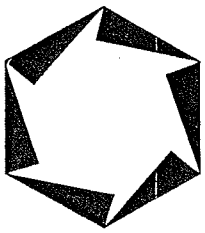
Proceedings of the fifteenth International Conference on Computational Linguistics

Actes du quinzième colloque international en linguistique informatique

COLING-92

Nantes, 23-28/8/1992



Organized on behalf of the

Organisée sous l'égide de l'

ICCL

International Committee on Computational Linguistics

and sponsored by

et sous le parrainage de

Ministère de la recherche et de la technologie,
Ministère des Affaires Étrangères ;
IMAG, CNRS, Universités de Grenoble, Université de Nantes ;
Conseil Général de Loire Atlantique, Ville de Nantes ;
ACL, AFCET, ATALA ;
SITE.

VOLUME II

Quasi-Destructive Graph Unification with Structure-Sharing*

Hideto Tomabechi
Carnegie Mellon University
109 EDSH, Pittsburgh, PA 15213-3890
tomabech@cs.cmu.edu

Abstract

Graph unification remains the most expensive part of unification-based grammar parsing. We focus on one speed-up element in the design of unification algorithms: avoidance of copying of unmodified subgraphs. We propose a method of attaining such a design through a method of structure-sharing which avoids log(d) overheads often associated with structure-sharing of graphs without any use of costly dependency pointers. The proposed scheme eliminates *redundant copying* while maintaining the quasi-destructive scheme's ability to avoid *over copying* and *early copying* combined with its ability to handle cyclic structures without algorithmic additions.

1 Motivation

Despite recent efforts in improving graph unification algorithms, graph unification remains the most expensive part of parsing, both in time and space. ATR's latest data from the SL-TRANS large-scale speech-to-speech translation project ([Morimoto, *et al*, 1990]) show 80 to 90 percent of total parsing time is still consumed by graph unification where 75 to 95 percent of time is consumed by graph copying functions.[1] Quasi-Destructive (Q-D) Graph Unification ([Tomabechi, 1991]) was developed as a fast variation of non-destructive graph unification based upon the notion of time-sensitive 'quasi-destruction' of node structures. The Q-D algorithm was proposed based upon the following accepted observation about graph unification:

Unification does not always succeed.
Copying is an expensive operation.

The design of the Q-D scheme was motivated by the following two principles for fast graph unification based upon the above observations:

- **Copying should be performed only for successful unifications.**
- **Unification failures should be found as soon as possible.**

and eliminated Over Copying and Early Copying (as defined in [Tomabechi, 1991][2]) and ran about twice the speed of [Wroblewski, 1987]'s algorithm.[3] In this paper we propose another design principle for graph unification based upon yet another accepted observation that:

Unmodified subgraphs can be shared.

At least two schemes have been proposed recently based upon this observation (namely [Kogure, 1990] and [Emele, 1991]); however, both schemes are based upon the incremental copying scheme and as described in [Tomabechi, 1991] incremental copying schemes inherently suffer from *Early Copying* as defined in that article. This is because, when a unification fails, the copies that were created up to the point of failure are wasted if copies are created incrementally. By way of definition we would like to categorize the sharing of structures in graphs into Feature-Structure Sharing (FS-Sharing) and Data-Structure Sharing (DS-Sharing). Below are our definitions:

- **Feature-Structure Sharing:** Two or more distinct paths within a graph share the same subgraph by converging on the same node - equivalent to the notion of *structure sharing* or *reentrancy* in linguistic theories (such as in [Pollard and Sag, 1987]).
- **Data-Structure Sharing:** Two or more distinct graphs share the same subgraph by converging on the same node – the notion of

*This research was done while the author was a Visiting Research Scientist at ATR Interpreting Telephony Research Laboratories.

[1] Based on unpublished reports from Knowledge and Data Processing Dept. ATR. The observed tendency was that sentences with very long parsing time requiring a large number of unification calls (over 2000 top-level calls) consumed extremely large proportion (over 93 percent) of total parsing time for graph unification. Similar data reported in [Kogure, 1990].

[2] Namely,

- **Over Copying:** Two dags are created in order to create one new dag. – This typically happens when copies of two input dags are created prior to a destructive unification operation to build one new dag.
- **Early Copying:** Copies are created prior to the failure of unification so that copies created since the beginning of the unification up to the point of failure are wasted.

Wroblewski defined Early Copying as follows: "The argument dags are copied *before* unification started. If the unification fails then some of the copying is wasted effort" and restricts early copying to cases that only apply to copies that are created prior to a unification. Our definition of Early Copying includes copies that are created during a unification and created up to the point of failure which were uncovered by Wroblewski's definition.

[3] Recent experiments conducted in the Knowledge and Data Processing Dept. of ATR shows the original Q-D algorithm consistently runs at about 40 percent of the elapsed time of Wroblewski's algorithm with its SL-TRANS large-scale spoken-language translation system (with over 10,000 grammatical graph nodes).

structure-sharing at the data structure level. [Kogure, 1990] calls copying of such structures *Redundant Copying*.

Virtually all graph-unification algorithms support FS-Sharing and some support DS-Sharing with varying levels of overhead. In this paper we propose a scheme of graph unification based upon a quasi-destructive graph unification method that attains DS-Sharing with virtually no overhead for structure-sharing. Henceforth, in this paper, structure-sharing refers to DS-sharing unless otherwise noted. We will see that the introduction of structure-sharing to quasi-destructive unification attains another two-fold increase in run-time speed. The graphs handled in the scheme can be any directed graph and cyclicity is handled without any algorithmic additions.

Our design principles for achieving structure-sharing in the quasi-destructive scheme are:

- **Atomic and Bottom nodes can be shared**[4] – Atomic nodes can be shared safely since they never change their values. Bottom nodes can be shared[5] since bottom nodes are always forwarded to some other nodes when they unify.
- **Complex nodes can be shared unless they are modified** – complex nodes can be considered modified if they are a target of the forwarding operation or if they received the current addition of complement arcs (into comp-arc-list in quasi-destructive scheme).

By designing an algorithm based upon these principles for structure-sharing while retaining the quasi-destructive nature of [Tomabechi, 1991]'s algorithm, our scheme eliminates Redundant Copying while eliminating both Early Copying and Over Copying.

2 Q-D Graph Unification

We would first like to describe the quasi-destructive (Q-D) graph unification scheme which is the basis of our scheme. As a data structure, a node is represented with five fields: type, arc-list, comp-arc-list, forward, copy, and generation.[6] The data-structure for an arc has two fields, 'label' and 'value'. 'Label' is an atomic symbol which labels the arc, and 'value' is a pointer to a node structure.

The central notion of the Q-D algorithm is the dependency of the representational content on the global timing clock (or the global counter for the current generation of unifications). Any modification made to comp-arc-list, forward, or copy fields during one top-level unification can be invalidated by one increment operation on the global timing counter. Contents of the comp-arc-list, forward and copy fields are

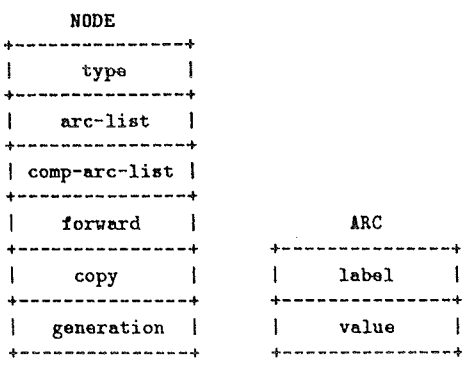


Figure 1: **Node and Arc Structures**

respected only when the generation mark of the particular node matches the current global counter value. Q-D graph unification has two kinds of arc lists: 1) arc-list and 2) comp-arc-list. Arc-list contains the arcs that are permanent (i.e.: ordinary graph arcs) and comp-arc-list contains arcs that are only valid during one top-level graph unification operation. The algorithm also uses two kinds of forwarding links, i.e., permanent and temporary. A permanent forwarding link is the usual forwarding link found in other algorithms ([Pereira, 1985], [Wroblewski, 1987], etc). Temporary forwarding links are links that are only valid during one top-level unification. The currency of the temporary links is determined by matching the content of the generation field for the links with the global counter; if they match, the content of this field is respected[7]. As in [Pereira, 1985], the Q-D algorithm has three types of nodes: 1) :atomic, 2) :bottom[8], and 3) :complex. :atomic type nodes represent atomic symbol values (such as 'Noun'), :bottom type nodes are variables and :complex type nodes are nodes that have arcs coming out of them. Arcs are stored in the arc-list field. The atomic value is also stored in the arc-list if the node type is :atomic. :bottom nodes succeed in unifying with any nodes and the result of unification takes the type and the value of the node that the :bottom node was unified with. :atomic nodes succeed in unifying with :bottom nodes or :atomic nodes with the same value (stored in the arc-list). Unification of an :atomic node with a :complex node immediately fails. :complex nodes succeed in unifying with :bottom nodes or with :complex nodes whose subgraphs all unify.[9] Figure 2 is the central quasi-destructive graph unification algorithm and Figure 3 is the dereferencing[10] function. Figure 4 shows the algorithm for copying nodes and arcs (called from unify0) while respecting the contents of comp-arc-lists.

[4] Atomic nodes are nodes that represent atomic values, Bottom nodes are nodes that represent variables.

[5] As long as the unification operation is the only operation to modify graphs.

[6] Note that [Tomabechi, 1991] used separate mark fields for comp-arc-list, forward, and copy; currently however, only one generation mark is used for all three fields. Thanks are due to Hidehiko Matsuo of Toyo Information Systems (TIS) for suggesting this.

[7] We do not have a separate field for temporary forwarding links; instead, we designate the integer value 9 to represent a permanent forwarding link. We start incrementing the global counter from 10 so whenever the generation mark is not 9, the integer value must equal the global counter value to respect the forwarding link.

[8] Bottom is called leaf in Pereira's algorithm.

[9] Arc values are always nodes and never symbolic values because :atomic and :bottom nodes may be (or become) pointed to by multiple arcs (i.e, FS-Sharing) depending on grammar constraints, and we do not want arcs to contain terminal atomic values.

[10] Dereferencing is an operation to recursively traverse forwarding links to return the target node of forwarding.

QUASI-DESTRUCTIVE GRAPH UNIFICATION

```
FUNCTION unify-dg(dg1,dg2);
  result ← catch with tag 'unify-fail
           calling unify0(dg1,dg2);
  increment *unify-global-counter*; ;; starts from 10 [11]
  return(result);
END;

FUNCTION unify0(dg1,dg2);
  if '*T* = unify1(dg1,dg2); THEN
    copy ← copy-dg-with-comp-arcs(dg1);
    return(copy);
END;

FUNCTION unify1 (dg1-underef,dg2-underef);
  dg1 ← dereference-dg(dg1-underef);
  dg2 ← dereference-dg(dg2-underef);
  IF (dg1.copy is non-empty) THEN
     dg1.copy ← nil; ;; cutoff uncurrent copy
  IF (dg2.copy is non-empty) THEN
     dg2.copy ← nil;
  IF (dg1 = dg2)[12]THEN
     return('*T*);
  ELSE IF (dg1.type = :bottom) THEN
     forward-dg(dg1,dg2,:temporary);
     return('*T*);
  ELSE IF (dg2.type = :bottom) THEN
     forward-dg(dg2,dg1,:temporary);
     return('*T*);
  ELSE IF (dg1.type = :atomic AND
            dg2.type = :atomic) THEN
            IF (dg1.arc-list = dg2.arc-list)[13]THEN
               forward-dg(dg2,dg1,:temporary);
               return('*T*);
            ELSE throw[14]with keyword 'unify-fail;
  ELSE IF (dg1.type = :atomic OR
            dg2.type = :atomic) THEN
            throw with keyword 'unify-fail;
  ELSE shared ← intersectarcs(dg1,dg2);
       forward-dg(dg2,dg1,:temporary);[15]
       FOR EACH arc IN shared DO
             unify1(destination of
                          the shared arc for dg1,
                    destination of
                          the shared arc for dg2);
       new ← complementarcs(dg2,dg1);[16]
       IF[17](dg1.comp-arc-list is non-empty) THEN
            IF (dg1.generation = *unify-global-counter*) THEN
               FOR EACH arc IN new DO
                     push arc to dg1.comp-arc-list;
            ELSE dg1.comp-arc-list ← nil;
       ELSE dg1.generation ← *unify-global-counter*;
            dg1.comp-arc-list ← new;
       return ('*T*);
END;
```

Figure 2: **The Q-D Unification Functions**

The functions Complementarcs(dg1,dg2) and Intersectarcs(dg1,dg2) return the set-difference (the arcs with labels that exist in dg1 but not in dg2) and intersection (the arcs with labels that exist both in dg1 and dg2). During the set-difference and set-intersection operations, the content of comp-arc-lists are respected as parts of arc lists if the generation mark matchs the current value of the global timing counter. Forward(dg1, dg2, :forward-type) puts dg2 in the forward field of dg1. If the keyword in the function call is :temporary, the current value of the *unify-global-counter* is written in the generation field of dg1. If the keyword is :permanent, 9 is written in the generation field of dg1.[18] The temporary forwarding links are necessary to handle reentrancy and cycles. As soon as unification (at any level of recursion through shared arcs) is performed, a temporary forwarding link is made from dg2 to dg1 (dg1 to dg2 if dg1 is of type :bottom). Thus, during unification, a node already unified by other recursive calls to unify1 within the same unify0 call has a temporary forwarding link from dg2 to dg1 (or dg1 to dg2). As a result, if this node becomes an input argument node, dereferencing the node causes dg1 and dg2 to become the same node and unification immediately succeeds. Thus, a subgraph below an already unified node will not be checked more than once even if an argument graph has a cycle.[19]

GRAPH NODE DEREFERENCING

```
FUNCTION dereference-dg(dg);
  forward-dest ← dg.forward;
  IF (forward-dest is non-empty) THEN
     IF (dg.generation = *unify-global-counter* OR
         dg.generation = 9) THEN
         dereference-dg(forward-dest);
     ELSE dg.forward ← nil; ;; make it GCable
          return(dg);
  ELSE return(dg);
END;
```

Figure 3: **The Q-D Dereference Function**

QUASI-DESTRUCTIVE COPYING

```
FUNCTION copy-dg-with-comp-arcs(dg-underef);
  dg ← dereference-dg(dg-underef);
  IF (dg.copy is non-empty AND
      dg.copy.generation[20]= *unify-global-counter*) THEN
      return(dg.copy);[21]
  ELSE IF (dg.type = :atomic) THEN
      newcopy ← create-node();[22]
      newcopy.type ← :atomic;
      newcopy.arc-list ← dg.arc-list;
      newcopy.generation ← *unify-global-counter*;
      dg.copy ← newcopy;
      return(newcopy);
  ELSE IF (dg.type = :bottom) THEN
      newcopy ← create-node();
      newcopy.type ← :bottom;
      newcopy.generation ← *unify-global-counter*;
      dg.copy ← newcopy;
      return(newcopy);
  ELSE
```

[11] 9 indicates a permanent forwarding link.

[12] Equal in the 'eq' sense. Because of forwarding and cycles, it is possible that dg1 and dg2 are 'eq'.

[13] Arc-list contains atomic value if the node is of type :atomic.

[14] Catch/throw construct; i.e., immediately return to *unify-dg*.

[15] This was performed after FOR EACH loop in [Tomabechi, 1991] which could have caused a problem with a successful cyclic call. Thanks are due to Marie Boyle of University of Tuebingen for suggesting the change.

[16] Complementarcs(dg2,dg1) was called before unify1 recursions in [Tomabechi, 1991], Currently it is moved to after all unify1 recursions successfully return. Thanks are also due to Marie Boyle for suggesting this.

[17] This check was added after [Tomabechi, 1991] to avoid over-writing the comp-arc-list when it is written more than once within one unify0 call. Thanks are due to Peter Neuhaus of Universität Karlsruhe for reporting this problem.

[18] Permanent forwardings may be needed by grammar compilers that merge graphs.

[19] Also, during copying subsequent to a successful unification, two arcs converging into the same node will not cause overcopying simply because if a node already has a copy then the copy is returned.

```
    newcopy ← create-node();
    newcopy.type ← :complex;
    newcopy.generation ← *unify-global-counter*;
    dg.copy ← newcopy;[23]
    FOR ALL arc IN dg.arc-list DO
        newarc ← copy-arc-and-comp-arc(arc);
        push newarc into newcopy.arc-list;
    IF (dg.comp-arc-list is non-empty AND
        dg.generation = *unify-global-counter*) THEN
            FOR ALL comp-arc IN dg.comp-arc-list DO
                newarc ← copy-arc-and-comp-arc(comp-arc);
                push newarc into newcopy.arc-list;
    dg.comp-arc-list ← nil;
    return (newcopy);
END;

FUNCTION copy-arc-and-comp-arc(input-arc);
  label ← input-arc.label;
  value ← copy-dg-with-comp-arcs(input-arc.value);
  return a new arc with label and value;
END;
```

Figure 4: **Node and Arc Copying Functions**

3 Q-D Copying + DS-Sharing

In order to attain structure-sharing during Quasi-Destructive graph unification, no modification is necessary for the unification functions described in the previous section. This section describes the quasi-destructive copying with structure-sharing which replaces the original copying algorithm. Since unification functions are unmodified, the Q-D unification without structure-sharing can be mixed trivially with the Q-D unification with structure-sharing if such a mixture is desired (by simply choosing different copying functions). Informally, the Q-D copying with structure-sharing is performed in the following way. Atomic and bottom nodes are shared. A complex node is shared if no nodes below that node are changed (a node is considered changed by being a target of forwarding or having a valid comp-arc-list). If a node is changed then that information is passed up the graph path using multiple-value binding facility when a copy of the nodes are recursively returned. Two values are returned, the first value being the copy (or original) node and the second value being the flag representing whether any of the node below that node (including that node) has been changed. Atomic and bottom nodes are always shared; however, they are considered changed if they were a target of forwarding so that the 'changed' information is passed up. If the complex node is a target of forwarding, if no node below that node is changed then the original complex node is shared; however, the 'changed' information is passed up when the recursion returns. Below is the actual algorithm description for the Q-D copying with structure-sharing.

Q-D Copying with Structure-Sharing

```
FUNCTION copy-dg-with-comp-arcs-share(dg-underef);
  dg ← dereference-dg(dg-underef);
  IF (dg.copy is non-empty AND
      dg.copy.generation = *unify-global-counter*) THEN
      IF (dg = dg.copy) THEN [24]
      newcopy ← create-node();
      newcopy.type ← :bottom;
      newcopy.generation← *unify-global-counter*;
      dg.copy ← newcopy;
      values(dg.copy,:changed); [25]
      ELSE values(dg.copy,:changed);
  ELSE IF (dg = dg-underef) THEN
      copy-node-comp-not-forwarded(dg);
  ELSE copy-node-comp-forwarded(dg);
END;

FUNCTION copy-node-comp-not-forwarded(dg);
  IF (dg.type = :atomic) THEN values(dg,nil);
  ;; return original dg with 'no change' flag.
  ELSE IF (dg.type = :bottom) THEN values(dg,nil);
  ELSE
      IF (dg.comp-arc-list is non-empty AND
          dg.generation = *unify-global-counter*) THEN
          newcopy ← create-node();
          newcopy.type ← :complex;
          newcopy.generation ← *unify-global-counter*;
          dg.copy ← newcopy;
          FOR ALL arc IN dg.arc-list DO
              newarc
                ← first value of copy-arc-and-comp-arc-share(arc);
              push newarc into newcopy.arc-list;
          FOR ALL comp-arc IN dg.comp-arc-list DO
              newarc
                ← first value of
                    copy-arc-and-comp-arc-share(comp-arc);
              push newarc into newcopy.arc-list;
          dg.comp-arc-list ← nil;
          values(newcopy,:changed);
      ELSE
          state ← nil, arcs ← nil;
          dg.copy ← dg[26], dg.generation ← *unify-global-counter*;
          FOR ALL arc IN dg.arc-list DO
              newarc,changed ← copy-arc-and-comp-arc-share(arc);
              push newarc into arcs;
              IF (changed has value) THEN
                  state ← changed;
          IF (state has value) THEN
              IF (dg.copy ≠ dg) THEN
                  dg.copy.arc-list ← arcs;
                  dg.copy.type ← :complex;
                  values(dg.copy,:changed);
              ELSE
                  newcopy ← create-node();
                  newcopy.type ← :complex;
                  newcopy.generation ← *unify-global-counter*;
                  newcopy.arc-list ← arcs;
                  dg.copy ← newcopy;
                  values(newcopy,:changed);
          ELSE dg.copy ← nil; ;;reset copy field
               values(dg,nil);
END;

FUNCTION copy-node-comp-forwarded(dg);
  IF (dg.type = :atomic) THEN values(dg,:changed);
  ;; return original dg with 'changed' flag.
  ELSE IF (dg.type = :bottom) THEN values(dg,:changed);
  ELSE
      IF (dg.comp-arc-list is non-empty AND
          dg.generation = *unify-global-counter*) THEN
          newcopy ← create-node();
          newcopy.type ← :complex;
```

[20] I.e., the 'generation' field of the node stored in the 'copy' field of the 'dg' node. The algorithm described in [Tomabechi, 1991] used 'copy-mark' field of 'dg'. Currently 'generation' field replaces the three mark field described in the article.

[21] I.e., the existing copy of the node.

[22] Creates an empty node structure.

[23] This operation to set a newly created copy node into the 'copy' field of 'dg' was done after recursion into subgraphs in the algorithm description in [Tomabechi, 1991] which was a cause of infinite recursion with a particular type of cycles in the graph. By moving up to this position from after the recursion, such a problem can be effectively avoided. Thanks are due to Peter Neuhaus for reporting the problem.

```
        newcopy.generation ← *unify-global-counter*;
        dg.copy ← newcopy;
        FOR ALL arc IN dg.arc-list DO
            newarc
              ← first value of copy-arc-and-comp-arc-share(arc);
            push newarc into newcopy.arc-list;
        FOR ALL comp-arc IN dg.comp-arc-list DO
            newarc
              ← first value of
                copy-arc-and-comp-arc-share(comp-arc);
            push newarc into newcopy.arc-list;
        dg.comp-arc-list ← nil;
        values(newcopy,:changed);
    ELSE
        state ← nil, arcs ← nil;
        dg.copy ← dg, dg.generation ← *unify-global-counter*;
        FOR ALL arc IN dg.arc-list DO
            newarc,changed ← copy-arc-and-comp-arc-share(arc);
            push newarc into arcs;
            IF (changed has value) THEN
                state ← changed;
        IF (state has value) THEN
            IF (dg.copy ≠ dg) THEN
                dg.copy.arc-list ← arcs;
                dg.copy.type ← :complex;
                values(dg.copy.:changed);
            ELSE
                newcopy ← create-node();
                newcopy.type ← :complex;
                newcopy.generation ← *unify-global-counter*;
                newcopy.arc-list ← arcs;
                dg.copy ← newcopy;
                values(newcopy.:changed);
        ELSE dg.copy ← nil;
             values(dg,changed); ;; considered changed
END;

FUNCTION copy-arc-and-comp-arc-share(input-arc);
  destination,changed
        ← copy-dg-with-comp-arcs-share(input-arc.value);
  IF (changed has value) THEN
      label ← input-arc.label;
      value ← destination;
      values(a new arc with label and value,:changed);
  ELSE values(input-arc,nil); ;; return original arc
END;
```

Figure 5: **Structure-Sharing Copying Functions**

4 Experiments

Table 1 shows the results of our experiments using an HPSG-based sample Japanese grammar developed at ATR for a conference registration telephone dialogue domain. 'Unifs' represents the total number of top-level unifications during a parse (i.e, the number of calls to the top-level 'unify-dg', and not 'unify1')[28]. 'USrate' represents the ratio of successful unifications to the total number of unifications. We parsed each sentence three times on a Symbolics 3620 using three unification methods, namely, Wroblewski's algorithm, a quasi-destructive method without structure-sharing, and a quasi-destructive method with structure-sharing. We took the shortest elapsed time for each method ('W' represents Wroblewski's algorithm with a modification to handle cycles and variables[29], 'QD' represents the quasi-destructive method without structure-sharing, and 'QS' represents the proposed method with structure-sharing). Data structures are the same for all three unification methods except for additional fields for comp-arc-list in the Q-D methods. Same functions are used to interface with Earley's parser and the same subfunctions are used wherever possible (such as creation and access of arcs) to minimize the differences that are not purely algorithmic. 'Number of Copies' represents the number of nodes created during each parse. 'Number of Arcs' represents the number of arcs created during each parse.

We used Earley's parsing algorithm for the experiment. The Japanese grammar is based on HPSG analysis ([Pollard and Sag, 1987]) covering phenomena such as coordination, case adjunction, adjuncts, control, slash categories, zero-pronouns, interrogatives, WH constructs, and some pragmatics (speaker, hearer relations, politeness, etc.) ([Yoshimoto and Kogure, 1989]). The grammar covers many of the important linguistic phenomena in conversational Japanese. The grammar graphs which are converted from the path equations contain 2324 nodes.[30] We used 16 sentences from a sample telephone conversation dialog which range from very short sentences (one word, i.e., *iie* 'no') to relatively long ones (such as *soredehakochirakarasochiranitourokuyoushiwoookuriitashimasu* 'In that case, we [speaker] will send you [hearer] the registration form.'). Thus, the number of (top-level) unifications per sentence varied widely (from 6 to over 500).

5 Discussion:

Pereira ([Pereira, 1985]) attains structure-sharing by having the result graph share information with the original graphs by storing changes to the 'environment'. There will be the log(d) overhead (where d is the number of nodes in a graph) associated with Pereira's method that is required during node access to assemble the whole graph from the 'skeleton' and the updates in the 'environment'. In the proposed scheme, since the arcs directly point to the original graph structures there will be no overhead for node accesses. Also, during unification, since changes are

[24] Currently, all nodes are copied in a cycle in order to prevent the split of the copy and the original when node above an unchanged original is modified. Thanks are due to Makoto Takahasi of TIS for suggesting the fix. Of course, a better method, if possible, would be to copy the whole cycle only when at least one node in the cycle is modified.

[25] 'Values' return multiple values from a function. In our algorithm, two values are returned. The first value is the result of copying, and the second value is a flag indicating if there was any modification to the node or to any of its descendants.

[26] Temporarily set copy of the dg to be itself.

[27] Multiple-value-bind call. The first value is bound to 'newarc', and the second value is bound to 'changed'.

[28] Unify1 is called several times the number of unify-dg in the grammar used in the experiment. For example unify1 was called 3299 times for sentence 9 when unify-dg was called 480 times.

[29] Kogure ([Kogure, 1989]) describes a trivial time modification to Wroblewski's algorithm to handle cycles which is used in our experiments.

[30] Disjunctive equations are preprocessed by the grammar reader module to expand into cross-multiples, whereas in ATR's SL-TRANS system, Kasper's method ([Kasper, 1987]) to handle disjunctive feature-structures is adopted.

sent#	Unifs	USrate	Elapsed time(sec)			Num of Copies			Num of Arcs		
			W	QD	QS	W	QD	QS	W	QD	QS
1	6	0.50	0.20	0.15	0.13	107	79	18	113	123	36
2	101	0.34	2.53	1.16	1.10	2285	1317	407	2441	1917	760
3	18	0.22	0.40	0.20	0.20	220	111	26	182	183	62
4	71	0.55	2.20	1.24	0.91	2151	1564	514	2408	2191	879
5	305	0.37	13.78	6.51	3.65	9092	5224	1220	9373	7142	2272
6	59	0.27	3.20	0.64	0.50	997	549	97	874	797	204
7	6	0.50	0.21	0.13	0.11	107	79	18	113	123	36
8	81	0.51	3.17	1.59	1.21	2406	1699	401	2572	2334	710
9	480	0.37	24.62	8.11	5.74	15756	8986	1696	17358	12427	3394
10	555	0.41	40.15	16.39	8.80	18822	11234	2737	20323	15375	5116
11	109	0.45	4.60	1.71	1.41	2913	1938	555	3089	2712	992
12	428	0.33	19.57	8.24	4.45	13363	7491	1586	14321	10218	3059
13	559	0.39	37.76	11.74	6.23	17741	9417	2483	19014	13055	4471
14	52	0.38	3.81	0.90	0.50	947	693	107	893	983	199
15	77	0.55	2.50	1.57	0.93	2137	1513	428	2436	2185	793
16	77	0.55	2.53	1.57	0.90	2137	1513	428	2436	2185	793
total	2984		161.23	61.85	36.77	91181	53407	12721	97946	73950	23776
(% for total)			100%	38.4%	22.8%	100%	58.6%	14%	100%	76%	24%

Table 1: **Comparison of three methods**

stored directly in the nodes (in the quasi-destructive manner) there will be no overhead for reflecting the changes to graphs during unification. We share the principle of storing changes in a restorable way with [Karttunen, 1986]'s reversible unification and copy graphs only after a successful unification. However, Karttunen's method does not use structure-sharing. Also, In Karttunen's method[31], whenever a destructive change is about to be made, the attribute value pairs[32] stored in the body of the node are saved into an array. The dag node structure itself is also saved in another array. These values are restored after the top level unification is completed. (A copy is made prior to the restoration operation if the unification was a successful one.) Thus, in Karttunen's method, each node in the entire argument graph that has been destructively modified must be restored separately by retrieving the attribute-values saved in an array and resetting the values into the dag structure skeletons saved in another array. In the Q-D method, one increment to the global counter can invalidate all the changes made to the nodes. [Karttunen and Kay, 1985] suggests the use of lazy evaluation to delay destructive changes during unification. [Godden, 1990] presents one method to delay copying until a destructive change is about to take place. Godden uses delayed closures to directly implement lazy evaluation during unification. While it may be conceptually straightforward to take advantage of delayed evaluation functionalities in programming languages, actual efficiency gain from such a scheme may not be significant. This is because such a scheme simply shifts the time and space consumed for copying to creating and evaluating closures (which could be very costly compared to 'defstruct' operations to create copies which are often effectively optimized in many commercial compilers). [Kogure, 1990] and [Emele, 1991] also use the lazy evaluation idea to delay destructive changes. Both Kogure and Emele avoid direct usage of delayed evaluation by using pointer operations. As Emele suggests, Kogure's method also requires a special dependency information to be maintained which adds an overhead along with the cost for traversing the dependency arcs. Also, a second traversal of the set of dependent nodes is required for actually performing the copying. Emele proposes a method of dereferencing by adding environment information that carries a sequence of generation counters so that a specific generation node can be found by traversing the forwarding links until a node with that generation is found. While this allows undoing destructive changes cheaply by backtracking the environment, every time a specific graph is to be accessed the whole graph needs to be reconstructed by following the forwarding pointers sequentially as specified in the environment list (except for the root node) to find the node that shares the same generation number as the root node. Therefore, similar to Pereira's method, there will be Nlog(d) overhead associated with constructing each graph every time a graph is accessed, where d is the number of nodes in the graph and N is the average depth of the environmental deference chain. This would cause a problem if the algorithm is adopted for a large-scale system in which result graphs are unified against other graphs many times. Like Wroblewski's method, all three lazy methods (i.e, Godden's, Kogure's and Emele's) suffer from the problem of *Early Copying* as defined in [Tomabechi, 1991]. This is because the copies that are incrementally created up to the point of failure during the same top-level unification are wasted. The problem is inherent in incremental copying scheme and this problem is eliminated completely in [Karttunen,

[31] The discussion of Karttunen's method is based on the D-PATR implementation on Xerox 1100 machines ([Karttunen, 1986]).

[32] I.e., arc structures: 'label' and 'value' pairs in our vocabulary.

1986] and in the Q-D method.[33]

There is one potential problem with the structure-sharing idea which is shared by each of the schemes including the proposed method. This happens when operations other than unification modify the graphs. (This is typical when a parser cuts off a part of a graph for subsequent analysis[34].) When such operations are performed, structure-sharing of bottom (variable) nodes may cause problems when a subgraph containing a bottom is shared by two different graphs and these graphs are used as arguments of a unification function (either as the part of the same input graph or as elements of dg1 and dg2). When a graph that shares a bottom node is not used in its entirety, then the represented constraint postulated by the path leading to the bottom node is no longer the same. Therefore, when such a graph appears in the same unification along with some other graph with which it DS shares the same bottom node, there will be a false FS-Sharing. (If the graph is used in its entirety this is not a problem since the two graph paths would unify anyway.) This problem happens only when neither of the two graphs that DS-Shares the same bottom node was unified against some other graph before appearing in the same unification.[35] (If either was once unified, forwarding would have avoided this problem). The methods to avoid such a problem can be 1) As long as these convergence of bottom nodes are used for features that are not passed up during parsing, the problems does not affect the result of parse in any way – which seems the case with the grammars at ATR and CMU. 2) A parser can be modified so that when it modifies a graph other than through graph unification[36], it creates copies of the arc structures containing the bottom nodes. In the proposed method this can be done by calling the copy function without structure-sharing before a parser modifies a graph. 3) A parser can be modified so that it does not cut off parts of graphs and use the graphs in their entirety (this should not add complexity once structure-sharing is introduced to unification). Thus, although the space and time reduction attained by structure-sharing can be significant, DS-Sharing can cause problems unless it is used with a caution (by making sure variable sharing does not cause erroneous sharing by using these or some other methods).

[33] Lazy methods delay copying until a destructive change is to be performed so that unnecessary copies are not created within a particular recursion into a unification function; however, since each shared arc recursion is independent (nondeterministic), even if there are no unnecessary copies created at all in one particular recursion, if there is a failure in some other shared arc recursion (at some depth), then the copies that are created by successful shared arc recursions up to the point of detection of failure will become wasted. As long as the basic control structure remains incremental, this is inherent in the incremental method. In other words, the problem is inherent in these incremental methods by definition.

[34] For example, many parsers cut off a subgraph of the path 0 for applying further rules when a rule is accepted.

[35] Such cases may happen when the same rule (such as V ⇒ V) augmented with a heavy use of convergence in the bottom nodes is applied many times during a parse.

[36] Such as when a rule is accepted and subgraph of 0 path is cut off.

6 Conclusion

The structure-sharing scheme introduced in this paper made the Q-D algorithm run significantly faster. The original gain of the Q-D algorithm was due to the fact that it does not create any Over Copies or Early Copies whereas incremental copying scheme inherently produces Early Copies (as defined in [Tomabechi, 1991]) when a unification fails. The proposed scheme makes the Q-D algorithm fully avoid Redundant Copies as well by only copying the lowest nodes that need to be copied due to destructive changes caused by successful unifications only. Since there will be virtually no overhead associated with structure-sharing (except for returning two values instead of one to pass up :changed information when recursion for copying returns), the performance of the proposed structure-sharing scheme should not drop even when the grammar size is significantly scaled up. With the demonstrated speed of the algorithm, as well as the ability to handle cyclicity in the graphs, and ease of switching between structure-sharing and non-structure sharing, the algorithm could be a viable alternative to existing unification algorithms used in current natural language systems.

References

[Emele, 1991] Emele, M. "Unification with Lazy Non-Redundant Copying". In *Proc. of ACL-91*, 1991.

[Godden, 1990] Godden, K. "Lazy Unification" In *Proc. of ACL-90*, 1990.

[Karttunen, 1986] Karttunen, L. "D-PATR: A Development Environment for Unification-Based Grammars". In *Proc. of COLING-86*, 1986. (Also, Report CSLI-86-61 Stanford University).

[Karttunen and Kay, 1985] Karttunen, L. and M. Kay. "Structure Sharing with Binary Trees". In *Proc. of ACL-85*, 1985.

[Kasper, 1987] Kasper, R. "A Unification Method for Disjunctive Feature Descriptions". In *Proc. of ACL-87*, 1987.

[Kogure, 1989] Kogure, K. *A Study on Feature Structures and Unification*. ATR Technical Report. TR-1-0032, 1988.

[Kogure, 1990] Kogure, K. "Strategic Lazy Incremental Copy Graph Unification". In *Proc. of COLING-90*, 1990.

[Morimoto, *et al*, 1990] Morimoto, T., H. Ilda, A. Kurematsu, K. Shikano, and T. Aizawa. "Spoken Language Translation Toward Realizing an Automatic Telephone Interpretation System". In *Proc. of InfoJapan 1990*, 1990.

[Pereira, 1985] Pereira, F. "A Structure-Sharing Representation for Unification-Based Grammar Formalisms". In *Proc of ACL-85*, 1985.

[Pollard and Sag, 1987] Pollard, C. and I. Sag. *Information-based Syntax and Semantics*. Vol 1, CSLI, 1987.

[Yoshimoto and Kogure, 1989] Yoshimoto, K. and K. Kogure *Japanese Sentence Analysis by means of Phrase Structur Grammar*. ATR Technical Report. TR-1-0049, 1989.

[Tomabechi, 1991] Tomabechi, H. "Quasi-Destructive Graph Unification". In *Proc. of ACL-91*, 1991.

[Wroblewski, 1987] Wroblewski, D."Nondestructive Graph Unification". In *Proc. of AAAI87*, 1987.

A LINEAR LEAST SQUARES FIT MAPPING METHOD FOR INFORMATION RETRIEVAL FROM NATURAL LANGUAGE TEXTS

YIMING YANG
CHRISTOPHER G. CHUTE

Section of Medical Information Resources
Mayo Clinic/Foundation
Rochester, Minnesota 55905 USA

ABSTRACT

This paper describes a unique method for mapping natural language texts to canonical terms that identify the contents of the texts. This method learns empirical associations between free-form texts and canonical terms from human-assigned matches and determines a Linear Least Squares Fit (LLSF) mapping function which represents weighted connections between words in the texts and the canonical terms. The mapping function enables us to project an arbitrary text to the canonical term space where the "transformed" text is compared with the terms, and similarity scores are obtained which quantify the relevance between the the text and the terms. This approach has superior power to discover synonyms or related terms and to preserve the context sensitivity of the mapping. We achieved a rate of 84% in both the recall and the precision with a testing set of 6,913 texts, outperforming other techniques including string matching (15%), morphological parsing (17%) and statistical weighting (21%).

1. Introduction

A common need in natural language information retrieval is to identify the information in free-form texts using a selected set of canonical terms, so that the texts can be retrieved by conventional database techniques using these terms as keywords. In medical classification, for example, original diagnoses written by physicians in patient records need to be classified into canonical disease categories which are specified for the purposes of research, quality improvement, or billing. We will use medical examples for discussion although our method is not limited to medical applications.

String matching is a straightforward solution to automatic mapping from texts to canonical terms. Here we use "term" to mean a canonical description of a concept, which is often a noun phrase. Given a text (a "query") and a set of canonical terms, string matching counts the common words or phrases in the text and the terms, and chooses the term containing the largest overlap as most relevant. Although it is a simple and therefore widely used technique, a poor success rate (typically 15% - 20%) is observed [1]. String-matching-based methods suffer from the problems known as "too little" and "too many". As an example of the former, *high blood pressure* and *hypertension* are synonyms but a straightforward string matching cannot capture the equivalence in meaning because there is no common word in these two expressions. On the other hand, there are many terms which do share some words with the query *high blood pressure*, such as *high head at term*, *fetal blood loss*, etc.; these terms would be found by a string matcher although they are conceptually distant from the query.

Human-defined synonyms or terminology thesauri have been tried as a semantic solution for the "too little" problem [2] [3]. It may significantly improve the mapping if the right set of synonyms or thesaurus is available. However, as Salton pointed out [4], there is "no guarantee that a thesaurus tailored to a particular text collection can be usefully adapted to another collection. As a result, it has not been possible to obtain reliable improvements in retrieval effectiveness by using thesauruses with a variety of different document collections".

Salton has addressed the problem from a different angle, using statistics of word frequencies in a corpus to estimate word importance and reduce the "too many" irrelevant terms [5]. The idea is that "meaningful" words should count more in the mapping while unimportant words should count less. Although word counting is technically simple and this idea is commonly used in existing information retrieval systems, it inherits the basic weakness of surface string matching. That is, words used in queries but not occurring in the term collection have no affect on the mapping, even if they are synonyms of important concepts in the term collection. Besides, these word weights are determined regardless of the contexts where words have been used, so the lack of sensitivity to contexts is another weakness.

We focus our efforts on an algorithmic solution for achieving the functionality of terminology thesauri and semantic weights without requiring human effort in identifying synonyms. We seek to capture such knowledge through samples representing its usage in various contexts, e.g. diagnosis texts with expert-assigned canonical terms collected from the Mayo Clinic patient record archive. We propose a numerical method, a "Linear