Unification Algorithms for Massively Parallel Computers[*]

Hiroaki Kitano
Center for Machine Translation, Carnegie Mellon University, Pittsburgh, PA 15213 U.S.A.
hiroaki@cs.cmu.edu
NEC Corporation, 2-11-5 Shibaura, Minato-ku, Tokyo, 108 Japan

ABSTRACT

This paper describes unification algorithms for fine-grained massively parallel computers. The algorithms are based on a parallel marker-passing scheme. The marker-passing scheme in our algorithms carry only bit-vectors, address pointers and values. Because of their simplicity, our algorithms can be implemented on various architectures of massively parallel machines without loosing the inherent benefits of parallel computation. Also, we describe two augmentations of unification algorithms such as multiple unification and fuzzy unification. Experimental results indicate that our algorithm attaines more than 500 unification per seconds (for DAGs of average depth of 4) and has a linear time-complexity. This leads to possible implementations of massively parallel natural language parsing with full linguistic analysis.

1. Introduction

This paper describes unification algorithms using parallel marker-passing scheme. The purpose of this paper is to show parallel unification algorithms which are simple enough to be implemented by massively parallel machines, and have some novel features.

Unification is a basic operation in computational linguistics. However, this operation is known to be computationally expensive, and thus is considered a major bottleneck in improving the performance of natural language processing systems. A search for efficient algorithms has been conducted by many researchers involving parallel algorithms such as [Yasuura, 1984]. However, theoretical lower-bound was shown by [Dwork et. al., 1984] that unifiability is log-space complete for P. This leads to [Knight, 1989]'s conclusion that use of massively parallel machines will not significantly improve the speed of unification. Then, why do we propose a parallel unification? We have three major reasons.

First, although theoretical limitation for speed up bas been shown for full unification, parallelization of unification actually improves performance of the entire system. This improvement of performance is a clear benefit for practical natural language processing systems, in particular for tasks like spoken language processing where real-time processing is essential. In addition, we propose parallel unification algorithms which attained a time-complexity of $o(D)$ where D is a depth of the deepest path in DAGs to be unified. We achieved this by assuming all disjunctions are pre-expanded into several DAGs so that each pair of DAGs does not contain disjunctions, and so that higher parallelism can be maintained through out the unification process. This is a reasonable assumption when we implement unification on massively parallel machines, where the basic implementation strategy is a memory-intensive approach allowing time-complexity to be converted into space-complexity. Thus, although we do not discover faster full unification with disjunction, we discovered a means to substantially speed up unification on the massively parallel machines.

Second, we designed our algorithm for massively parallel machines where each processor has relatively low processing capability. We only require each processing unit to have some basic operations and the capability to pass bit-markers, pointers to other processing units, and numeric values. This design decision aims at the accomplishment of two things — development of practical unification algorithms for massively parallel computers such as SNAP [Moldovan et. al., 1989] and Connection Machine [Hillis, 1985], and development of algorithms for specialized unification hardware such as unification chips or unification co-processors. Functionalities of massively parallel machines are severely limited due to the weak processing capability of each unit. Advantages of massively parallel machines for semantic processing, such as contextual priming, are widely recognized. However, in implementing serious natural language parsers, unification operation is essential. Unfortunately, we have not seen any algorithm which assumes low processing capability of each processor in massively parallel machines. Although some machines support high-level language, such as C or lisp, automatic parallelization does not guarantee efficiency of actual operations. Thus, designing unification algorithms for massively parallel machines has great impact on exploring maximum potential of these machines for natural language processing. One other reason is that, by assuming each processor has

[*]This work has been supported in part by the National Science Foundation under grant MIP-90/09109.

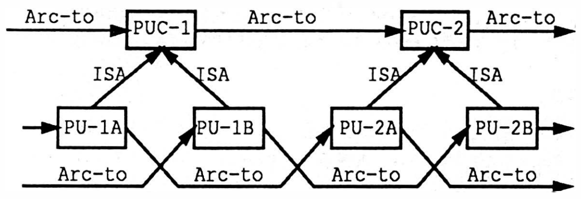

Figure 1: PU Class Nodes and PUs

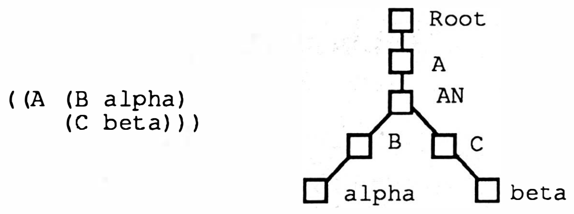

Figure 2: Representation of Nodes and Arcs

low computation power, our algorithms could be implementable as unification co-processor boards using numbers of less-powerful processors. A possibility for such a compact acceralator would be the clear benefit for the natural language community.

Third, our algorithms can easily entail some novel features such as multiple unification and fuzzy unification. These features have not been considered in past unification literature. It can also incorporate typed unification. Multiple unification is a unification between more than two trees or DAGs. Our algorithms enable this scheme without undermining its performance. Fuzzy unification allows unification of un-unifiable DAGs, but assigns a cost of violations. This would be useful for applications such as spoken language processing where handling of ungrammatical input is essential, because subtle ungrammaticalities can be overlooked.

2. Architecture, Representation and Notations

2.1. Architecture

We assume a parallel architecture where numbers of processing units are interconnected. The Processor Unit (PU) is a basic element of the system. It has its own processing capability and memory. This can be physical or logical, but, of course, we assume each unit is actually implemented as hardware. The Processor Unit Class (PUC) is a class of PUs which has several PUs as instances of the PUC. For each PUC, one PU is assigned to manage instances of the class. Figure 1 illustrates relations between PUCs and PUs. PUC-1 has instances PU-1A and PU-1B, and PUC-2 has instances PU-2A and PU-2B. This relation will be established when DAGs are loaded onto the unification co-processor.

We assume each PU's memory is is composed of a bit markers register, value register, and pointer memory for fan-in connections, fan-out connections, and address registers.

2.2. Represenation of Tree and DAGs

Trees or DAGs are represented as PUs and their connections. Each arc and node is assigned to each PU. Figure 2 shows how trees and DAGs are represented using PUs. In Figure 2, PUs are represented as square. Lines represent directed arcs. PUs in the middle of arcs represent labels of arcs. Each PU is connected by an Arc-to type link. When mapping feature structures on PUs, all PUs representing tree-0 or DAG-0 are marked with a marker 0, and all PUs representing tree-1 or DAG-1 are marked with a marker 1. PUs representing values have a marker V, and that of features have a marker F. Root PUs have a marker R.

2.3. Notations

The following notations will be used in describing algorithms:

PU(a,b,...,z): PU with specific markers set. PU(1,S,V) means that the PU has marker 1, S, and V. Negation can be used. For example, PU(1,S,-V) means PU has marker 1 and S set, but not V. Unspecified markers are don't care markers. Predicates can be used to specify conditions.

&PU(a,b,...,z): Address of PU which satisfies conditions specified.

Propagate: Propagation of markers through Arc-to link forward, i.e. direction from root to edge.

Back-Propagate: Propagation of markers through Arc-to link backward, i.e. direction from edge to root. This should not be confused with back-propagation in connectionist learning.

P-Address: Variable which can propagate or back-propagate an address of a PU.

The following instruction set will be used:

Propagate (Marker, Origin, Destination, Initial-action, Intermediate-action, Final-action): Propagate marker from origin to destination. Before propagation starts, do initial-action. At each PU during propagation, do intermediate-action, and at the destination PU, do final-action. In some special cases, destination is specified as 1. This means that markers are propagated only for one traverse.

Back-Propagate (Marker, Origin, Destination, Initial-action, Intermediate-action, Final-action): Back-propagation version of propagate instruction.

Mark(Marker,PU): Set marker to PUs. When PU is not specified (i.e. Mark(V)), the mark operation is performed to a current PU.

Set(Variable,Value): Set operator set a value specified in the second argument to the variable specified in the first argument. For example, Set(P-Address,&PU) sets an address of current PU to P-Address.

Connect(Arc-type,Origin,Destination): Create link of arc-type between origin and destination.

Other instructions such as **Create-Node(a,b,...,z), In(P-Address, From-Address), Equal(P-Address, &PU)**, and **GLB-Search(...)** will be explained in sections where they are used. In some cases, if-then-else control sequence is used for ease of understanding. However, obviously, this can be implemented using logical bit-marker operations such as (AND 1 2 4) followed by a propagation instruction, such as Propagate(P-Address,PU(4),PU(V)....). This case, (AND 1 2 4) is a logical operation that set marker 4 when markers 1 and 2 exist. This instruction sequence should be read as: if there are PUs such that PU(1,2), then propagate(P-Address,PU(1,2),PU(V)....).

3. Pseudo-Unification

Pseudo-unification or tree-unification is a unification between trees [Tomita and Knight, 1988]. The advantage of using pseudo-unification, instead of full-unification (or graph-unification), is that it can be implemented easier (less resource requirements and a simpler algorithm) and faster than full-unification. Yet, practically, pseudo-unification can cover a substantial range of linguistic phenomena. Actually, KBMT-89 [Nirenberg et. al., 1989] (a knowledge-based machine translation system based on LFG, and developed at the Center for Machine Translation at Carnegie Mellon University) was implemented using pseudo-unification.

3.1. The Algorithm

The algorithm which we describe in this section accounts for all non-disjunctive cases of pseudo-unification. Tree-0 and Tree-1 are unified (figure 3). Our algorithm for destructive tree unification consists of three parts:

1. Shared Node Detection
2. Failure Detection
3. Merging

3.1.1. Shared Node Detection

The goal of the shared node detection stage, or the common feature detection stage, is to set S markers to all nodes that are shared between trees. Step 1 carry out this stage.

Figure 3(a) shows the initial state of trees loaded into a PU network. First of all, an address of a PUC of a root PU of the tree-0 is set to P-Address. Then, P-Address is propagated until it gets to a PU which has V marker set. During this propagation, Check-Shared is conducted at each PU which P-Address traverses through. &ISA(Root) returns an address of the PUC of the Root PU. By the same token, &ISA(PU-0) returns an address of the PUC of the PU-0. The result is shown in 3(b). All shared PUs are indicated by solid circles. Some important markers on each PU are shown in brackets, but some markers are ignored due to diagram space.

3.1.2. Failure Detection

Next, we would like to detect conflicts. We assume that if two different value units are linked to the PUs both under the same PUC, and the PU is a shared arc unit, then unification should fail. Step 2 and 3 carry out this stage.

Back-Propagate starts from terminal nodes which are not shared. The purpose of this back-propagation is to identify pre-terminal PUs which are Arcs. In case of Figure 3, tree-0 and tree-1 are unifiable.

3.1.3. Merging

Since unifiability is assured in the failure detection stage, all we need is to merge two trees. Step 4, 5, 6, and 7 carry out this stage.

Back-propagation is used to search PUs which unshared leaves should be connected to. Figure 3(c) indicate PUs involved in this process. Propagation starts from PU(1,V,-S) and goes up until it meets a PU which is shared. These PUs are places where unshared branches should be connected. Next, propagate an address of each PUs for one traverse. Now, relevant PUs have an address of PUs which should be connected. Connect a PU with markers P-Address, 0, and B and a PU with markers P-Address, 1, and T with Arc-to. Propagate marker 0 from PU with P-Address, 0, and B. As a result, we get a unified tree consisting of PUs marked with 0.

4. Full-Unification

Although pseudo-unification does quite a good job in most practical cases, there are cases where graph-unification is necessary. Lack of the re-entrance in the pseudo-unification forces grammar writers to subdivide their grammar rules to cope with various cases of re-entrance because re-entrant structure must be expanded to trees. This section presents full-unification (destructive version).

```
1: Propagate(P-Address, Root, PU(V), Set(P-Address,&ISA(Root)), Check-Shared, nil)
   Check-Shared: If there is a PU (PU-1), under the same PUC, such that PU(1,In(P-Address, From-Addresses)),
   then Mark(S), Mark(S,PU-1), and Set(P-Address,&ISA(PU-0)), else abort propagation.
2: Back-Propagate(PT,PU(V,-S),1,nil,nil,Mark(PT))
3: If there is a PU such that PU(PT,S), then unification is a failure.
4: Back-Propagate(P-Address,PU(1,V,-S),PU(S), Set(P-Address, &PU(1,V,-S)), nil, Mark(B,PU(S,P-Address)))
5: Propagate(P-Address, PU(B), 1, Set(P-Address,&PU(B)), nil, Mark(T))
6: Connect(Arc-to, PU(P-Address,0,B), PU(P-Address,1,T))
7: Propagate(0, PU(0,B), PU(V), nil, Mark(0), Mark(0))
```

Table 1: Pseudo-Unification Algorithm

4.1. The Algorithm

In full-unification, we only need to add merging of arcs which is not covered in the pseudo-unification algorithm.

1. Shared Node Detection Stage
2. Failure Detection Stage
3. Merging Stage
4. Arc Merging Stage

DAG-0 and DAG-1 are unified (figure 4). In figure 4(a), shared nodes are detected and indicated by solid circles. Figure 4(b) and (c) shows the merging stage. In figure 4(b) top and bottom PUs are marked and then merged in figure 4(c). Up to this point, we can simply apply algorithms presented for pseudo-unification. However, in unifying DAGs, we must take into account the existence of unshared arcs which are in between shared PUs that are not handled in the merging stage in the pseudo-unification algorithm. An arc merging stage merges such arcs into the DAG. The algorithm presented here covers most of practical cases of non-disjunctive graph unification, but there are some cases which the algorithm does not provide correct result. However, even in such cases, a simple post-processing can modify the graph to provide correct results.

4.1.1. Arc Merging

The arc merging stage for the destructive graph unification is shown in table 2. For all nodes with marker F and 1, but not S, propagate marker E. Propagation stops when it arrives at a node marked S. Back-Propagate P-Address until it arrives at a node with S. For all nodes which have S and P-Address, mark B. Propagate marker B for one traverse, and mark destination node with T. Connect a node with markers P-Address, 0, and B and a node with markers P-Address, 1, and T with Arc-to. Propagate marker 0 from a node with P-Address, 0, and B.

5. Nondestructive Unification

So far we have been discussing destructive unification algorithms where represented feature structures are destroyed in the process of unification. Obviously, this would be problematic because (1) it destroys the original feature structure even when the feature structure needs to have its unifiability examined against more than one feature structure, and (2) destructive unification involves over-copying and early-copying [Wroblewski, 1988]

In this section, we further extend algorithms presented so far, and present a nondestructive graph unification algorithm. To implement the nondestructive graph unification, new nodes and arcs need to be created by assigning them on empty PUs. Instead of passing only P-Address, as we have been using so far, we pass P-Address and N-Address (an address of the newly assigned PU). Given two DAGs, the algorithm in table 3 creates a new DAG as a result of unification.

Figure 5, 6 and 7 show intermediate processes. DAG-0 and DAG-1 are unified and result in DAG-2. Figure 5 is a state after the shared node is detected. Solid circles indicate PUs for shared nodes. In figure 6, all unifiable branches of DAG-0 and DAG-1 are merged to DAG-2 to create New DAG-2. In figure 7 intermediate arcs are merged into DAG-2, and create Final DAG-2. One big difference between nondestructive graph unification and destructive unification is that, in nondestructive unification, new PUs are assigned when unifiable subgraphs from DAG-0 and DAG-1 are merged into DAG-2, whereas destructive unification is simply marked with 0 at the merging process. For this reason, **Append-New-Node** assigns a new PU for each node merged to DAG-2, and connects it to existing DAG-2 structure. Then, pointers to the merged PU in DAG-2 and an equivalent PU in DAG-0 or DAG-1, are propagated so that the next PU can be connected to them.

6. Typed Unification

The ψ-terms proposed in [Ait-Kaci, 1984] are similar to the feature structure, but the functor is retained. This provides a filter under unification because two feature structures with incompatible functors cannot be unified. When a conflict is detected, it is resolved by finding the greatest lower bound (GLB) of two items

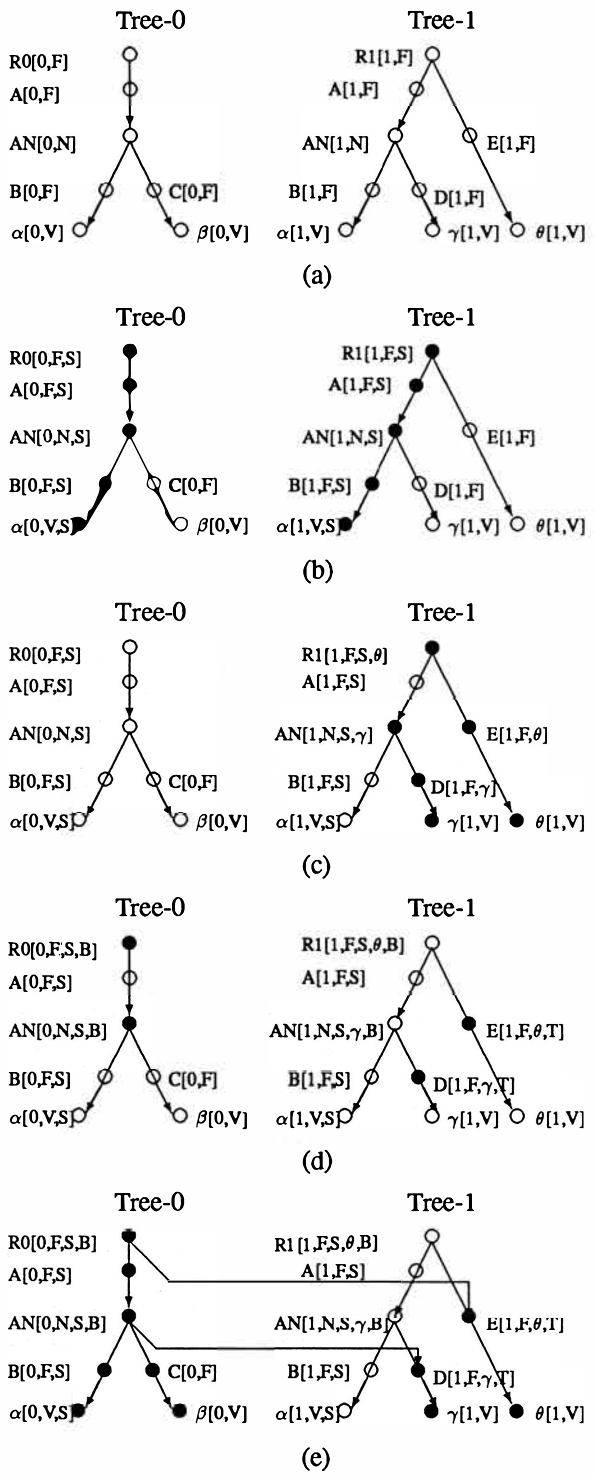

Figure 3: Pseudo-Unification

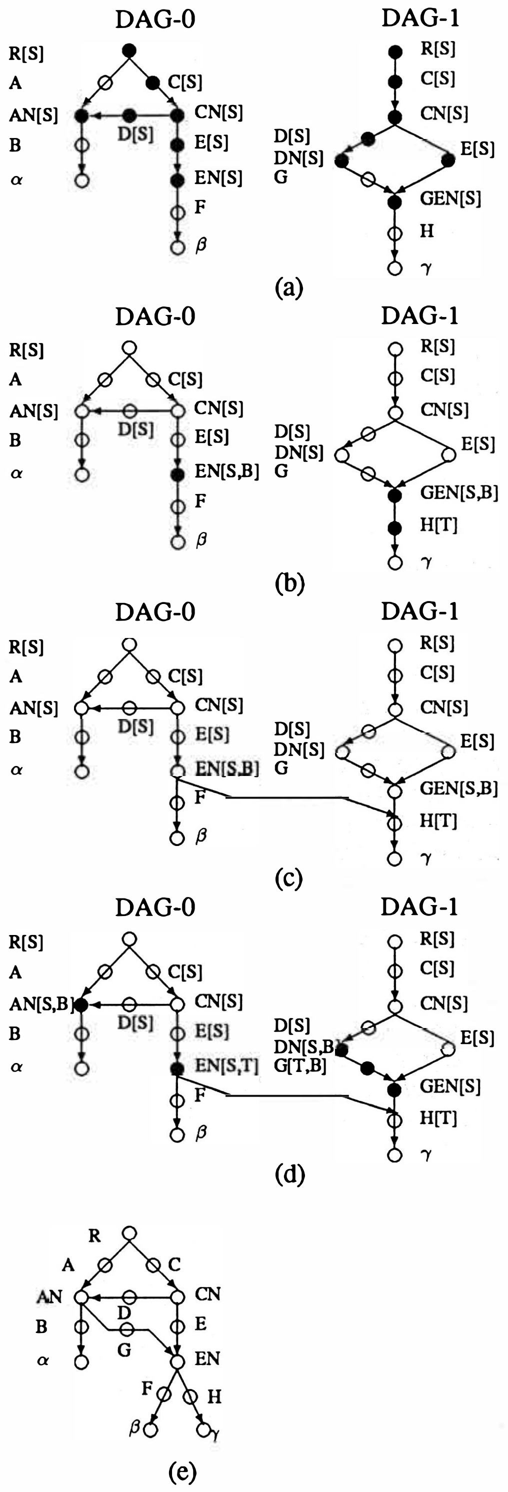

Figure 4: Graph Unification

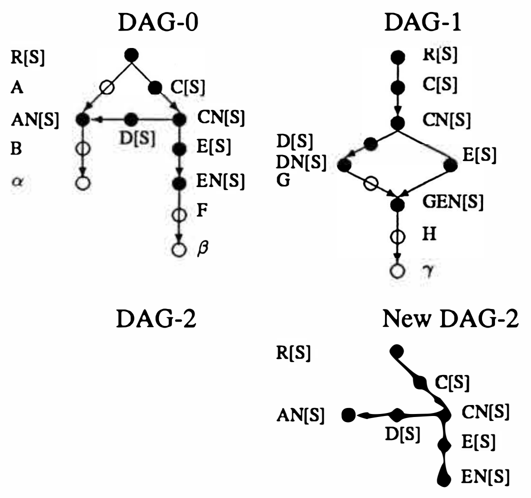

Figure 5: Nondestructive Graph Unification: Detect Shared Nodes

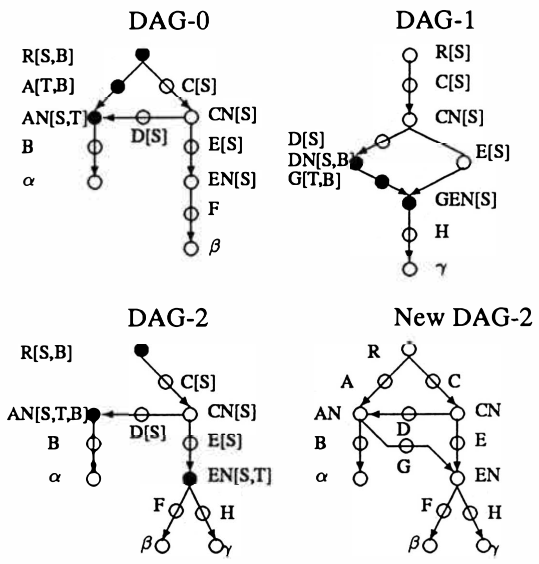

Figure 7: Nondestructive Graph Unification: Merge Internal Arcs

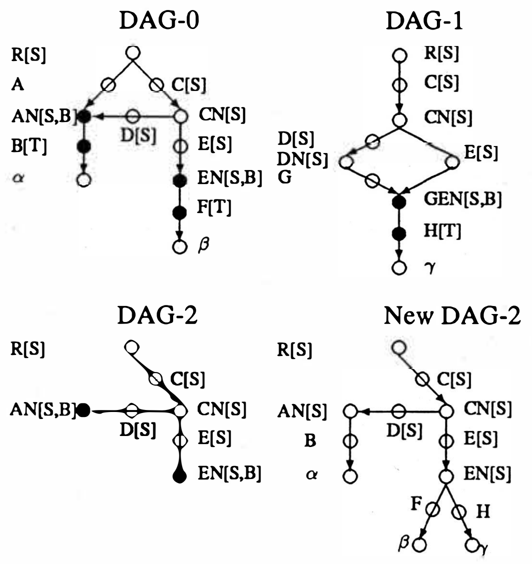

Figure 6: Nondestructive Graph Unification: Merge

```
8:  Propagate(E,PU(F,1,-S),PU(S),nil,nil,Mark(E))
9:  Back-Propagate(P-Address, PU(E), PU(S), Set(P-Address,&PU), Set(P-Address,&PU), Mark(B))
10: Propagate(P-Address, PU(B), 1, Set(P-Address,&PU), nil, Mark(T))
11: Connect(Arc-to, PU(P-Address,0,B), PU(P-Address,1,T))
12: Propagate(0, PU(P-Address,0,B), PU(V), nil, Mark(0), Mark(0))
```

Table 2: Arc Merging Stage in Destructive Graph Unification

```
1:  Propagate(P-Address N-Address, Root, PU(V), Set(P-Address,&ISA(Root)) Set(N-Address,&ISA(New-Root)),
    Check-Shared, nil)
    Check-Shared: If there is a PU (PU-1), under the same PUC, such that PU(1,In(P-Address,From-Addresses)),
    then Mark(S), Mark(S,PU-1), Set(P-Address,&ISA(PU-0)), Create-Node(2,S,N-Address),
    Connect(Arc-to,PU(N-Address),PU(2,S,N-Address)), and Set(N-Address,&PU(2,S,N-Address)),
    else abort propagation.
2:  Back-Propagate(PT,PU(V,-S),1,nil,nil,Mark(PT))
3:  If there is a PU such that PU(PT,S), then unification is failure.
4:  Back-Propagate(P-Address, PU(1,V,-S), PU(S), Set(P-Address,&PU(1,V,-S)), nil, Mark(B,PU(S,P-Address)))
5:  Propagate(P-Address, PU(B), 1, Set(P-Address,&PU(B)), nil, Mark(T))
6:  Propagate(P-Address N-Address, PU(B), PU(V), Set(P-Address,&PU(2,P-Address), Append-New-Nodes, nil)
    Append-New-Nodes: If a cuurent PU is PU(0,-S) or PU(1,-S),
    then Create-Node(2,N-Address), Connect(Arc-to,PU(P-Address),PU(2,N-Address)),
    Set(N-Address,&PU(2,N-Address)), and Set(P-Address,&PU(2,P-Address)),
    else abort propagation.
7:  Propagate(E, PU(F,1,-S), PU(S), nil, nil, Mark(E))
8:  Back-Propagate(P-Address, PU(E), PU(S), Set(P-Address,&PU), Set(P-Address,&PU), Mark(B))
9:  Propagate(P-Address, PU(B), 1, Set(P-Address,&PU), nil, Mark(T))
10: Propagate(P-Address N-Address, PU(B), PU(E), Set(P-Address,&PU(2,P-Address), Append-New-Nodes,
    Connect(Arc-to,N-Address,PU))
```

Table 3: Non-destructive Graph Unification Algorithm

in the taxonomic hierarchy. One way of implementing this scheme is to incorporate a search of hierarchy at the shared node detection. Perform the instructions shown in table 4 immediately after the shared node detection stage:

GLB-Search is a special instruction where propagation of markers start from nodes with V markers set but not S markers, and P-Address is propagated through ISA hierarchy downward. At each PU during the traversal, the current PU's address is set to GLB-Address, and it is propagated through ISA link upward. When GLB-Address arrives at a PU with V marker set but not S marker, it means there are GLB between the origin PU and the destination PU. Now, GLB-Search is conducted backwards, starting from the previous destination PU. This gives an address of the GLB PU to the originated PU. Thus, both PUs have an addres of the GLB PU. When one PU (PU-a) is under the other PU's (PU-b) ISA hierarchy, a GLB PU should be PU-a. Using the same mechanism, an address of PU-a is given to both PUs. However, this time GLB-Address propagation is not involved since GLB-PU itself is a destination PU. At the merging stage, PUs representing GLB should be merged instead of PUs in DAG-0 or DAG-1 (when GLB is one of the PU in DAG-0 or DAG-1, the PU in these DAGs can be merged). This can be done by using pointers to the GLB PUs propagated to PUs in DAG-0 and DAG-1. This mechanism enables typed unification.

7. Augmenting Unification

7.1. Fuzzy Unification

Traditionally, unification has been a logical operation, and thus, its failure resulted in hard rejection. We propose an alternative scheme called a *fuzzy unification* or a *soft rejection unification*. Contrary to the traditional unification which only returns nil when failed, a new unification scheme returns a partially unified feature structure and a value which indicates the degree of failure. In the soft rejection unification, each arc is assigned with a value which is accumulated when unification in its subgraph was failed. Meanings of the value can vary depending upon application and specific implementation. It can be a cost of violation or a probability measure of which violation will happen.

Unification operation with such property is significant for many applications which require robust parsing. For example, speech input processing requires integrated processing of a speech recognition module and linguistic parsing in order to limit the scope of search (reduce perplexity) which in turn improves recognition

Typed 1:	GLB-Search(P-Address GLB-Address, PU(V,-S), PU(V), Set(P-Address,&PU) Set(GLB-Address,&PU), Set(GLB-Address,&PU), nil)
Typed 2:	GLB-Search(P-Address GLB-Address, PU(V,-S,P-Address), PU(V), Set(P-Address,&PU) Set(GLB-Address,&PU), Set(GLB-Address,&PU), nil)
Typed 3:	Mark(S, PU(Equal(P-Address,&PU)))

Table 4: Type Checking in Typed Unification

rate. While spoken language inherently involves erroneous sentences, use of the traditional hard-rejection-type unification cannot be applied as it is — parsing needs to proceed even with minor syntactic failures. Some relaxation techniques have been proposed for detecting and overlooking minor errors by allowing some of the constraints to be ignored. However, traditional relaxation methods require multiple unification operations to check against sets of constraint equations, resulting in substantial overhead against conventional unification-based parsing. In addition, these relaxation methods did not assign weights or the probability that certain violations will happen. This would have adverse effects in reducing perplexity, because all possible errors are granted or predicted with equal weights. Since the likelihood of certain violations happening can be statistically obtained, providing *a priori* probability of such violations would help improve recognition rate.

For example, in a sentence *John want to attend the conference.* Although *John* and *want* cause violation in the third-person-present-singular constraint, we do not want that parse to be aborted since its semantics can be easily recovered in a post-processing. However, we want to add a cost to such parse so that if a speech recognition module provided two word hypotheses of *want* and *wants*, *John wants ...* would be selected as a most probable hypothesis.

This extension is trivial in our algorithm. The failure detection stage is revised as seen in table 5.

ADD-value adds values of markers at the root node. Alternatively, more sophisticated computation, instead of ADD, can be used to determine the degree of unifiability.

7.2. Multiple Unification

Traditionally, unification has been defined as an operation between between two DAGs; it takes two DAGs and returns a unified DAG or nil when failed. We extend this notion and propose multiple unification — unification of more than two DAGs. This extension would benefit processing of linguistic analysis which uses N-branching trees where N > 2. Although such N-branching trees have been commonly used in liguistic analysis, unification operations to directly handle these analyses have not been proposed. Multiple-unification would unify feature structures propagated from each branch of trees simultaneously, and result in a considerable reduction in computational cost. This would benefit, particularly, parsing of Japanese where each case-marked NP can be subcategorized by VP at the top-level.

In our algorithm, multiple-unification is handled simply by assigning M markers for each tree or DAG identification where binary unification uses only markers 0 and 1. The algorithm itself should be changed by re-locating the failure detection stage to the end of the entire process, so that all merging is completed when failure detection is performed:

1. Shared Node Detection Stage
2. Merging Stage
3. Arc Merging Stage
4. Failure Detection Stage

Since unifiability of DAGs must be tested for all combinations, it is more efficient to merge first rather than to test unifiability N(N-1)/2 times before the merging stage.

8. Efficiency of the Algorithm

8.1. A Brief Complexity Analysis

The algorithm is efficient. Let's assume that we have DAGs with N nodes, depth D and width W. Shared node detection stage requires propagation of markers from roots to each value node. Since this can be done in parallel, computational cost is approximately D $\times$ (P + CSH) whereas P is a time required for propagation of a marker for one depth, and CSH is a cost for detecting wether two nodes has a same PUC. In the failure detection stage, back-propagation of markers for one travarsal backward is requried. The cost is P. The merging stage requires $2 \times D \times P + P$ at worst cases. The arc merging state costs $3 \times D \times P + P$ at worst cases. Thus, in total, $6 \times D \times P + 3 \times P + C + CSH \times D$ is the computational cost of the full unification in our algorithm with 2N-1 processors. Thus, in rough estimation, a complexity of the algorithm is of order of O(D). When the number of processors (M) is less than 2N-1, efficiency might degrade depending upon allocation of nodes onto processors. If we can allocate nodes in a same path to one processor, we only require

```
2: Back-Propagate(PT, PU(V,-S), 1, nil, nil, Mark(PT))
3: If there is a PU with PT and S, then Back-Propagate(Value, PU(PT,S), PU(R), nil, nil, ADD-Values)
```

Table 5: The failure detection stage of the fuzzy unification

W processors to maintain the efficiency close to the estimation above. This is because a marker-propagation in the same path is sequential. However, W processor condition may degrade its efficiency due to synchronization required for marker-propagation at arc merging and branching crossing processor boundary. The worst case of W processor condition is $O(\frac{D \times N}{W})$, but, of course, this can be easily avoided by designing memory allocation optimally. When unification failed, then the computational cost is $D \times P$ (cost for shared node detection) and P (cost for failure detection). Let S be a success rate of the unification (which is usually between 40% to 20%), expected computational costs will be: $S \times (D \times (6 \times P + CSH) + 3 \times P + C) + (1 - S) \times (D \times (P + CSH) + P)$

8.2. Experimental Results

We have implemented our algorithms on a simulator for a fine-grained parallel machine which assumes actual computation time for each instruction. To unify the DAGs shown in figure 4, the destructive graph unification took 1957 micro seconds (510 unification per second). The rate of performance degradation is about 330 micro seconds for each additional depth.

Table 6 shows numbers of each instruction executed, and computational cost in one example of the unification operation. Statistics clearly show that the shared node detection stage is the most computationally expensive. Particularly, the extensive numbers of address propoagation and bit check operation are two major causes of the computational cost. The estimated time for propagating an address for one traverse is set to 15 micro seconds, which can be reduced to 3 micro seconds on SNAP architecture, thus attining substantial speed up. Algorithms described in this paper has been implemented on the SNAP massively parallel computer as a part of the joint project between Carnegie Mellon University and University of Southern California.

9. Conclusion

This paper described unification algorithms using marker-passing. We only assumed passing of bit-markers, pointers to PUs, and values. Operations required for our unification algorithms are simple and easily implementable in massively parallel machines which use numbers of processing units with a relatively low-processing capability. Actually, operations and marker-passing schemes assumed in this paper are readily available in actual massively parallel machines such as SNAP [Moldovan et. al., 1989].

The algorithms are simple. It requires passing of bit-markers and addresses to PUs for conventional unifications. Despite its simplicity, our algorithms cover all non-disjunctive cases of unification of trees and most practical cases of unification of graphs. However, investigations should be conducted to identify a class of graphs which our algorithms can and cannot handle. Should a class of graphs which can be handled by our algorithms cover a class of graphs appearing in natural language processing, our algorithms can be a very powerful scheme of parallel unification processing. Typed-unification, originally proposed by [Ait-Kaci, 1984], can be naturally incorporated in our algorithms since our algorithms are based on marker-passing which is originally proposed for an intersection search. Conformity with lattice search is obvious.

The algorithms are efficient on massively parallel machines. Even in nondestructive graph unification, it requires only nine propagations and back-propagations and some checking instructions. For the graphs with depth D, unification should be done at $6 \times D \times P + 3 \times P + C + CSH \times D$ whereas P is a time required for propagation of a marker for one arc traverse, C is a total cost of condition checks, and CSH is a total cost for detecting whether two nodes has a same PUC. Thus, the complexity is of order of O(D). The processor requirement is linear to the size of graphs. This simple estimation indicates that our algorithm would be fast enough for practical applications.

Novel features such as multiple-unification and fuzzy unification adds new dimensions to our unification algorithms. Also, our unification algorithms are easily augmented for typed unification. In practical cases, needs for unification of more than two feature structures are commonly observed, yet this has not been proposed in the past. Use of multiple-unification reduces the amount of copying and thereby improves performance. Fuzzy unification would be a very useful concept for applications such as spoken language processing. Instead of rejecting at the detection of unification failure, the fuzzy unification adds a cost of violation in such cases, and allows processing of violated hypotheses to proceed. Where application domains inevitably involve ungrammatical inputs, the fuzzy-unification would be a powerful extension to the traditional unification approach.

Acknowledgement

The author would like to thank Hitoshi Iida, Hideto Tomabechi, Dan Moldovan, and members of the SNAP

Stage	Propagate Markers	Propagate Address	Bit Check	Address Check	Store Address	Time (micro-seconds)
Shared Node Detection	0	74	157	15	0	1778
Failure Detection	1	0	4	0	0	30
Merge	0	2	14	0	2	108
Internal Arc Merge	0	2	8	0	2	78
Total	1	78	183	15	4	1994

Table 6: Number of Instructions at each stage of unification

project for discussions, and Masaru Tomita and Jaime Carbonell for their supports.

References

[Ait-Kaci, 1984] Ait-Kaci, H., *A Lattice Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures,* Ph.D. Thesis, University of Pennsylvania, 1984.

[Dwork et. al., 1984] Dwork, C., Kanellakis, P. and Mitchell, J., "On the Sequential Nature of Unification," *Journal of Logic Programming,* vol. 1, 1984.

[Hillis, 1985] Hillis, D., *The Connection Machine,* The MIT Press, Cambridge, 1985.

[Knight, 1989] Kevin, K., "Unification: A Multi-Disciplinary Survey," *ACM Computing Surveys, Vol. 21, Number 1,* 1989.

[Moldovan et. al., 1989] Moldovan, D., Lee, W., and Lin, C., *SNAP: A Marker-Propagation Architecture for Knowledge Processing,* University of Southern California Technical Report CENG 89-10, 1989.

[Nirenberg et. al., 1989] Nirenberg, S. (Ed.), *Knowledge-Based Machine Translation,* Center for Machine Translation Project Report, Carnegie Mellon University, 1989.

[Pollard and Sag, 1987] Pollard, C. and Sag, I., *An Information-based Syntax and Semantics,* Volume 1., Chicago University Press, 1987.

[Tomita and Knight, 1988] Tomita, M. and Kevin, K., "Pseudo-Unification and Full Unification," CMU-CMT-88-MEMO, 1988.

[Wroblewski, 1988] Wroblewski, D., "Nondestructive Graph Unification," in *Proceedings of AAAI-88,* 1988.

[Yasuura, 1984] Yasuura, H., "On Parallel Computational Complexity of Unification," in *Proceedings of the International Conference on Fifth Generation Computer Systems,* 1984.