Nondestructive Graph Unification

David A. Wroblewski
MCC
3500 West Balcones Center Drive
Austin, Texas 78759

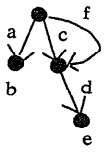
Abstract

Graph unification is sometimes implemented as a destructive operation, making it neccesary to copy the argument graphs before beginning the actual unification. Previous research on graph unification claimed that this copying is a computation sink, and has sought to correct this.

In this paper I claim that the fundamental problem is in designing graph unification as a destructive operation. This forces it to both *over copy* and *early copy*. I present a nondestructive graph unification algorithm that minimizes over copying and eliminates early copying. This algorithm is significantly simpler than recently published solutions to copying problems, but maintains the essential efficiency gains of older techniques.

I. Directed Acyclic Graphs

In this paper I will deal with unification of rooted, connected, acyclic graphs, or "DAGs". The efficiency of graph unification has recently received attention because of the popularity of graph-unification-based formalisms in computational linguistics [Karttunen 86], [Shieber 84], [Wittenburg 86], [Periera 85], [Karttunen and Kay 85]. In these parsers, the lexical entries and grammar rules are represented as DAGs, and graph unification is the mechanism whereby rules are applied to sentence constituents. Unfortunately, graph unification is an expensive process; any attempt to build a practical parser based on graph unification must address the issue of making it efficient. Past research has identified the copying involved in graph unification as a computational sink. This paper presents a graph unification algorithm that does not need to copy its argument graphs because it is nondestructive.




```
[a: b
 c: 1 [d:e]
 f: <1>]
```

Figure 1: The Graph Matrix Notation

For the purposes of discussion, I will use the matrix notation (whenever possible) for graphs as used in [Wittenburg 86], [Shieber 84], and which has become somewhat of a *de facto* standard in the field. In this notation, reentrant structures are indicated with a *mark*, a number preceding a subgraph, and a *pointer*, a number enclosed in "<" and ">". When a pointer of the form <n> is encountered, it should be interpreted as: "The following graph is *the very same graph* as the one marked by the number *n* elsewhere in the graph." Figure 1 shows a graph in the matrix notation with a reentrant pointer, marked and pointed to by the number 1. Beside it is an equivalent picture of the same graph.

Abstractly, a DAG consists of a set of nodes, a set of arcs, and a special node designated the *root*. In practice, we implement DAGS with structures that are analogical; there is a *node* structure and an *arc* structure as shown in Figure 2[1].

Node Structure

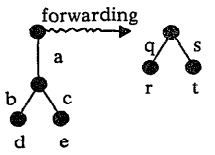
```
+--------------+
|  forward     |
+--------------+
|  arc list    |
+--------------+
```

Arc Structure

```
+--------------+
|  label       |
+--------------+
|  value       |
+--------------+
```

Figure 2: The Implementation Of DAGs

Finally, in our implementation, nodes in a DAG can be *forwarded* to other nodes. For instance, the DAG graphically described in Figure 3 shows the root node forwarded to another node; beside it is the result of printing this graph in the matrix notation. Note that the contents of the forwarded node are completely ignored. For all purposes, the node being forwarded has been discarded and replaced with node B. It is important that all operations on graphs must honor a forwarding pointer above all else; forwarding is the highest priority operation. The process of resolving these forwarding pointers (there may be chains of them) is known as "dereferencing" a node [Periera 85].




```
[q: r
 s: t]
```

Figure 3: Forwarding Links Override Node Definitions

[1]Our implementation of the destructive unification algorithm presented in Section 3 and the structures shown in Figure 2 owes much to the work of Shieber [Shieber 84], Karttunnen [Karttunen 84], and Wittenburg [Wittenburg 86]

II. Graph Unification

Conceptually, the algorithm for graph unification is quite simple, and much like ordinary term unification used in theorem-proving programs [Warren 83], [Boyer and Moore 72], extended to the more complex structure of DAGs. Like term unification, graph unification both matches the argument DAGs and builds a new DAG. If the DAGs to be unified are somehow incompatible, then unification produces nothing; if the DAGs are compatible then unification returns a new DAG which is compatible with, and more specific than, both of the argument DAGS.

```
[a: [b: c]  + [a: 1[b: c] ==> [a: 1[b: c
 d: [e: f]]    d: <1>             e: f]
               g: [h: j]]      d: <1>
                               g: [h: j]]
```

Figure 4: The Successful Unification Of Two DAGs

For instance, Figure 4 shows two DAGs and the result of unifying them. Figure 5 shows two pairs of incompatible DAGs, i.e. two DAGs over which unification would fail.

```
[c: d]      +  [c: e]       ==>  Failure!

[a: 1[x:y] +  [a: [c:d]     ==>  Failure!
 e: <1>]       e: [c:e]]
```

Figure 5: The Unsuccessful Unification of Two DAGs

III. Destructive Graph Unification

The following version of the destructive unification algorithm is taken, with modification, from [Periera 85]. It takes as input two nodes, initially the roots of the DAGs to be unified. It recurses on the subgraphs of each DAG until an "atomic" arc value is found[2].

`Unify1` assumes the existence of two utility functions. `complementarcs(d1,d2)` takes two nodes as input and returns the arc labels that are unique to `d1` with respect to `d2`. `intersectarcs(d1,d2)` takes two nodes as input and returns the arc labels that exist in both d1 and d2. These operations are equivalent to the set complement and set intersection, respectively, of the set of arc labels for each node.

`Unify1` modifies its argument DAGs in two ways. First, all the nodes from one DAG are forwarded to the other. Subsequent operations on either of these DAGs will always dereference either node to the same structure. Second, arcs may be added to the `d2` nodes,

```
PROCEDURE Unify1 (d1 d2)
  Dereference d1 and d2.
  IF d1 and d2 are identical THEN
    success: return d1 or d2.
  ELSE
    new = complementarcs(d1,d2).
    shared = intersectarcs(d1,d2).
    Forward d1 to d2.
    FOR each arc in shared DO
      Find the corresponding arc in d2.
      Recursively unify1 the arc-values.
      IF unify1 failed THEN
        Return failure
      ELSE
        Replace the d2 arc value
          with the result.
      ENDIF
    FOR all arcs in new DO
      Add this arc to d2.
  Return d2 or d1 arbitrarily.
  ENDIF
ENDPROCEDURE
```

as in the arc labelled `g` in Figure 4. Finally, note that the order in which `shared` arcs are unified is a nondeterministic choice.

Since `Unify1` ravages its argument DAGs, they must be copied before it is invoked if the argument DAGs need to be preserved. For instance, if a grammar rule is represented as a DAG, then it surely should not be permanently changed during the application of the rule to a DAG representing a sentence constituent. Thus the rule DAG must be copied before the application of the rule.

IV. Issues in Graph Unification

Previous research [Karttunen and Kay 85], [Periera 85] has identified DAG copying as significant overhead. However, *some* amount of copying must be done to create the result DAG. Exactly when is copying wrong, then? The answer is: when the algorithm copies too much or copies too soon. Destructive unification makes both of these mistakes. They are named:

- **Over Copying.** Copies are made of *both* DAGs, and then these copies are ravaged by the unification algorithm to build a result DAG. This would appear to require the raw materials for *two* DAGs in order to create just one new DAG. A better algorithm would only allocate enough memory for the resulting DAG.
- **Early Copying.** The argument DAGs are copied *before* unification is started. If the unification fails, then some of the copying is wasted effort. A better algorithm would copy incrementally, so that if a failure occurred, only the minimal copying would be done before the failure was detected.

The important point here is that these are two distinct features of a unification algorithm, and have no

[2] In this paper, atomic cases are not considered, since they are trival. An actual implementation of this would include a type check for atomic DAGs and clause testing for atomic equality if so. This has been left out for clarity.

neccesary connection. Previous attempts to improve unification have not acknowledged their independence, and are perhaps more complicated than they need to be because of that. On the other hand, the algorithm presented in the next section deals completely with early copying but only partially with over copying; they are treated as independent problems.

V. Nondestructive Graph Unification

In this section, I present a nondestructive graph unification algorithm that incrementally copies its argument graphs. It avoids, whenever possible, over copying, and completely eliminates early copying. The price to be paid for this is a slightly more complicated algorithm and graph representation.

Intuitively, unification could be nondestructive if it were to build the result DAG as it proceeds, making all changes in this new DAG, and leaving the argument DAGs untouched. This means that we will have to associate with each component of the argument DAGs, its copy. If, at each step during unification, we return the copy structure as the result of the unification, then we will finally be left with a pointer to the root of the newly constructed DAG, or a failure-indicator.

A. Incremental Copying Means More Bookkeeping

For this algorithm, we will extend the representation of nodes somewhat as shown in Figure 6. This is essentially the same as shown in Figure 2 except that a `copy` and `status` field have been added. We will use the `copy` field to associate a node with it's copy. We will use the `status` field to indicate whether or not a given node is part of a copy or part of the original graph; it will hold one of two possible values: `"copy"` or `"not-copy"`.

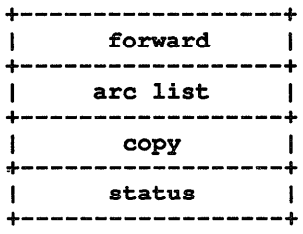


Figure 6: Nondestructive Unification Node

B. Graph Unification With Incremental Copying

The procedure `Unify2` takes as input two nodes, initially the roots of the DAGs to be unified. It recurses on the subgraphs of each DAG until an "atomic" arc value is found. It differs from the algorithm `Unify1` in that it never alters `d1` or `d2`; rather it puts these modifications in the new node being created, named `copy`.

Note that when a copy already exists for one graph or the other, but not both, this algorithm will perform an operation very much like `unify1`, but no forwarding will be done since the changes can all be safely recorded in the copy. This is what is meant by the line marked with an asterisk.

```
PROCEDURE Unify2 (d1 d2)
  Dereference d1, d2.
  IF neither d1 nor d2 have copies THEN
    copy = a new node.
    copy.status = "copy".
    d1.copy, d2.copy = copy.
    newd1  = complementarcs(d1,d2).
    newd2  = complementarcs(d2,d1).
    shared = intersectarcs(d1,d2).
    FOR all arcs in shared DO
      Find the corresponding arc in d2.
      Recursively unify2 the arc values.
      IF unify2 failed THEN
        Return failure.
      ELSE
        Add a new arc in copy.
      ENDIF
    FOR arc in union(newd1,newd2) DO
      Copy the arc-value of each arc,
      honoring existing copies within,
      and place this value in copy.
    Return Copy.
  ELSE if d1 xor d2 has a copy THEN
    Without loss of generality, assume
          d1 has the copy.
*   unify1(d1.copy,d2) preserving d2.
    Return d1.copy.
  ELSE if both d1 and d2 have copies THEN
    Unify1(d1.copy, d2.copy).
  ENDIF
ENDPROCEDURE
```

C. An Example Of Nondestructive Unification

In this section, we will partially walk through the unification of the graphs shown in Figure 4 using the procedure `Unify2`. In the following series of figures, dashed lines indicate the contents of the `copy` field, darkened circles represent "non-copy" nodes, and hollow circles represent nodes which are copies.

Figure 7 shows the state of unification after the path `(a,b)` has been followed during unification. `Unify2` has recursed twice and returned to the top node; three new nodes have been created, one a copy of the root, one a copy of the node on the path `(a)` and the last a copy of the node on the path `(a,b)`. The `copy` field of the appropriate nodes in DAG1 and DAG2 have been filled with the copy nodes, as indicated by the dashed lines.

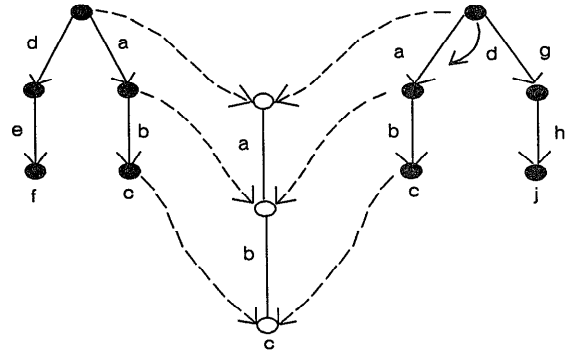


Figure 7: Nondestructive Unification: Snapshot 1

In Figure 8, `Unify2` has followed the path `(d)` on the argument DAGs. But notice that the nodes at the end of path `(a)` and at the end of path `(d)` in DAG2 are the same; a copy of this node was previously made when traversing the path `(a,b)`, *and so this copy is reused* rather than allocating a new node. Subsequently, an arc labelled `e` is added to this reused copy. Finally, `Unify2` recursion unwinds back to the root node of both DAGs.

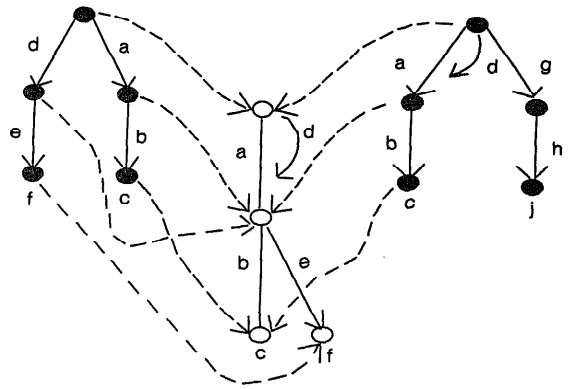


Figure 8: Nondestructive Unification: Snapshot 2

In Figure 9, `Unify2` has added the arc labeled `g` in DAG2 to the result graph, making a copy of the subgraph at the end of that arc and placing it in the result graph. Notice that the subgraph `[h: j]` of DAG2 was copied even though there existed no corresponding subgraph in DAG1. Later we will see that this leads to possible over copying on the part of `Unify2` in some special cases.

The result graph is shown in completed form in Figure 9. Notice that DAG1 and DAG2 have been left unchanged except for their `copy` fields. The new DAG can be returned, with a total of 6 new nodes created, and 6 new arcs created. To unify these DAGs nondestructively with procedure `Unify1`, 10 nodes and 9 arcs would have been created, i.e. a copy of both argument DAGs.

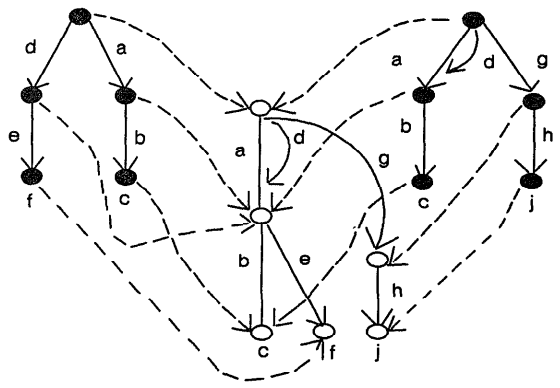


Figure 9: Nondestructive Unification: Final Result

D. Advantages of Incremental Copying

Incrementally copying graphs during unification means over copying is avoided and early copying is eliminated. This incremental copying scheme has the potential for being more efficient than destructive unification (including the preceding copying) both in space and speed. Even if the unification can be guaranteed to succeed, `Unify2` potentially uses less space and time copying than `Unify1`, because it avoids over-copying.

E. Disadvantages of Incremental Copying

`Unify2` is not a perfect algorithm. It can, in some cases, over copy. Such a case is illustrated in the unification of the DAGs in Figure 10. If the top level arcs are unified in the order `x` then `y` then `z`, double copying occurs during the unification of the `z` subgraph.

```
 DAG1              DAG2                 RESULT
[x:[a:b]         [x:1[a:b]            [x:1[a:b
 y:[c:d]          y:2[c:d]      ==>        e:f
 z:[p:1[e:f]      z: [p:<1>                c:d]
    q:<1>]]          q:<2>]]          y:<1>
                                      z:[p:<1>
                                         q:<1>]]
```

Figure 10: Two DAGs That Force Double Copying

To understand this, notice that when the `X` and `Y` subgraphs are unified, a new copy of the graphs `[a: b]` and `[c: d]` was made and associated with the original nodes in DAG1 and DAG2. When unification takes place along the path `(z,p)` a new arc/value of `[e: f]` is combined with the existing copy of `[a: b]` to make the result graph look like `[a:b e:f]`. Finally, the reentrant structure in DAG1 forces the values at the ends of the paths `(z,p)` and `(z,q)` to be unified. But in this case, there is now a copied graph already associated with each of these paths!

The correct result can be obtained by invoking the *destructive* unification routine `Unify1` on both copies, as is done in the final conditional clause of `Unify2`. This provides the correct result DAG, but is unsatisfying with respect to the goals of having a "perfect" unification algorithm, because the algorithm has still over copied, even though it produces the correct result. I have been unable to discover a way to retain the incremental copying scheme but still completely avoid this sort of over copying, although somehow combining "reversible unification" (discussed in the next section) with this algorithm seems to be a promising approach.

VI. Comparing Other Approaches

Several other graph unification algorithms that avoid early copying and over copying have been proposed and implemented. Each of them have emphasized the importance of dealing with copying efficiently. In this section I will compare the nondestructive unification algorithm presented here with these previous techniques.

This comparison of alternate approaches will proceed along the following dimensions:

- Does it eliminate early copying?
- Does it eliminate over copying?
- Does it impose an overhead on DAG operations?
- Is it limited to a certain context?

A. Unification With Structure Sharing

Pereira [Periera 85] has proposed a structure-sharing approach to graph unification, analogous to the structure-sharing techniques used in theorem-proving programs [Boyer and Moore 72], [Warren 83]. In this scheme, a DAG is represented by a *skeleton* and an *environment*. The skeleton is a simple DAG in the same sense used above. However, it must be interpreted along with an environment in which changes to the graph, such as arc bindings or node forwardings, may be added. The unification procedure in such a system looks much like `Unify1`, except that it records changes to the argument DAG nodes in the environment instead of in the nodes themselves. The effect of this technique is to make unification nondestructive and thus non-over and non-early copying. Even in the cases where `Unify2` would over copy, this structure sharing algorithm would not.

Unfortunately, structure sharing has its own set of costs. The mechanism of structure sharing itself places a fixed-cost overhead on *all* node accesses; in Periera's implementation this overhead is log(d), where d is the number of the nodes in the DAG. Any operation manipulating a graph must suffer this log(d) overhead in order to assemble the whole DAG from the skeleton and the updates in the environment. Also, this technique ties each DAG to the derivational environment in which it was created; this appears to have been done as a efficiency measure, in order to share the structure of the environments to the greatest extent.

I found the environment/skeleton scheme hard to implement and extend in a Lisp environment. In fact, it was my discouraging experience when trying accelerate unification via structure-sharing that led to the design of the incremental copying scheme described here. In my implementation, most of the speed-advantages of the structure-sharing were cancelled by the speed cost of the log(d) node access overhead. All the disadvantages of structure-sharing are avoided using incremental copying. Each node in the graph can be accessed in constant time, and the result of a unification is not necessarily tied to the derivational context in which the unification was done. Finally, it is significantly easier to implement and extend than the structure-sharing mechanism.

B. Reversible Unification

Karttunen [Karttunen 86] has implemented a "reversible unification" scheme in which the changes to the argument DAGs are made in a semi-permanent way. After successful unification, a fresh graph is copied from the two altered argument DAGs, and the argument graphs are then restored by undoing all the changes made during unification. If the unification fails, then the argument DAGs are restored and no result graph is made. Reversible unification does not appear to be restricted to any special context.

The most important difference between reversible unification and `Unify2` concerns the restoration process. `Unify2` only changes the original graphs in their `copy` fields. More radical unification changes are made in the copies themselves. Thus, restoring the argument DAGs is only a matter of invalidating the `copy` fields of the argument DAGs. This can be done in constant time by adding a `mark` field which indicates the validity of the `copy` field iff it is equal to some global counter; all the currently valid `copy` fields can be simultaneously invalidated by incrementing the global counter.[3]. This trick is not possible for reversible unification, since it alters its argument DAGs more radically; instead the algorithm must consider each node separately when restoring.

Another difference between reversible unification and `Unify2` is that reversible unification does not incrementally copy it's argument DAGs. This forces it to add a constant-time "save" operation before all modifications and to make a second pass over the result DAGs to create the copy; in `Unify2` this work is traded for a copy-dereferencing operation each time a node is examined.

A possible argument for reversible unification over `Unify2` would be its simplicity, possibly making it easier to implement, validate, and maintain. Reversible unification also avoids the need for adding two fields (`copy`, `status`) to each node through the use of the restoration records. Further, reversible unification will never over copy, even in cases where `Unify2` would.

VII. Conclusions

Graph unification is sometimes implemented as a destructive operation, making it neccesary to copy the argument graphs before beginning the actual unification. Previous research on graph unification showed that this copying is a computation sink, and has sought to correct this.

In this paper I have claimed that the fundamental problem is in designing graph unification as a destructive operation. This forces it to both *over copy* and *early copy*. I have presented a nondestructive graph unification algorithm that minimizes over copying and eliminates early copying. In retrospect, it can be seen that earlier attempts to fix the efficiency problems also addressed the problems of early copying and over copying. The new algorithm presented here is simpler than structure-sharing, and replaces the restoration process of reversible unification with a (small) constant time operation.

There are clearly some tradeoffs to be considered in implementing graph unification. I have tried to outline four that I know of: over copying, early copying, DAG access overhead, and restrictiveness to certain contexts. Complicating this is the surprising complexity possible in the simple structure of a DAG under unificaton; implementing any graph unification algorithm and testing its correctness is a formidable task. One of the problems with the algorithm presented here is that it

[3]Thanks to Mark Tarlton for suggesting this.

has not been proven correct (nor has any other graph unification algorithm, to my knowledge), although we have informally tested it and have been using it on a daily basis for about 5 months.

Future research in this area should strive toward understanding how various design decisions in unification-based parsers affect design decisions for unification. For instance, some parsers may be able to intelligently eliminate rule applications that would fail without invoking unification; one such system is Astro [Wittenburg 86]. If it is known that unification will succeed most of the times it is applied, then one would prefer to optimize the successful case of the unification algorithm. This would mean that early copying might not be a bad design decision.

Another consideration is the purpose to which the unification result will be put. Some DAGs have a short lifespan, such as those on chart edges. Other DAGs produced via unification might have a relatively permanent existence, such as lexical definition graphs. Finally, sometimes one would like to provide detailed information about the causes of unification failure (for debugging grammars, say) while at other times space and time is at a premium, and debugging information is not required. The author's experience suggests that the "perfect graph unification algorithm" may not exist, and is best thought of as a family of related algorithms optimized for different purposes.

VIII. Acknowledgments

This paper has been greatly improved by the thoughtful comments of Elaine Rich and Kent Wittenburg. I am also indebted to Elaine, Kent and the rest of the MCC Lingo group for many interesting discussions on this topic, and to MCC for providing the computational and intellectual environment in which this work took place.

IX. References

[Boyer and Moore 72] R. Boyer and J. Moore. The Sharing of Structure in Theorem-Proving Programs. In *Machine Intelligence 7*. John Wiley and Sons, New York, New York, 1972.

[Karttunen 84] Lauri Karttunen. Features and Values. In *Proceedings of Coling84*, pages 28-33. 1984.

[Karttunen 86] Lauri Karttunen. *D-PATR: A Development Environment For Unification-Based Grammars*. Technical Report CSLI-86-61, Center for the Study of Language and Information, August, 1986.

[Karttunen and Kay 85] L. Karttunen and M. Kay.

Sharing Structure With Binary Trees. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, pages 133-136a. 1985.

[Periera 85] Fernando C. N. Periera. A Structure-Sharing Representation for Unification-Based Grammar Formalisms. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, pages 137-144. 1985.

[Shieber 84] S. Shieber. The Design of a Computer Language for Linguistic Information. In *Proceedings of Coling84*, pages 362-366. 1984.

[Warren 83] David H. D. Warren. *Applied Logic - Its Use And Implementation As A Programming Tool.* Technical Report 290, SRI International, June, 1983.

[Wittenburg 86] Kent B. Wittenburg. *Natural Language Parsing With Combinatory Categorial Grammar In A Graph-Unification-Based Formalism.* PhD thesis, University Of Texas-Austin, August, 1986.