots for equivalence in ©, the congruence closure of $R(\alpha)$.

The construction is bottom-up, so that each circuit input corresponds to a pair of leaves and each OR and AND gate to a subgraph with two pairs of input and two pairs of output nodes each; the last OR-subgraph has only one pair of output nodes $u(\alpha)$, $v(\alpha)$. The inputs and outputs are connected in a pattern similar to that of the proof of Theorem 1. The input leaves are represented in Figure 6a, note that for inputs that are 1 the two leaves are in the same equivalence class of $R(\alpha)$. The OR-subgraph is in Figure 6b, and two pairs of internal nodes form equivalence classes of $R(\alpha)$. The AND-subgraph is in Figure 6c. It is simple to verify that the gate subgraphs simulate the gate logic and that $u(\alpha)$©$v(\alpha)$ iff the output of the circuit α is 1. □

5. A Parallel Algorithm for Term Matching

Unification is a practical sequential algorithm for matching since unification can be done in linear time. However, unification is not a good parallel approach to matching. We show how MATCH(G,u,v) can be computed in $\log^2 n$ parallel time using polynomially many processors. In addition, we prove that MATCH(G,u,v) is in co-NLOGSPACE. If G is a simple dag then MATCH(G,u,v) is actually in DLOGSPACE.

When we wish to determine whether s matches t, we will assume w.l.o.g. that no variables appear in t. In Section 6 we further clarify the relationship between matching and unification. Since MATCH(G,u,v) is the same as UNIFY(G,u,v) when no variables appear in t_v, we know that t_u matches t_v iff there is a homogeneous c-e relation ~ on G with u~v. A refinement of this characterization of term matching suggests an efficient parallel algorithm.

Lemma 1: Let G be a labeled dag with nodes u and v, and let the subgraph of G induced by the descendants of v have no nodes labeled with variables. Let R be the minimal correspondence relation on G with uRv, S be the minimal equivalence relation containing R, and T be the minimal correspondence relation containing S. Then t_u matches t_v iff T is homogeneous.

Proof: If t_u matches t_v then since t_u and t_v are unifiable, the minimal c-e relation ~ with u~v is homogeneous. Since ~ must contain T, it follows that T is homogeneous.

For the converse, suppose that T is homogeneous. We will define a substitution σ such that $\sigma(t_u) = t_v$. Let G_u, G_v be the subgraphs of descendants of u,v respectively. We first show that for every node x in G_u there is a node y in G_v such that xRy. If, on the contrary, there is some x in G_u without xRy for any y in G_v, then let w be the last node in some path from u to x with wRz for some z in G_v. Since w has a son, label(w) is a k-ary function symbol for some k>0. By similar reasoning, label(z) is a zero-ary function symbol. But then label(w)≠label(z) and hence T is not homogeneous. It follows from this contradiction that every S-equivalence class contains at least one node from G_v.

For each S-equivalence class E, pick some node e from G_v. If w is another G_v node in E, then since T is homogeneous and no variables appear in G_v, we can argue that $t_w = t_e$ (here we have the problem EQUAL). We now define the substitution σ. For any variable x in t_u, let E be the S-equivalence class of the node labeled x and define $\sigma(x) = t_e$. It is easy to check by induction on the height of a node w in G_u that if wRz, then $\sigma(t_w) = t_z$. Thus $\sigma(t_u) = t_v$ and t_u matches t_v.□

Given any relation, we can find the minimal correspondence relation R containing it, in $\log^2 n$ parallel time and $n^{O(1)}$ processors on a PRAM, using a transitive closure algorithm [Ch]. If G is a labeled dag with n nodes, we define an n^2 by n^2 boolean *correspondence matrix* C_G. We associate each (unordered) pair of nodes of G with a row and a column of C_G and define the entries of C_G:

$C_G(\{u, v\}, \{x, y\}) = 1$ iff x and y are u and v or corresponding sons of u and v.

Lemma 2: Let G be a labeled dag with nodes u and v, and let R be the min. correspondence relation s.t. uRv. Then xRy iff the $(\{u, v\}, \{x, y\})$ entry of C_G's transitive closure equals 1. □

Now given relation R, we can find the minimal equivalence relation S containing R using a connected components algorithm. It is well-known that connected components can be computed in $\log^2 n$ parallel time and $n^{O(1)}$ processors on a PRAM [CHS].

Since computing correspondence relations twice, connected components once and testing for homogeneity are sufficient to decide matching, we have that MATCH(G,u,v) can be computed in $\log^2 n$ parallel time and $n^{O(1)}$ processors on a PRAM (or equivalently MATCH $\in$ NC).

In fact, we can show somewhat tighter complexity upper bounds, since DLOGSPACE $\subseteq$ NLOGSPACE $\subseteq$ NC:

Theorem 4: The set of $\langle G,u,v \rangle$ such that MATCH(G,u,v) = *false* is in NLOGSPACE. Furthermore, if G is a simple dag, then this recognition problem is in DLOGSPACE.

Proof: Let G be a dag with MATCH(G,u,v)=*false*. Let R,S,T be relations on the nodes of G as in the statement of Lemma 1. By Lemma 1, there must be nodes x and y of G such that xTy, but label(x) and label(y) are two different function symbols. We show that there is a log-space bounded nondeterministic Turing machine M_T capable of guessing all pairs (x,y) such that xTy, and checking whether x and y have the same labels. Thus recognizing the $\langle G,u,v \rangle$'s, such that, MATCH(G,u,v) = *true* is a problem in co-NLOGSPACE (also a subset of the class NC).

To begin with, let M_R be a nondeterministic machine that starts with the pair (u,v) on its worktape. A move of M_R consists of replacing a pair (x,y) with a pair (x_i,y_i) of corresponding sons of x and y. Clearly M_R is capable of guessing (x,y) iff xRy.

We now define a nondeterministic machine M_S using M_R. The machine M_S begins by running M_R some nondeterministic number of steps to guess a pair (x,y). Subsequently, M_S repeats the following 3 steps nondeterministically:
(1) If one pair (x,y) or two pairs (x,y) (w,z) are on the worktape, then it may replace (x,y) by (y,x).
(2) If (x,y), (y,z) are on the worktape, then it may replace both by the single pair (x,z).
(3) If the single pair (x,y) is on the worktape, then it may run M_R some number of steps to guess (w,z) and end up with both pairs (w,z), (x,y) on the worktape.

With these primitive steps M_S may guess (x,y) iff xSy.

Finally, we build M_T from M_S. This machine behaves just like M_R, but instead of starting with (u,v), starts with any pair (x,y) that M_S is capable of guessing. This concludes the proof of the first part of the theorem, which in a way describes the PRAM algorithm sketched above, but from the point of view of nondeterministic log-space.

If G is a simple dag, then M_R can easily be made a deterministic depth-first enumerator of pairs (x,y). This machine MD_R always maintains the pair immediately preceeding the current one, so that it can backtrack from leaf nodes. Backtracking from internal nodes is staightforward since each has indegree 1.

Using a log-space preprocessor we can treat the subgraph rooted at v as a tree. Recall that this graph has no variables, so that all we need to do is duplicate leaves labeled with constants. By doing this we limit the number of times step (2) of M_S must be repeated to only two. Thus we can construct a deterministic machine MD_S that enumerates all (x,y) such that xSy. Finally, we build a deterministic MD_T from MD_R and MD_S as before. □

A corollary of Theorem 4 is that for simple dags deciding whether MATCH(G,u,v) = *true* is also in DLOGSPACE, since DLOGSPACE is closed under complement. From the analysis in [FW] it also follows that this problem can be solved in O(log n) parallel time on a PRAM.

6. Conclusions and Open Problems

We have demonstrated that several versions of unification are complete for P. This suggests, by way of the parallel computation thesis, that unification is inherently sequential. It is unlikely that significant improvements in the speed of theorem provers, interpreters for logic programs, and the like will be brought about by the development of parallel unification algorithms. However, for the special case of term matching, the prospects are much brighter. Term matching can be accomplished in log n or $\log^2 n$ parallel time, depending on whether the input is in the form of a simple dag.

We might also point out that unification of terms s and t is complete for P even if s and t do not contain any variables in common (this is different from t having no variables). Also, if s and t are unifiable this does not imply that s matches t or that t matches s. However, if s matches t then s and t are unrestricted unifiable. If s matches t and t matches s they are unifiable.

Intuitively, congruence closure appears to be a "dual" of unification. It, too, is complete for P. As a consequence, various congruence closure problems, such as the decision problem for the first-order quantifier-free theory of equality [DST] are not conducive to extremely fast parallel solutions.

There are remarkable similarities between the sequential algorithms for unification and testing equivalence of deterministic finite automata. However, the *inequivalence* of deterministic finite automate can be detected nondeterministically using only logarithmic space. A machine can see that two automata A_1 and A_2 are equivalent by guessing an input string, character by character and simulating the actions of both machines as it goes. If one ends up in an accept state while the other rejects, then the two are clearly different. If A_1 and A_2 differ, then some sequence of characters must surely uncover this. Thus unification is subtly, but fundamentally different from this "almost identical" problem.

Some interesting open problems remain unresolved, namely; (1) lower bounds for the complexity of MATCH and EQUAL, or can our upper bounds be improved, (2) the number of processors used in the transitive closure of a correspondence matrix is unrealistically large, and it would be of some practical significance to decrease it to even n^3, and finally (3) what is the complexity of *commutative* matching, i.e., if function symbols stand for commutative operations.

References

[C] Cook, S.A., "An Overview of Computational Complexity", *CACM* 26(6), 1983, pp 400-409.

[Ch] Chandra, A.K., "Maximal Parallelism in Matrix Multiplication", IBM report, RC 6193, 1976.

[CM] Clocksin, W.F., Mellish, C.S., "Programming in Prolog", Springer-Verlag, 1981.

[DST] Downey, P.J., Sethi, R., Tarjan, R.E., "Variations on the Common Subexpression Problem", *JACM* 27(4), 1980, pp 758-771.

[FW] Fortune, S., Wyllie, J., "Parallelism in Random Access Machines", *Proc 10th ACM STOC*, pp 114-118.

[G1] Goldschlager, L.M., "The Monotone and Planar Circuit Value Problems are Log Space Complete for P", *SIGACT News* 9(2), 1977, pp 25-29.

[G2] Goldschlager, L.M., Shaw, R.A., Staples, J., "The Maximum Flow Problem is Log Space Complete for P", *TCS* 21, 1982, pp 105-111.

[GKM] Guttag, J.V., Kapur, D., Musser, D.R., "On Proving Uniform Termination and Restricted Termination of Rewriting Systems", *Siam J. Computing* 12(1), 1983, pp 189-214.

[HCS] Hirschberg, D.S., Chandra, A.K., Sarwate, D.V., "Computing Connected Components on Parallel computers", *CACM* 22(8), 1979, pp 461-464.

[J] Johnson, D.S., "The NP-Completeness Column: An Ongoing Guide", *J. of Algorithms* 4, 1983, pp 189-203.

[LS] Lewis, H.R., Statman, R., "Unifiability is Complete for co-NLOGSPACE", *IPL* 15(5), 1982, pp 220-222.

[M] Milner, R., "A Theory of Type Polymorphism in Programming", *JCSS* 17, 1978, pp 348-375.

[MM] Martelli, A., Montanari, U., "An Efficient Unification Algorithm", *ACM Trans. on Programming Languages and Systems*, 4(2), 1982.

[MPS] MacQueen, D., Plotkin, G., Sethi, R., "An Ideal Model for Recursive Polymorphic Types", *Proc. 1984 ACM POPL*, to appear.

[PW] Paterson, M.S., Wegman, M.N., "Linear Unification", *JCSS* 16, 1978, pp 158-167.

[R] Robinson, J.A., "A Machine Oriented Logic Based on the Resolution Principle", *JACM* 12(1), 1965, pp 23-41.

[S] Shapiro, E.Y., "A Subset of Concurrent Prolog and its Interpreter", ICOT report TR-003, Tokyo, JAPAN, 1983.

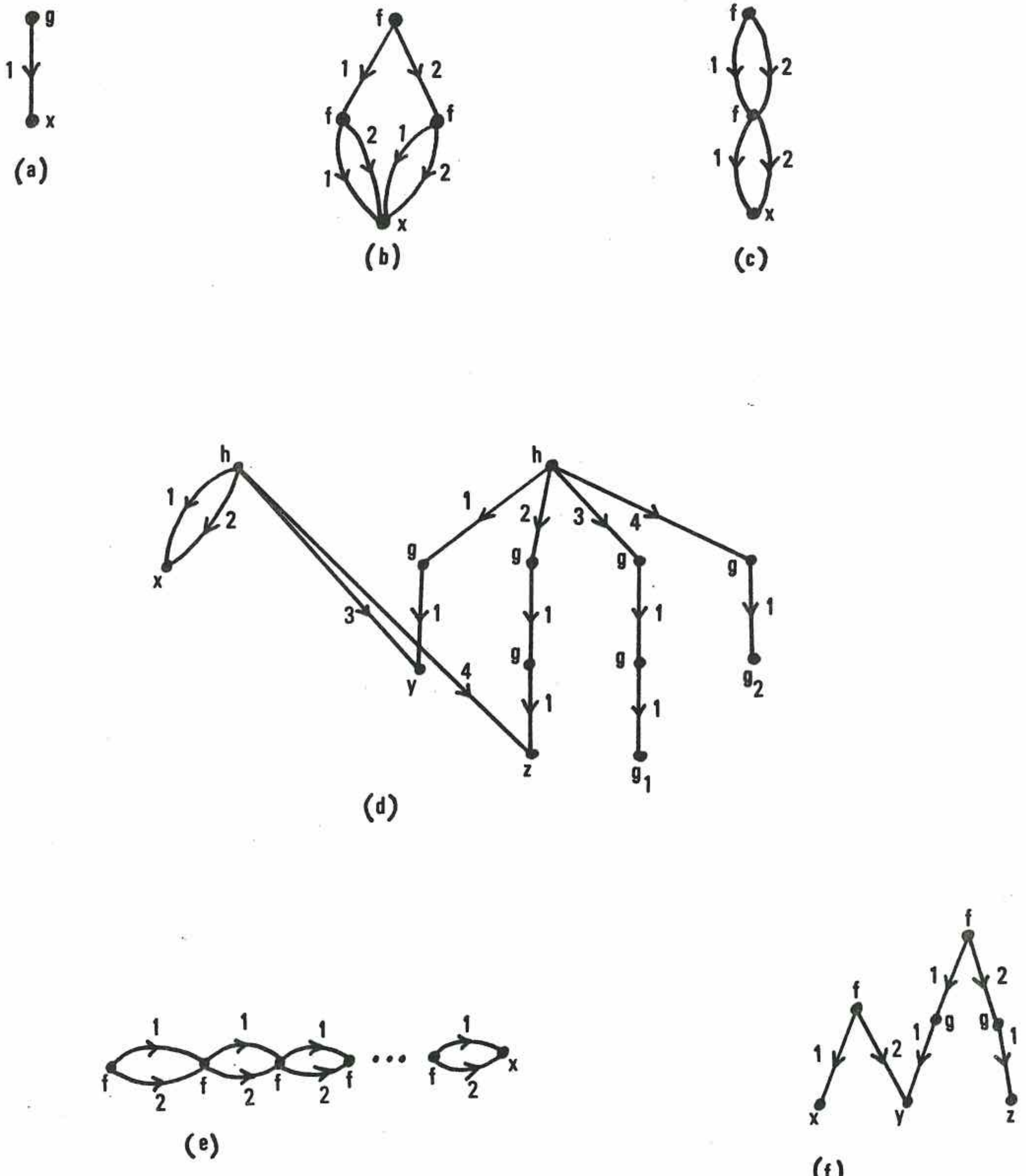
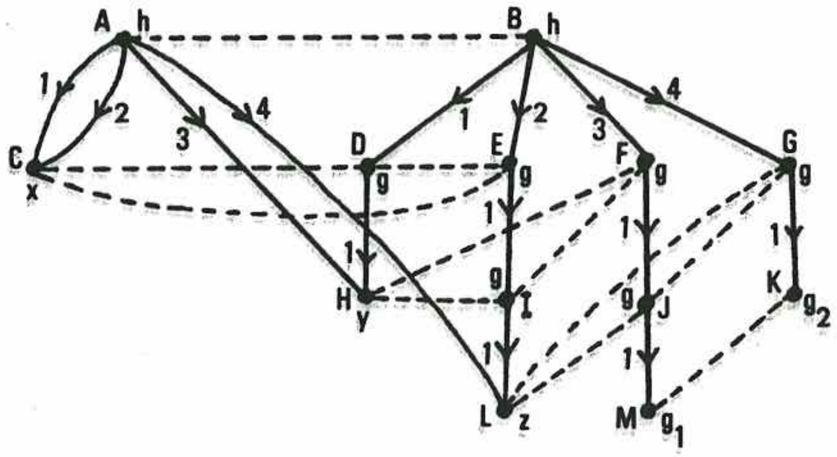
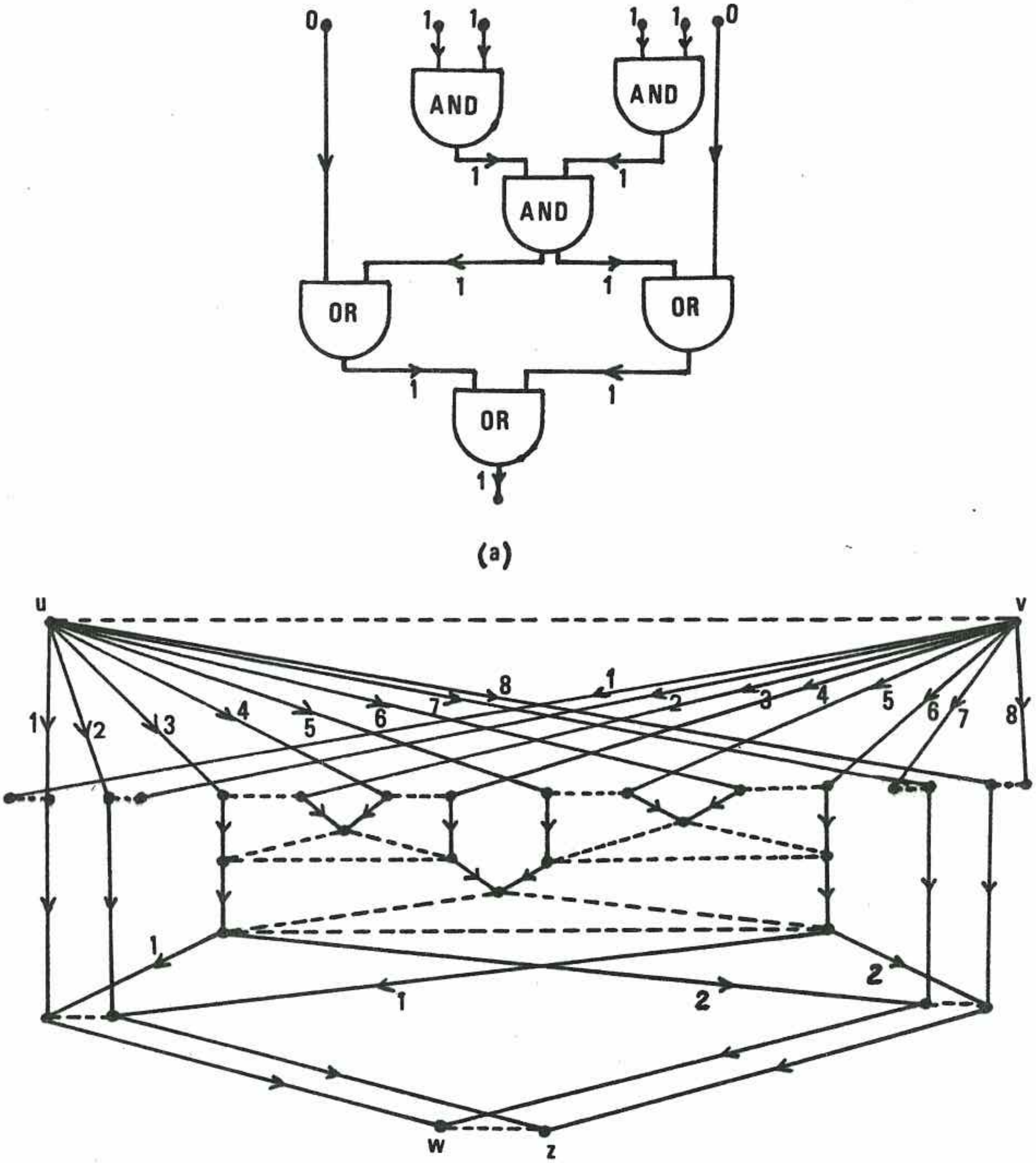

g
1
x
(a)
f
1
2
f
f
2
1
1
2
x
(b)
f
1
2
f
1
2
x
(c)
h
1
2
x
3
4
y
z
h
1
2
3
4
g
g
g
g
1
1
1
1
g
g
1
1
g_2
g_1
(d)
1
1
1
1
f
2
f
2
f
2
f
f
2
x
...
(e)
f
f
1
2
1
2
g
g
1
1
x
y
z
(f)

Figure 1: labeled dags


A h
B h
1
2
3
4
C
x
D g
E g
F g
G g
1
H
y
I g
J g
K g_2
L z
M g_1

1. A®B
2. C®D, C®E, H®F, L®G (propagation)
3. D®E (transitivity)
4. H®I (propagation)
5. F®I (transitivity)
6. L®J (propagation)
7. J®G (transitivity)
8. M®K (propagation)
9. ununifiable because M and K have distinct labels g_1 and g_2.

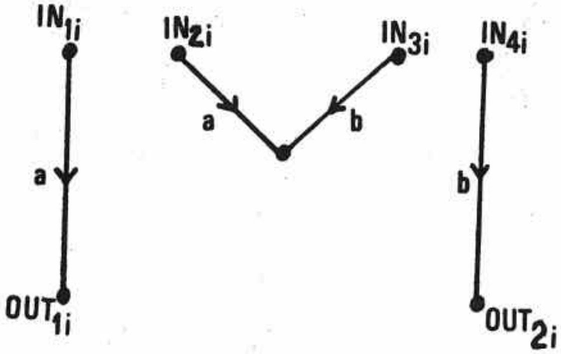
Figure 2: illustrating naive-unification


0
1
1
1
1
0
AND
AND
AND
1
1
1
1
OR
OR
1
1
OR
1
u
v
1
2
3
4
5
6
7
8
1
2
3
4
5
6
7
8
1
1
2
2
w
z

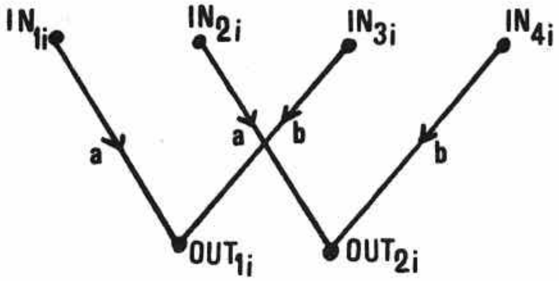
(a)

label(u) = label(v) = h, label(w) = $g_1 \neq g_2$ = label(z)
label(node of outdegree 1) = g
label(node of outdegree 2) = f
label(leaf other than w, z) = distinct variable
(b)

Figure 3

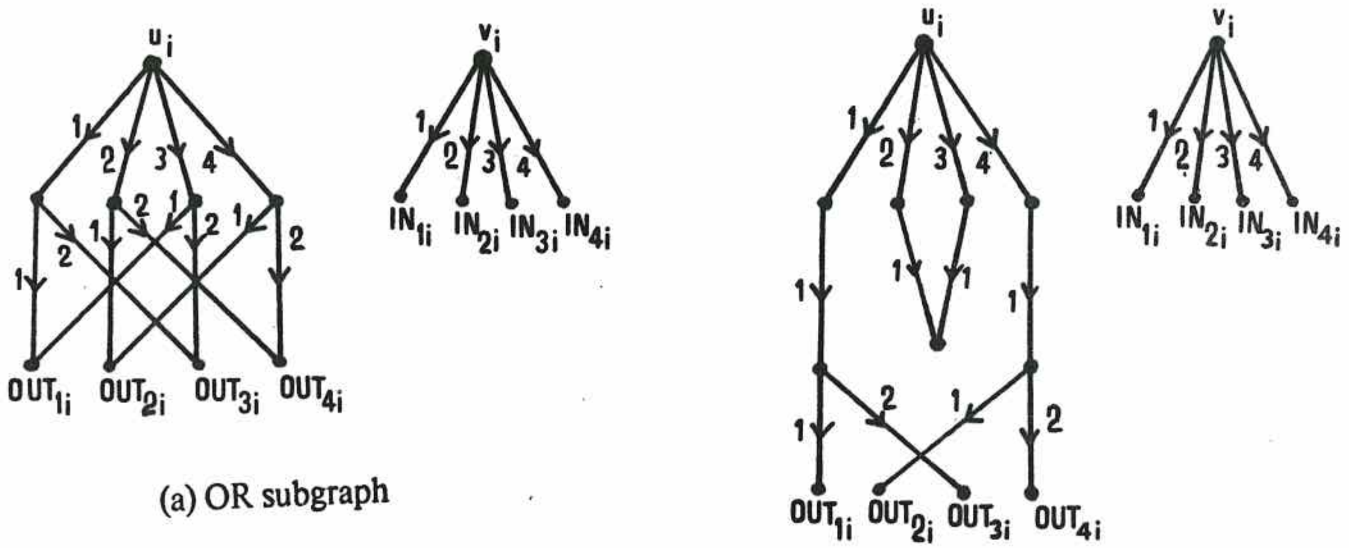

IN_{1i}
IN_{2i}
IN_{3i}
IN_{4i}
a
a
b
b
OUT_{1i}
OUT_{2i}

(a) G_{and}


IN_{1i}
IN_{2i}
IN_{3i}
IN_{4i}
a
a
b
b
OUT_{1i}
OUT_{2i}

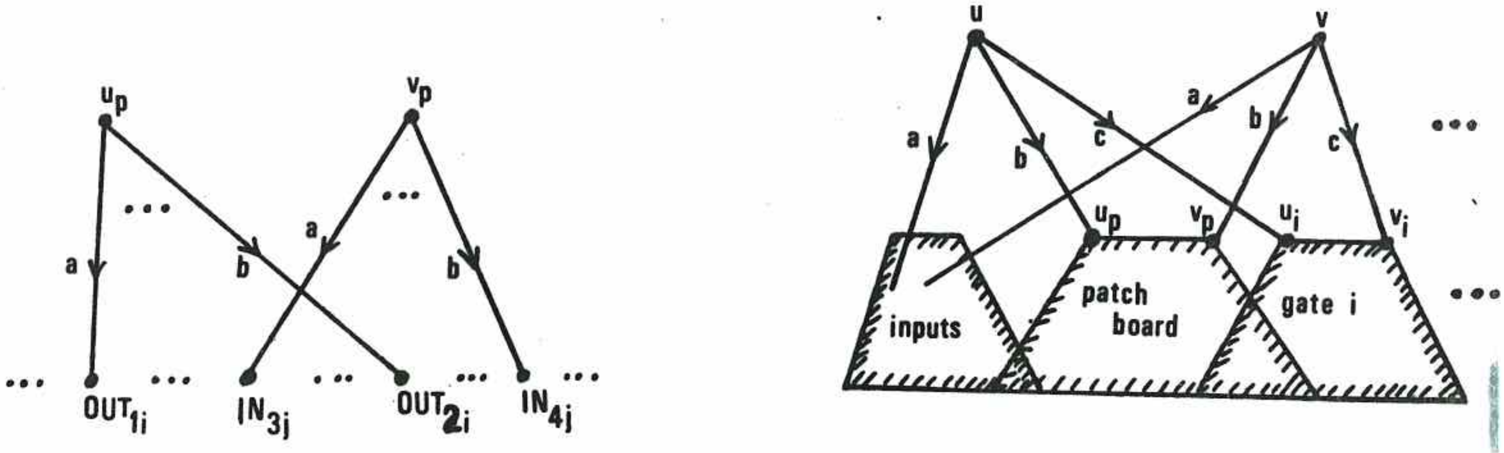
(b) G_{or}

Figure 4: Theorem 1 subgraphs


u_i
1
2
3
4
2
1
2
1
1
2
1
2
OUT_{1i}
OUT_{2i}
OUT_{3i}
OUT_{4i}
v_i
1
2
3
4
IN_{1i}
IN_{2i}
IN_{3i}
IN_{4i}
u_i
1
2
3
4
1
1
1
1
1
2
1
2
OUT_{1i}
OUT_{2i}
OUT_{3i}
OUT_{4i}
v_i
1
2
3
4
IN_{1i}
IN_{2i}
IN_{3i}
IN_{4i}

(a) OR subgraph

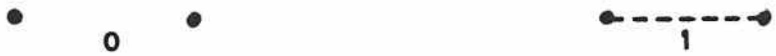
(b) AND subgraph


u_p
v_p
a
b
a
b
OUT_{1i}
IN_{3j}
OUT_{2i}
IN_{4j}
u
v
a
b
c
a
b
c
u_p
v_p
u_i
v_i
inputs
patch board
gate i

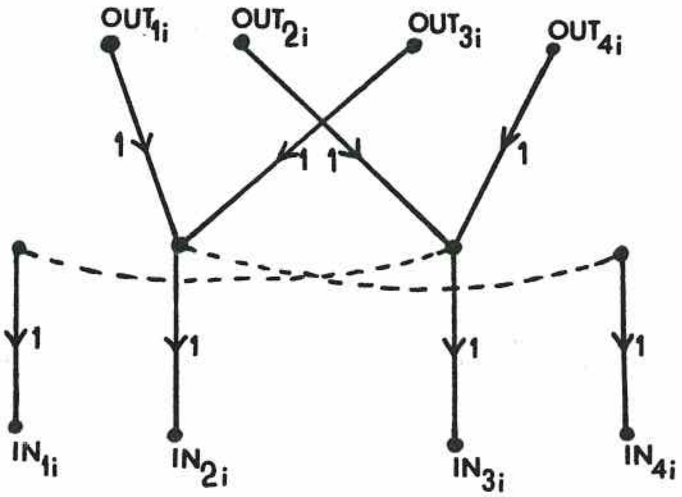
(c) example use of PATCH

(d) putting everything together

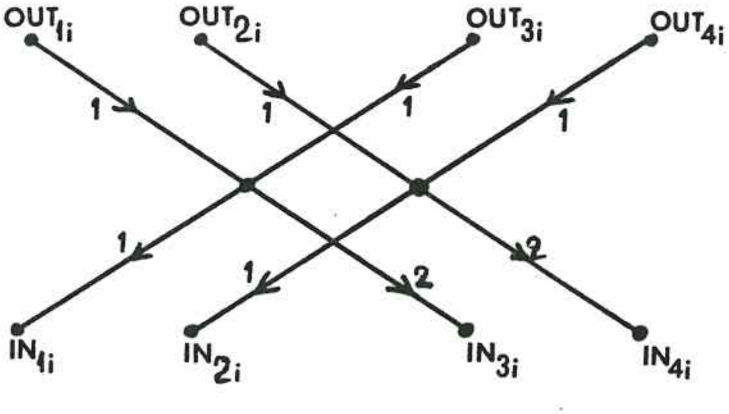
Figure 5: Theorem 2 subgraphs



(a) inputs


OUT1i
OUT2i
OUT3i
OUT4i
1
1
1
1
1
1
1
1
IN1i
IN2i
IN3i
IN4i

(b) OR subgraph


OUT1i
OUT2i
OUT3i
OUT4i
1
1
1
1
1
1
2
2
IN1i
IN2i
IN3i
IN4i

(c) AND subgraph

Figure 6: Theorem 3 subgraphs
(R is denoted by ---------)