12-1-1997

FASTER MUSE CSP ARC CONSISTENCY ALGORITHMS

Mary P. Harper
Purdue University School of Electrical and Computer Engineering

Christopher M. White
Purdue University School of Electrical and Computer Engineering

Randall A. Helzerman
Purdue University School of Electrical and Computer Engineering

Stephen A. Hockema
Purdue University School of Electrical and Computer Engineering

FASTER MUSE CSP ARC CONSISTENCY ALGORITHMS

MARY P. HARPER
CHRISTOPHER M. WHITE
RANDALL A. HELZERMAN
STEPHEN A. HOCKMA

TR-ECE 97-14
DECEMBER 1997

SCHOOL OF ELECTRICAL
AND COMPUTER ENGINEERING
PURDUE UNIVERSITY
WEST LAFAYETTE, INDIANA 47907-1285

Faster MUSE CSP Arc Consistency Algorithms*

Mary P. Harper, Christopher M. White,

Randall A. Helzerman, and Stephen A. Hockema

School of Electrical and Computer Engineering

1285 Electrical Engineering Building

Purdue University,

West Lafayette, IN 47907

'This research was supported by a grant from the Intel Research Council and the National Science Foundation under Grant No. IRI-9704358.

Abstract

MUSE CSP (Multiply *SE*gmented Constraint Satisfaction Problem) [5, 6] is an extension to the constraint satisfaction problem (CSP) which is especially useful for problems that segment into multiple instances of CSP that share variables. In Belzerman and Harper [6], the concepts of MUSE node, arc, and path consistency were defined and algorithms for MUSE arc consistency, MUSE AC-1, and MUSE path consistency were developed. MUSE AC-1 is similar to the CSP arc consistency algorithm AC-4 [11]. Recently, Bessikre developed a new algorithm, AC-6 [1], which has the same worst-case running time as AC-4 and is faster than AC-3 and AC-4 in practice. In this paper, we focus on developing two faster MUSE arc consistency algorithms: MUSE AC-2 which directly applies Bessikre's method to improve upon MUSE AC-1, and MUSE AC-3, which uses our new "lazy" evaluation method for keeping track of the additional sets required by the MUSE approach. These new algorithms decrease the number of steps required to achieve arc ccnsistency in randomly generated MUSE CSP instances when compared to MUSE AC-1.

Keywords: Problem Solving, Constraint Satisfaction, MUSE arc consistency.

1 Introduction

Constraint satisfaction, with a rich history in Artificial Intelligence, provides a convenient way to represent and solve certain types of problems. In general, these are problems that can be solved by assigning mutually compatible values to a predetermined number of variables under a set of constraints.. This approach has been used in a variety of disciplines including machine vision, belief maintenance, temporal reasoning, graph theory, circuit design, diagnostic reasoning, and natural language processing.

Constraint Dependency Grammar (CDG) parsing, as introduced by Maruyama [8, 9, 10], was framed as a constraint satisfaction problem (CSP); the parsing rules are the constraints and the solutions are the parses. Unfortunately, the CSP approach does not support the simultaneous analysis of sentences with multiple alternative lexical categories (e.g., ***can*** is a noun, verb, or modal) nor multiple feature values (e.g., *the* as a determiner can modify nouns that are third person singular or third person plural), nor the simultaneous parsing of sentences contained in a word graph produced by a speech recognizer. Hence, Harper and Helzerman [3] adapted the parsing algorithm to support the simultaneous parsing of alternative sentences resulting from lexical or feature ambiguity or from word segmentation ambiguity during speech recognition. From this work, the concept of a MUSE CSP (Multiply *SE*gmented Constraint Satisfaction *P*roblem) [5, 6] was developed to support the efficient simultaneous processing of multiple alternative CSP problems. A MUSE CSP is defined as follows:

Definition **1** (MUSE CSP)
N $= \{i, j, \ldots\}$ *is the set of* nodes *(or variables), with* $|N| = n$,
Σ $\subseteq 2^N$ *is a set of* segments *with* $|\Sigma| = s$,
L $= \{a, b, \ldots\}$ *is the set of* labels, *with* $|L| = l$,
L_i $= \{a | a \in L$ *and* (i, a) *is admissible in at least one segment*$\}$,
$R1$ *is a unary constraint, and* (i, a) *is admissible if* $R1(i, a)$,
$R2$ *is a binary constraint,* $(i, a) - (j, b)$ *is admissible if* $R2(i, a, j, b)$

The segments in Σ are the different sets of nodes representing CSP instances which are combined to form a MUSE CSP. Since a MUSE CSP is represented as a directed acyclic graph, segments are defined as paths through the MUSE CSP from a special purpose start to end node. Helzerman and Harper [6] also defined the concepts of MUSE node, arc, and path consistency and developed algorithms for MUSE arc consistency, MUSE AC-1, and MUSE path consistency, MUSE PC-1. These algorithms are similar to the CSP arc consistency algorithm AC-4 [11] and path consistency algorithm P'C-4 [2]. Although AC-4 has a worst-case running time of $\Theta(el^2)$ (where e is the number of constraint arcs), AC-3 [7] with a worst-case running time of $\Theta(el^3)$ often performs better than AC-4 in practice. Recently, Bessikre developed a new algorithm, AC-6 [1], which has the sarne worst-case running time as AC-4 and is faster than AC-3 and AC-4 in practice. In this paper, we focus on providing faster MUSE arc consistency algorithms by applying techniques similar to those developed by Bessikre.

MUSE arc consistency, as the focus of this paper, is defined as follows:

Definition 2 (MUSE Arc Consistency) *An instance of MUSE CSP is said to be MUSE* arc consistent *if and only if for every label a in each domain L_i there is at least one segment σ whose nodes' domains contain at least one label b for which the binary constraint R2 holds, i.e.:*

$$\forall i \in N : \forall a \in L_i : \exists \sigma \in \Sigma : i \in \sigma \wedge \forall j \in \sigma : j \neq i \Rightarrow \exists b \in L_j : R2(i,a,j,b)$$

When enforcing arc consistency in a CSP, a label a $\in L_i$ can be eliminated from node i whenever any other domain L_j has no labels that together with a satisfy the binary constraints. However, in a MUSE: CSP, before a label can be eliminated from a node, it must be no longer supported by the arcs of every segment in which it appears, as required by the above definition. Hence, there is more information that must be tracked in any MUSE arc consistency algorithm.

MUSE arc consistency is enforced by removing those labels in each L; that violate the conditions of Definition 2. MUSE AC-1 [6] builds and maintains several data structures defined in Figure 1, some that are similar to those used by AC-4 and some that are new (i.e., those below the double line in Figure 1), to allow it to correctly perform this operation. As in AC-4, MUSE AC-1 keeps track of how much support each label a $\in$ L; has from the labels in L_j by counting the number that are compatible with a and storing that number in Counter$[(i,j),a]$. The algorithm also keeps track of the set of labels in L, that are compatible with $a \in L_i$ as the set S$[(i,j),a]$. In AC-4, if Counter$[(i,j),a]$ becomes zero, a can be immediately removed from L; because a could never appear in any solution. However, in the case of MUSE arc consistency, even though a does not participate in a solution for any of the segments that contain i and j, there could be another :segment for which a would be perfectly legal. A label cannot become globally inadmissible until it is incompatible with every segment. Hence, in MUSE CSP, if Counter$[(i,j),a]$ is zero, the algorithm must record the fact that a is illegal in the segment involving the variables i and j and propagate that information throughout the MUSE CSP.

MUSE AC-1 uses the fact that the MUSE CSP is a DAG to determine when labels become inadmissible either within a segment or globally. To do this, the algorithm uses four new sets: Prev-Support, Next-Support, Local-Prev-Support, and Local-Next-Support, which are defined in Figure 1. For each variable i and j, $j \neq i$, and label $a \in$ L,, MUSE AC-1 keeps track of the sets Prev-Support$[(i,j),a]$ and Next-Support$[(i,j),a]$. Prev-Support$[(i,j),a]$ keeps track of those variables that precede j and support at least one label in L, and the label a in L_i; whereas, Next-Support$[(i,j),a]$ tracks variables that follow j and meet the support criteria. If, during MUSE AC-1, either of these sets become empty, then a is no longer consistent with any segment involving the variables i and j. This segment information is very important for updating the sets that track the global admissibility of a label, which is monitored by the Local-Prev-Support and Local-Next-Support sets. For each variable i and each label a $\in L_i$, Local-Prev-Support(i,a) keeps track of the variables that precede i and support the label a $\in L_i$; whereas, Local-Next-Support(i,a) keeps track of those that follow i and support the label a. If either local set becomes empty, then a is no longer a part of any MUSE arc consistent instance and should be eliminated from L_i because it is globally inadmissible.

MUSE AC-1 uses the Counter array and S sets to keep track of the labels that are supported in the MUSE CSP. If Counter$[(i,j),a]$ becomes 0, then $[(i,j),a]$ is placed on *List* (as in AC-4), indicating that a is inadmissible in all segments containing the variables i and j. When $[(i,j),a]$ is popped off of List during arc consistency, Counter$[(j,i),b]$ is decremented for all $(j,b) \in S[(i,j),a]$ (possibly resulting in $[(j,i),b]$ being placed on List). The fact that $a \in L_i$ is unsupported by the variable j is also used to determine whether other segments disallow $a \in L_i$. Because of a's loss of support by the variable j, any variable k that precedes j in the DAG must determine whether j was its only successor supporting $a \in L_i$. In addition, any variable k that follows j in the DAG must determine whether j was its only predecessor supporting $a \in L_i$. This is done by determining whether Next-Support$[(i,k),a]$ (if j was a successor) or Prev-Support$[(i,k),a]$ (if j was a predecestor) becomes empty after (i,j) is removed from the set. If the set becomes empty, then a is also inadmissible in the segment involving i and k; if $M[(i,k),a] \neq 1$, then the tuple $[(i,k),a]$ is placed on *List* and $M[(i,k),a]$ is set to 1. Finally, if j directly precedes i in the MUSE DAG, then (i,j) is removed from Local-Prev-Support(i,a), and if j directly follows i, then (i,j) is removed from Local-Next-Support(i,a). If either local set becomes empty, then a is removed from L_i and additional tuples are stored on List to clean up the Counters and DAG sets associated with the label $a \in L_i$.

The worst-case running time and space complexity of MUSE AC-1 is $O(n^2l^2 + n^3l)$, where n is the number of nodes in a MUSE CSP and l is the domain size. By comparison, the worst-case running time and space complexity for CSP arc consistency is $O(n^2l^2)$, assuming that there are n^2 constraint arcs. Note that for applications where $l = n$, or Σ is a planar DAG (in terms of Prev-Edge and Next-Edge, not E), the worst-case running times of the algorithms are the same order. Because MUSE AC-1 inherits AC-4's poor average-case performance, we can improve the average-case running time of MUSE AC by using techniques similar to those used by Bessière to develop AC-6. We first review Bessikre's AC-6 algorithm and then describe two new MUSE arc consistency algorithms: MUSE AC-2, which directly applies Bessikre's method to improve upon MUSE AC-1, and MUSE AC-3, which, in addition to Bessikre's method, uses our "lazy" evaluation method for keeping track of the Prev-Support, Next-Support, Local-Prev-Support, and Local-Next-Support sets.

2 AC-6

To improve the average-case performance while maintaining the same worst-case time complexity of AC-4, Bessikre [1] developed AC-6, an algorithm conceptually similar to AC-4 but which avoids much of the work always carried out by AC-4. AC-6 assumes that the labels in the domains are stored in a data structure that enforces an ordering on the labels and supports the following

Notation	Meaning
E	All node pairs (i, j) such that there exists a path of directed edges in G between i and j. If $(i, j) \in$ E, then $(j$, i) $\in$ E.
L_i	$\{a \mid a \in$ L and $(i$, a) is permitted by the constraints (i.e., admissible)$\}$
$R2(i, a, j, b)$	$R2(i$, a, j, b) = 1 indicates the admissibility of $a \in$ L; and $b \in L_j$ given binary constraints.
Counter$[(i, j), a]$	The number of labels in L_j that are compatible with a $\in L_i$ (used only by MUSE AC-1).
S$[(i, j), a]$	$\{(j,b) \mid R2(i$, a, j, b) = 1) (MUSE AC-1) or $\{(j,b) \mid (i$, a) with a being the smallest value in L_i supporting $(j,b)\}$ (MUSE AC-2, MUSE AC-3).
M$[(i, j), a]$	M$[(i, j)$, a] = 1 indicates that the label a is not admissible for (and has already been eliminated from) all segments containing i and j.
	The set of node pairs (i, j) such that there exists a directed edge from i to j.
List	A queue of arc support to be deleted.
Next-Edge,	The set of all node pairs (i,j) such that there exists a directed edge (i, j) $\in$ G. It also contains $(i$, end) if there is no x $\in$ N such that $(i, x) \in$ G.
Prev-Edge,	The set of all node pairs (j, i) such that there exists a directed edge $(j, i) \in$ G. It also contains (start,i) if there is no x $\in$ N such that $(x$, i) $\in$ G.
Local-Prev-Support$(i$, a)	A set initialized to $\{$(i, $x) \mid (i$, x) $\in$ E $\wedge$ (x,i) $\in$ Prev-Edge,) $\cup$ $\{$(i, start) $\mid$ (start, i) $\in$ Prev-Edge,). After arc consistency, if $(i, j) \in$ Local-Prev-Support$(i$, a) and $j \neq$ start, a must be compatible with at least one of j's labels.
Local-Next-Support$(i$, a)	A set initialized to $\{(i, x) \mid (i, x) \in$ E $\wedge$ (i,x) $\in$ Next-Edge,) $\cup$ $\{(i$, end) $\mid (i$, end) $\in$ Next-Edge,$\}$. After arc consistency, if $(i, j) \in$ Local-Next-Support$(i$, a) and $j \neq$ end, a must be compatible with at least one of j's labels.
Prev-Support$[(i, j), a]$	A set initialized to $\{$(i, $x) \mid (i, x) \in$ E A (x,j) $\in$ Prev-Edge,) $\cup$ $\{(i, j) \mid (i, j) \in$ Prev-Edge$_j\}$ $\cup$ $\{$(i, start) $\mid$ (start, j) $\in$ Prev-Edge$_j\}$. After arc consistency, if (i, k) $\in$ Prev-Support$[(i, j), a]$ and k #start, then $a \in L$, is compatible with at least one of j's and one of k's labels.
Next-Support$[(i, j), a]$	A set initialized to $\{$(i, $x) \mid (i, x) \in$ E $\wedge$ $(j$, x) $\in$ Next-Edge,) $\cup$ $\{(i, j) \mid (j, i) \in$ Next-Edge,) $\cup$ $\{(i$, end) $\mid (j$, end) $\in$ Next-Edge,). After arc consistency, if (i, k) $\in$ Next-Support$[(i, j)$, a] and k $\neq$ end, then a $\in L$, is compatible with at least one of j's and one of k's labels.

Figure 1: Data structures and notation used by the MUSE CSP arc consistency algorithms.

```
procedure AC-6-initialize (){
    Last := 0;
    for $i \in N$ do
        for $a \in L_i$ do {
            $S[i,a] := \emptyset$;
            $M[i,a] := 0$; }
    for $(i,j) \in E$ do
        for $a \in L_i$ do {
            b:= first($L_j$); nextsupport(i, j, a, b, emptysupport);
            if emptysupport then {
                remove($a, L_i$);
                $M[i,a] := 1$;
                List := List $\cup \{(i, a)\}$;}
            else $S[j,b] := S[j,b] \cup \{(i,a)\}$;} }
```

Figure 2: The procedure to initialize the AC-6 data structures.

```
procedure AC-6 (){
    AC-6-initialize();
    while List $\neq 0$ do {
            pop (j,b) from List;
            for $(i,a) \in S[j,b]$ do {
                $S[j,b] := S[j,b] - \{(i,a)\}$;
                if $M[i,a] = 0$ then {
                    c:=b; nextsupport(i, j, a, c, emptysupport);
                    if emptysupport then {
                        remove($a, L_i$);
                        $M[i,a] := 1$
                        List := List $\cup \{(i,a)\}$;}
                    else $S[j,c] := S[j,c] \cup \{(i,a)\}$; } } } }
```

Figure 3: The procedure AC-6.

constant-time operations:

1. **first**(L_i) returns the smallest $a \in L_i$ if $L_i \neq \emptyset$, else returns 0.
2. **last**(L_i) returns the greatest $a \in L_i$ if $L_i \neq \emptyset$, else returns 0.
3. If $a \in (L_i - \textbf{last}(L_i))$, **next**$(a, L_i)$ returns the smallest $b \in L_i$ such that $a < b$.
4. **remove**(a, L_i) removes a from L_i.

If a and b are both elements of L_i, then a $<$ b means that a comes before b in L_i. Additionally, $0 <$ a is true for all values in L_i. AC-6 uses the sets $S[i, a] = \{(j,b) \mid (i,a)$ is the smallest value in L_i supporting $(j,b)\}$ to determine whether the label a is supported by any label associated with at least one other variable. $M[i, a]$ is used to indicate whether or not the label a is admissible and has been eliminated from L_i. Figure 2 shows the code for initializing the data structures, and Figure 3 contains the algorithm for eliminating inconsistent labels from the domains. Figure 4 contains the procedure nextsupport, which is used by both of these routines.

The major reason for AC-4's poor average-case running time is that during the initialization phase, AC-4 exhaustively checks every single constraint to build the support sets $S[i, a]$. Instead of calculating all the support for all assignments (i,a) during initialization (as is done by AC-4),

```
procedure nextsupport (in i, j, a; in out b; out emptysupport){
1.  if b $\leq$ last($L_j$) then {
2.     emptysupport:= false;
3.     while M[$j$,b] do b :=next($b$, $L_j$);
4.     while not $R2(i, a, j, b)$ and not emptysupport do
5.           if b < last($L_j$) then b :=next($b$, $L_j$);
6.           else emptysupport:= true; }
7.  else emptysupport:= true; }
```

Figure 4: The procedure nextsupport is used by AC-6, and MUSE AC-2, and MUSE AC-3.

AC-6 initially only looks for the first label $b \in L_j$ for each node j that supports (i,a). If b is ever eliminated from L_j, then AC-6 looks for the *next* label c $\in L_j$ that supports (i,a), and if no such label can be found, it eliminates a from L_i. AC-6 uses the procedure nextsupport to find the smallest value in L_j that is greater than or equal to b and supports (i, a). In this way, AC-6 avoids scanning through all the constraints. It only checks as many of the constraints as is necessary to enforce arc consistency, resulting in a much better average-case running time than AC-4.

3 MUSE AC-2 and MUSE AC-3

Because MUSE AC-1 was built on top of AC-4, it inherits AC-4's poor average-case running time. However, by constructing a MUSE arc consistency algorithm that adapts the mechanisms of AC-6, we can improve the average-case running time. We have developed two MUSE arc consistency algorithms. The first, MUSE AC-2, builds and maintains the same data structures as MUSE AC-1 described in Figure 1 except that the Counter array is eliminated and the S sets are maintained as in AC-6 The DAG support sets (i.e., Prev-Support, Next-Support, Local-Prev-Support, and Local-Next-Support) are initialized and updated as in MUSE AC-1. In particular, all possible tuples are initially stored in the DAG support sets (as described in Figure 1), and those sets are updated depending on tuples stored on List. Figure 5 shows the algorithm for initializing the data structures and Figure 6 contains the algorithm for eliminating inconsistent labels from the domains. The procedures to update the DAG support sets for MUSE AC-2 are the same as for MUSE AC-1 (see [6]) and appear in Figure 7.

MUSE AC-3 manages the S support sets in the same way as in MUSE AC-2; however, it uses a new method for initializing the DAG support sets and updating them in Figure 5. Rather than initializing Local-Prev-Support(i, a) to a set of all (i, j) pairs such that there is a directed edge from j to i in the DAG and Local-Next-Support(i, a) to a set of all (i, j) pairs such that there is a directed edge from i to j, it sets Local-Prev-Support(i, a) to (i,j) such that j is the smallest node that precedes i in the DAG, and Local-Next-Support(i, a) to (i,j) such that j is the smallest node that follows i in the DAG. Note that start and end are defined to be the smallest and largest possible nodes, respectively. The Prev-Support[(i, j),a] set is initialized to (i,k) such that k defaults first to

```
procedure MUSE-AC-initialize {
1.  List := $\emptyset$;
2.  $E := \{(i,j) | \exists\sigma \in \Sigma : i,j \in \sigma \wedge i \neq j \wedge i,j \in N\}$;
3.  for $i \in N$ do
4.      for $a \in L_i$ do {
5.          for $j \in N$ such that $(i,j) \in E$ do {
6.              S[$(i,j)$,a] := $\emptyset$;
7.              M[$(i,j),a$] := 0;
8.              initialize_support(i, j, a); }
9.          initialize_local_support(i, a); }
10. for $i \in N$ do
11.     **for** $a \in L_i$ **do** {
12.         for $j \in N$ such that $(i,j) \in E$ do {
13.             $b$ := **first**($L_j$); nextsupport(i, j, a, b, emptysupport);
14.             if emptysupport then {
15.                 List := List $\cup$ {[$(i$, j),a]);
16.                 M[$(i$, j),a] := 1; }
17.             **else** S[$(j,i),b$] := S[$(j,i),b$] $\cup$ {$(i$, a)); } }
```

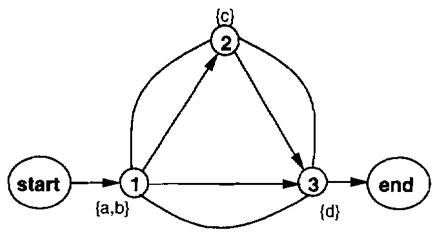
Figure 5: Initialization of the data structures for MUSE AC-2 and MUSE AC-3. The two algorithms differ only in the version of initialize-support and initialize-localsupport used.

```
procedure MUSEAC (){
1.  MUSE-AC-initialize();
2.  while List $\neq \emptyset$ do {
3.      pop [$(j$, i),$b$] from List;
4.      for $(i,a) \in$ S[$(j$, i),$b$] do {
5.          S[$(j,i),b$] := S[$(j,i),b$] - {$(i,a)$};
6.          if M[$(i,j),a$] = 0 then {
7.              c:=b; nextsupport(i, j, a, c, emptysupport);
8.              if emptysupport then {
9.                  List := List $\cup$ {[$(i$, j),a]);
10.                 M[$(i,j),a$]:=1; }
11.             **else** S[$(j,i),c$]:=S[$(j,i),c$] $\cup$ {$(i$, a)); } }
12.     Update_Support_Sets([$(j$, i),$b$]);
13.     Update_Local_Support_Sets([$(j,i)$, b]); } }
```

Figure 6: Eliminating inconsistent labels from the domains in MUSE AC-2 and AC-3. Update_Support_Sets and Update-Local-Supportsets are different for MUSE AC-2 and MUSE AC-3.

```
procedure Update-Support-Sets ([(j,i), b]) {
1.  for $(j,x) \in$ Prev-Support[$(j,i),b$] A $x \neq i \wedge x \neq start$ do {
2.      remove($(j,x)$, Prev-Support[$(j,i),b$]);
3.      Next-Support[$(j,x),b$]:=Next-Support[$(j,x),b$] $- \{(j,i)\}$;
4.      if Next-Support[$(j,x),b$] $= \emptyset$ A M[$(j,x),b$] $= 0$ then {
5.          $List := List \cup \{[(j,x),b]\}$;
6.          M[$(j,x),b$] := 1; } }
7.  for $(j,x) \in$ Next-Support[$(j,i),b$] A $x \neq i$ A $x \neq end$ do {
8.      remove($(j,x)$, Next-Support[$(j,i),b$]);
9.      Prev-Support[$(j,x),b$]:=Prev-Support[$(j,x),b$] $- \{(j,i)\}$;
10.     if Prev-Support[$(j,x),b$] $= 0 \wedge$ M[$(j,x),b$] $= 0$ then {
11.         $List := List \cup \{[(j,x),b]\}$;
12.         M[$(j,x),b$] := 1; } } }

procedure Update_Local_Support_Sets ([(ji),b]){
1.  if $(i,j) \in$ Prev-Edge$_j$ then
2.      Local-Prev-Support$(j,b)$:=Local-Prev-Support$(j,b) - \{(j,i)\}$;
3.  if Local-Prev-Support$(j,b) = 0$ then {
4.      $L_j := L_j - \{b\}$;
5.      for $(j,x) \in$ Local-Next-Support$(j,b)$ A $x \neq i \wedge x \neq end$ do {
6.          remove($(j,x)$, Local-Next-Support$(j,b)$);
7.          if M[$(j,x),b$] $= 0$ then {
8.              $List := List \cup \{[(j,x),b]\}$;
9.              M[$(j,x),b$] := 1; } } }
10. if $(j,i) \in$ Next-Edgej then
11.     Local-Next-Support$(j,b)$:=Local-Next-Support$(j,b) - \{(j,i)\}$;
12. if Local-Next-Support$(j,b) = 0$ then {
13.     $L_j := L_j - \{b\}$;
14.     for $(j,x)$ E Local-Prev-Support$(j,b)$ A $x$ # $i$ A $x \neq start$ do {
15.         remove($(j,x)$, Local-Prev-Support$(j,b)$);
16.         if M[$(j,x),b$] $= 0$ then {
17.             $List := List \cup \{[(j,x),b]\}$;
18.             M[$(j,x),b$] := 1; } } } }
```

Figure 7: The procedures for updating the DAG support sets in MUSE AC-2.



a. The network.

Prev-Support[(1,2),a] = {(1,2)}	Next-Support[(1,2),a] = {(1,3)}
Prev-Support[(1,3),a] = {(1,2),(1,3)}	Next-Support[(1,3),a] = {(1,end)}
Prev-Support[(1,2),b] = {(1,2)}	Next-Support[(1,2),*b*] = ((1.3))
Prev-Support[(1,3),*b*] = {(1,2),(1,3)}	Next-Support[(1,3),*b*] = {(1,end)}
Prev-Support[(2,1),*c*] = {(2,start)}	Next-Support[(2,1),*c*] = {(2,1),(2,3)}
Prev-Support[(2,3),c] = {(2,3),(2,1)}	Next-Support[(2,3),c] = {(2,end))
Prev-Support[(3,1),*d*] = {(3,start)}	Next-Support[(3,1),*d*] = {(3,1),(3,2)}
Prev-Support[(3,2),*d*] = {(3,1)}	Next-Support[(3,2),*d*] = {(3,2))
Local-Prev-Support(1,*a*) = {(1,start)}	Local-Next-Support(1,a) = {(1,2),(1,3)}
Local-Prev-Support(1,*b*) = {(1,start)}	Local-Next-Support(1,b) = {(1,2),(1,3))
Local-Prev-Support(2,*c*) = {(2,1)}	Local-Next-Support(2,*c*) = {(2,3)}
Local-Prev-Support(3,*d*) = {(3,1),(3,2)}	Local-Next-Support(3,*d*) = {(3,end)}

b. DAG support set initialization in MUSE AC-2.

Prev-Support[(1,2),*a*] = {(1,2)}	Next-Support[(1,2),a] = {(1,3)}
Prev-Support[(1,3),*a*] = {(1,3)}	Next-Support[(1,3),a] = {(1,end)}
Prev-Support[(1,2),*b*] = {(1,2))	Next-Support[(1,2),*b*] = {(1,3)}
Prev-Support[(1,3),b] = {(1,3))	Next-Support[(1,3),*b*] = {(1,end)}
Prev-Support[(2,1),c] = ((2,start))	Next-Support[(2,1),c] = {(2,1)}
Prev-Support[(2,3),*c*] = ((2.3))	Next-Support[(2,3),c] = ((2,end))
Prev-Support[(3,1),d] = ((3,start))	Next-Support[(3,1),*d*] = {(3,1)}
Prev-Support[(3,2),*d*] = ((3.1))	Next-Support[(3,2),*d*] = {(3,2)}
Local-Prev-Support(1,a) = {(1,start))	Local-Next-Support(1,*a*) = {(1,2)}
Local-Prev-Support(1,*b*) = {(1,start))	Local-Next-Support(1,*b*) = {(1,2)}
Local-Prev-Support(2,c) = {(2,1)}	Local-Next-Support(2,*c*) = {(2,3)}
Local-Prev-Support(3,*d*) = {(3,1)}	Local-Next-Support(3,*d*) = ((3,end))

c. DAG support set initialization for MUSE AC-3.

Figure 8: A simple example of the initialization of the DAG support sets by MUSE AC-2 and MUSE AC-3.

start and then to j (if either is a possible entry) or is the smallest node that precedes *j* in the DAG. Similarly, the Next-Support$[(i, j), a]$ set is initialized to (i,k) such that k defaults first to end and then to j (if either is a possible entry) or is the smallest node that follows *j* in the DAG. Setting k in this way makes dealing with the boundary cases easier and minimizes the need to update the entry until a is no longer supported by all segments (in the case of start and end) or the tuple $[(i, j), a]$ is placed on List. Figure 8 illustrates the initialization of the DAG support sets by MUSE AC-2 (same as for MUSE AC-1) and MUSE AC-3.

Assuming that each node in the network is numbered with a unique integer, it is a simple matter to determine a total order on the nodes that precede or follow a certain node. The following constant time functions have been defined:

1. **first_prev**(i) returns the smallest node n such that $(n,i) \in$ Prev-Edge;, else returns 0.
2. **last_prev**(i) returns the greatest node n such that $(n,i) \in$ Prev-Edge$_i$, else returns 0.
3. If k=**last_prev**(i) and $(n,i) \in$ (Prev-Edge$_i$ - (k, i)), **next_prev**(i, j) returns the smallest n such that

(n,i) ∈ Prev-Edge$_i$ and $j<n$. If k=**last_prev**(i), **next_prev**(i, k) returns 0.
4. **first_next**(i) returns the smallest node n such that (i,n) ∈ Next-Edge$_i$, else returns 0.
5. **last_next**(i) returns the greatest node n such that (i,n) ∈ Next-Edge$_i$, else returns 0.
6. If k=**last_next**(i) and (i,n) ∈ (Next-Edge$_i$ - (i,k)), **next_next**(i, j) returns the smallest n such that (i,n) ∈ Next-Edge$_i$ and $j<n$. If k=**last_next**(i), **next_next**(i,k) returns 0.

Whenever MUSE AC-3 pops [(i,j),a] from List, this information must be propagated to the DAG support sets using Update-Support-Sets and Update_Local_Support_Sets in Figures 9 and 10, respectively. If, during these procedures, the single element in one of the DAG support sets is deleted, a new member for the set must be located if one is available; otherwise, the set is considered to be empty (with consequences like those in MUSE AC-1 and AC-2). Note that due to the initialization preference of setting k in the (i, k) tuple for Prev-Support[(i, j), a] to the value of j or start, it becomes necessary to invoke Get-Non-Default-Prev when updating these sets to reset the (i, k) tuple so that k is the first prev node that is not equal to j or start. Similarly, due to the initialization preference of setting k in the (i, k) tuple for Next-Support[(i, j),a] to the value of j or end, it becomes necessary to invoke Get-Non-Default-Next when updating these sets to reset the (i, k) tuple so that k is the first next node that is not equal to j or end. The procedure next_local_prev_support locates the next largest j preceding i (shown in Figure 11) to update Local-Prev-Support(i, a), next_local_next_support locates the next largest j following i (shown in Figure 11) to update Local-Next-Support(< a), next_prev_support locates the next largest k preceding j (shown in Figure 12) to update Prev-Support[(i, j), a], and next_next_support locates the next largest k following j (shown in Figure 12) to update Next-Support[(i, j), a]. Each of these procedures sets the associated empty flag to true if there is no such next element (just as in nextsupport).

The worst-case running time for MUSE AC-2 and MUSE AC-3 is the same as for MUSE AC-1, O($n^2l^2 + n^3l$), where n is the number of nodes in the MUSE CSP and l is the number of labels. The proof of correctness of the algorithms is comparable to that for MUSE AC-I. [5, 6], and so we will not give the proof here.

4 Experiments, Results, and Conclusions

In order to compare the performance of MUSE AC-1, MUSE AC-2, and MUSE AC-3, we have conducted experiments in which we randomly generate MUSE CSP instances with three different topologies. The tree topology (Figure 13(a)) is characterized by two parameters: the branching factor (how many nodes follow each non-leaf node in the tree) and the path length (how many nodes there are in a path from the root node to a leaf node). The random split topology (Figure 14(a)) is characterized by three parameters: the number of nodes in the initial chain, the probability that a node is split during an iteration, and the number of iterations. Finally, the *lattice* topology (Figure 15(a)) is characterized by its branching factor and path length.

In addition to the topology, a DAG in a MUSE CSP has three other defining parameters: the

```
procedure Update-Support-Sets $([(j,i),b])\{$
1.  if Prev-Support$[(j,i), b] \neq NIL$ then $\{$
2.      emptyprev := false;
3.      $(j,x)$ := Prev-Support$[(j,i),b]$;
4.      $x$ := Get-Non-Default-Prev$(i,j,x,emptyprev)$;
5.      while (not emptyprev) do $\{$
6.          if $(x \neq i$ && $x \neq start)$ then $\{$
7.              if $(j,i) \in$ Next-Support$[(j,x),b]$ then $\{$
8.                  remove($(j,i)$,Next-Support$[(j,x),b]$);
9.                  new-next := $i$; next_next_support($j$,$x$,$b$,new-next,emptynext);
10.                 if emptynext $\wedge$ M$[(j,x),b] = 0$ then $\{$
11.                     List := List $\cup \{[(j,x),b]\}$;
12.                     M$[(j,x),b]$ := 1; $\}$
13.                 else Next-Support$[(j,x),b]$:= (j,new-next); $\}$ $\}$
14.         next_prev_support($j$,$i$,$b$,$x$,emptyprev); $\}$ $\}$
15. if Next-Support$[(j,i),b] \neq NIL$ then $\{$
16.     emptynext := false;
17.     $(j,x)$ := Next-Support$[(j,i),b]$;
18.     $x$ := Get-Non-Default-Next$(i,j,x,emptynext)$;
19.     while (notemptynext) do $\{$
20.         if $(x \neq i$ && $x \neq end)$ then $\{$
21.             if $(j,i) \in$ Prev-Support$[(j,x),b]$ then $\{$
22.                 remove($(j,i)$,Prev-Support$[(j,x),b]$);
23.                 new-prev := $i$, next_prev_support($j$,$x$,$b$,new-prev,emptyprev);
24.                 if emptyprev $\wedge$ M$[(j,x),b] = 0$ then $\{$
25.                     List := List $\cup \{[(j,x),b]\}$;
26.                     M$[(j,x),b]$ := 1; $\}$
27.                 else Prev-Support$[(j,x),b]$ := (j,new-prev); $\}$ $\}$
28.         next_next_support($j$,$i$,$b$,$x$,emptynext); $\}$ $\}$
```

Figure 9: Updating DAG support sets in MUSE AC-3.

number of labels in each node, the probability of a constraint existing between two nodes, and the probability of $R2(i, a, j, b) = 1$ given that i and j are constrained. The number of labels in this experiment was set to sixteen per node. We randomly generated constraints between variables in the MUSE instances. The probability of a constraint between two nodes assumes that a constraint is allowed between them. For example, nodes that are on the same level in the tree topology are in different segments, and so constraints cannot occur between them. For this experiment the probability of a constraint between two variables was set to fifty percent for all topologies. The probability that $R2(i, a, j, b) = 1$ was varied from 0% to 100% in steps of 5% (The lower the probability that $R2(i, a, j, b) = 1$, the tighter the constraints). For each probability, 6 instances were generated.

In the first experiment, we ran all three versions of MUSE AC on the three graph topologies; we used trees with a branching factor of two and a path length of four, random splits with an initial length of four and a fifty percent chance of a split per iteration with four iterations, and lattices with a branching factor of three and a path length of four. The results of this experiment are displayed in Figures 13(b), 14(b), and 15(b). Each subfigure displays the running times for each of the three MUSE AC algorithms. Following Bessière [1], we measure the running times of each MUSE AC algorithm by counting the number of times an atomic operation is executed. Measuring

```
procedure Update_Local_Support_Sets ([($j$,i),b]) {
1.  if $(i,j) \in$ Prev-Edge, $\wedge$ $(j,i) \in$ Local-Prev-Support($j$, b) then {
2.    remove(($j$, a), Local-Prev-Support($j$, b));
3.    newlocalprev := $i$; next_local_prev_support($j$, b, newlocalprev, emptylocalprev);
4.    if emptylocalprev then {
5.      remove($b, L_j$);
6.      for (j,$x$) $\in$ Local-Next-Support($j$, b) do {
7.          emptylocalnext := false;
8.          remove(($j,x$),Local-Next-Support($j$, b));
9.          if $x$ = end then continue;
10.         while not emptylocalnext do {
11.                if M[($j$,x),b] = 0 then {
12.                   List := List $\cup$ {[(j, x),$b$]};
13.                   M[($j,x$),$b$] := 1; }
14.                next_local_next_support($j$, b, x, emptylocalnext); } } }
15.   else Local-Prev-Support($j$, $b$):=($j$, newlocalprev); }
16. if ($j$, a) $\in$ Next-Edge, $\wedge$ $(j,i) \in$ Local-Next-Support($j$, b) then {
17.   remove(($j, i$), Local-Next-Support($j$, b));
18.   newlocalnext := $i$; next_local_next_support($j$, b, newlocalnext, emptylocalnext);
19.   if emptylocalnext then {
20.     remove($b$, L,);
21.     for ($j$,x) $\in$ Local-Prev-Support($j$, b) do {
22.         emptylocalprev := false;
23.         remove(($j,x$),Local-Prev-Support($j$, b));
24.         if x = start then continue;
25.         while not emptylocalprev do {
26.                if M[($j$,x), b] = 0 then {
27.                   List := Last $\cup$ {[(j,x),b]);
28.                   M[($j,x$),$b$] := 1; }
29.                next_local_prev_support($j$, b, $x$, emptylocalprev); } } }
30.   else Local-Next-Support($j$, $b$):= ($j$, newlocalnext); } }
```

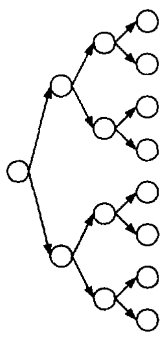
Figure 10: Updating **DAG** local support sets in MUSE **AC-3.**

```
procedure next_local_prev_support(in i, a; in out prev; out emptylocalprev) {
1.  if prev $\leq$ last_prev($i$) then {
2.    prev :=next_prev($i$,prev);
3.    while (prev && M[($i$,prev),$a$]) do
4.          prev :=next_prev($i$,prev);
5.    if prev then emptylocalprev:= false;
6.    else emptylocalprev:= true; }
7.  else emptylocalprev:= true; }

procedure next_local_next_support(in i, a; in out next; out emptylocalnext) {
1.  if next $\leq$ last_next($i$) then {
2.    next :=next_next($i$,next);
3.    while (next && M[($i$,next),$a$]) do
4.          next :=next_next($i$,next);
5.    if next then emptylocalnext:= false;
6.    else emptylocalnext:= true; }
7.  else emptylocalnext:= true; }
```

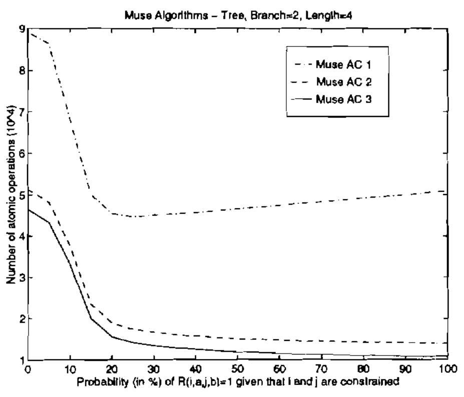
Figure 11: The procedures next_local_prev_support and next_local_next_support are used by MUSE **AC-3.**

```
procedure next_prev_support(in i, j, a; in out prev; out emptyprev) {
1.  if (prev = $j$ || prev $\leq$ last_prev($j$)) then {
2.     if (prev = j) then prev :=next_prev($j$,$i$);
3.     else prev :=next_prev($j$,prev);
4.     while (prev && (M[($i$,prev),$a$] || (prev = i)) ||
                  ((i,prev) $\notin$ E && prev $\neq$ start)) do(
5.           if (prev = j) then prev :=next_prev($j$,$i$);
6.           else prev :=next_prev($j$,prev); }
7.     if prev then emptyprev:= false;
8.     else emptyprev:= true; }
9.  else emptyprev:= true; }

procedure next_next_support(in i, j, a; in outv next; out emptynext) {
1.  if (next = j || next $\leq$ last_next($j$)) then {
2.     if (next = j) then next :=next_next($j$,$i$);
3.     else next :=next_next($j$,next);
4.     while (next && (M[($i$,next),$a$] || (next = $i$)) ||
                  ((i,next) $\notin$ E && next $\neq$ end)) do{
5.           if (next = j) then next :=next_next($j$,$i$);
6.           else next :=next_next($j$,next); }
7.     if next then emptynext:= false;
8.     else emptynext := true; }
9.  else emptynext:= true; }
```

Figure 12: The procedures next_prev_support and next_next_support are used by MUSE AC-3.

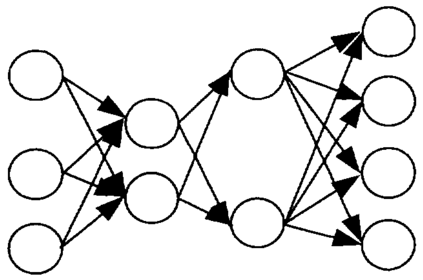


(a) Tree topology

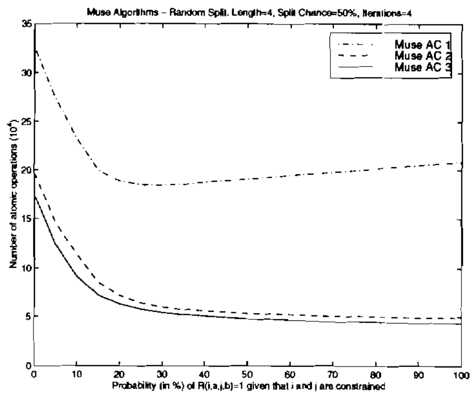


(b) Tree results

Figure 13: 'The number of operations performed by MUSE AC-1, MUSE AC-2, and MUSE AC-3 for a tree of depth 4, each node having 16 labels. The probability of a constraint between two nodes is 50 percent.

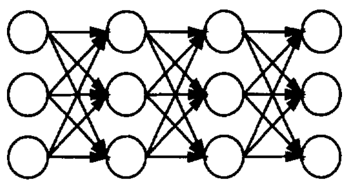


(a) Random split topology

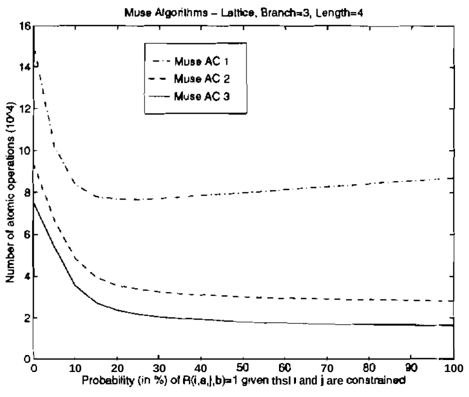


(b) Random split results

Figure 14: The number of operations performed by MUSE AC-1, MUSE AC-2, and MUSE AC-3 for a randoin-split DAG with initial length of 4, a *50* percent chance of splitting, run for 4 iterations, each node having 16 labels. The probability of a constraint between two nodes is *50* percent.



(a) Lattice topology



(b) Lattice results

Figure 15: 'The number of operations performed by MUSE AC-1, MUSE AC-2, and MUSE AC-3 for a lattice with branching factor **3** and path length 4, each node having 16 labels. The probability of a constraint between two nodes is *50* percent.

the running time this way has the advantage of being implementation independent while remaining proportional to the actual execution time. We considered the following operations to be atomic: consulting the value of a constraint; eliminating a label from a domain; setting or consulting an entry in M; adding, removing, or consulting an entry in a DAG support set or S set; adding or removing a tuple from List; and, initializing the entries for the data structures.

As can be seen in Figures 13(b), 14(b), and 15(b), adapting ideas from Bessière's AC-6 in order to formulate the MUSE AC-2 and MUSE AC-3 algorithms greatly reduces the amount of computation needed for all three topologies when compared to the MUSE AC-1 algorithm (based on AC-4). Also, the modifications that were made to MUSE AC-2 to create MUSE AC-3 (as described in the section on MUSE AC-2 and MUSE AC-3) produce a modest gain in computational speed. Also evident in the figures is that all topologies start off with a high number of atomic operations that sharply decreases and then levels off as the probability of $R2(i, a, j, b) = 1$ increases, given that i and j are constrained. The decrease and leveling off are due to the network stabilizing because of the increase in the admissibility of the labels from loose constraints. The asymptotic level reached for moderate-to-loose constraints largely represents the cost of initializing the MUSE CSP.

As a second experiment, we investigated the influence of a lattice's shape on the number of operations required by MUSE AC-3. We generated lattices with path lengths three, four, or five and branching factors of three, four, or five with a limit of sixteen nodes in a lattice. As can be seen in Figure 16, the number of nodes in a lattice affects the number of atomic operations performed, as one might expect. The lattice with nine nodes used fewer operations than that of both of the twelve node lattices, which used fewer operations than the fifteen and sixteen node lattices. Also, the shape of the lattice affects the number of operations needed. Lattices with a shorter path length but a greater branching factor require fewer operations than lattices with the same number of nodes but with a longer path (see the twelve node and fifteen node lattices where the 4x3 and 5x3 (branching factor x path length) lattices require fewer operations than the 3x4 and 3x5 lattices, respectively). This is because there are a greater number of alternative segments in the lattices with higher branching factors; these alternative segments increase the chances that a label will be MUSE arc consistent.

The MUSE AC-1, MUSE AC-2, and MUSE AC-3 algorithms have been incorporated into our CDG parser to perform arc consistency prior to extraction of legal parses. This parser uses methods developed by Harper and Helzerman [4] to parse a sentence containing words with multiple lexical categories and multiple feature values. This method applies constraints in multiple stages such that, at the initial stage, the MUSE network representing the sentence is fairly small. During each stage the MUSE network representing the sentence is expanded so that new types of constraints can be applied to it (e.g., number agreement constraints, and subcategorization constraints). This method of parsing keeps the size of the MUSE network relatively small. For this experiment, we randomly chose 100 sentences to parse from the Resource Management Corpus [12] in order to compare the

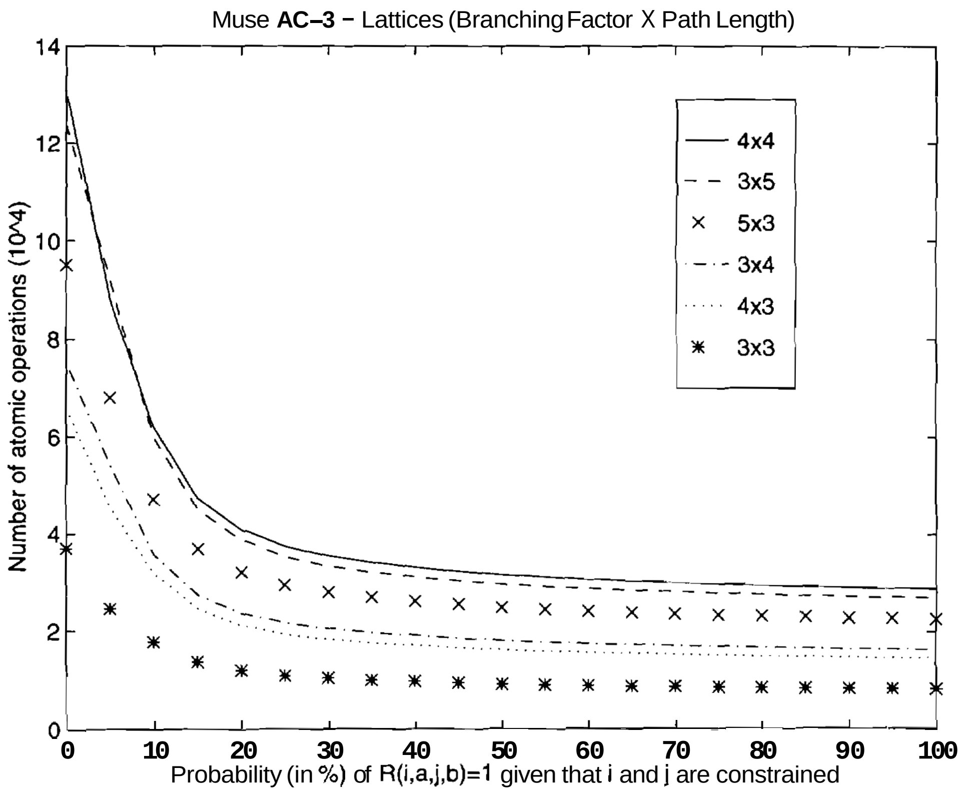


Figure 16: The number of operations performed by MUSE **AC-3** for lattices of various sizes specified by branching factor $\times$ path length. Each node has 16 labels, and the probability of a constraint between two nodes is 50 percent.

Comparing MUSE	Mean	Median	Std. Deviation
AC-1 to AC-2	2.07	1.68	2.40
AC-2 to AC-3	29.36	30.25	6.28
AC-1 to AC-3	30.83	31.77	6.28

Figure 17: Percent improvement statistics for MUSE AC-1, MUSE AC-2, and MUSE AC-3

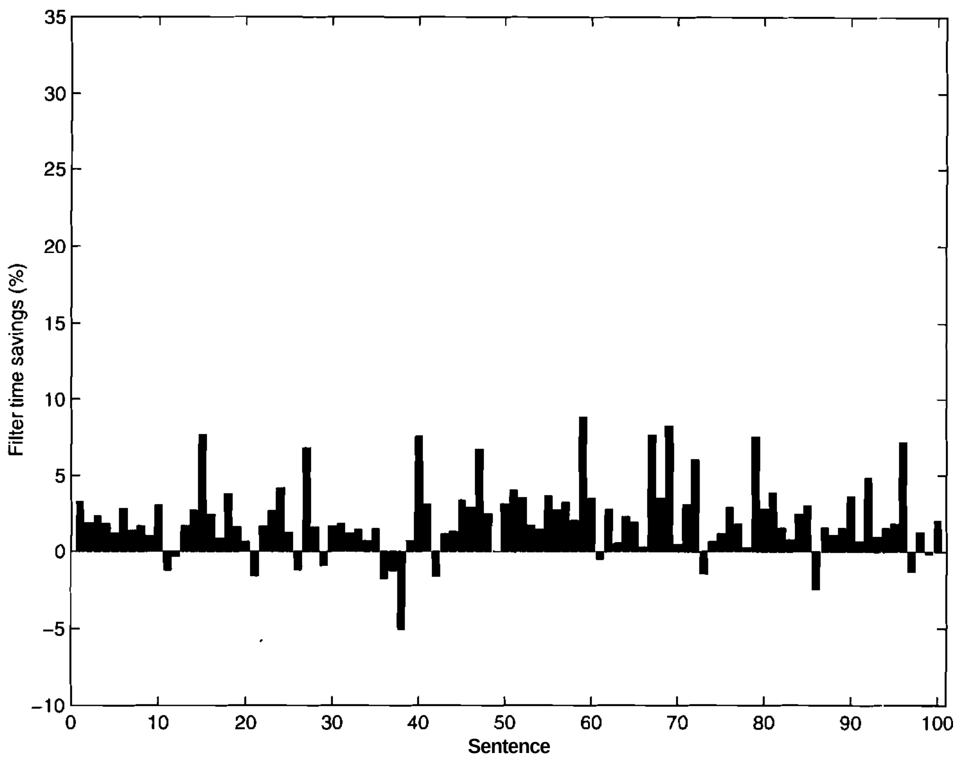


Figure 18: Average percent difference in filtering times between MUSE AC-1 and MUSE AC-2 for each sentence.

running times of MUSE AC-1, MUSE AC-2, and MUSE AC-3. We calculated the amount of time used by each algorithm during each sentence parse (averaged over three test runs) and examined the percent decrease in running time between each arc consistency algorithm. These results are displayed in Figure 17. Note that the most dramatic reduction in running resulted from using MUSE AC-3 as opposed to MUSE AC-1 or MUSE AC-2. Clearly MUSE AC-3 was the outright winner for all sentences parsed, as can be seen in Figures 18 and 19. MUSE AC-2 only resulted in modest improvements over MUSE AC-1. This is likely due to the fact that our parsing algorithm applies the arc consistency algorithm to moderately small constraint networks over multiple stages of parsing.

We have shown that MUSE AC-2 performs fewer operations than MUSE AC-1 simply by using the method of initializing and updating the S sets as in [1]. MUSE AC-3 improves upon MUSE AC-2 by using our "lazy" method to initialize and update the DAG support sets. These runtime improvements have proven useful for applications using a MUSE CSP based parser, including spoken

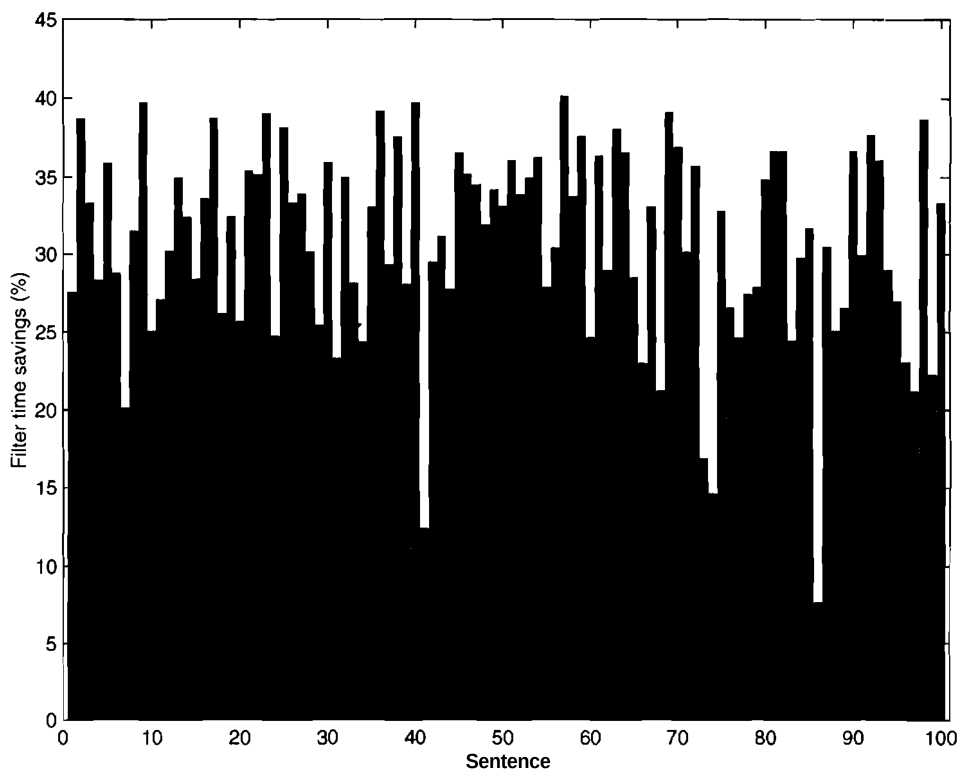


Figure 19: Average percent difference in filtering times between MUSE AC-1 and MUSE AC-3 for each sentence.

language understanding systems [3] and natural language front ends for multiple databases [4]. These algorithms should also be effective for other CSP problems that have problems comparable to lexical ambiguity, feature ambiguity, or ambiguity resulting from the inability to segment a signal into higher-level chunks in a single way. Two examples of comparable domains are visual understanding and handwriting analysis.

References

[1] Christian Bessikre. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994.

[2] C. Han and C. Lee. Comments on Mohr and Henderson's path consistency algorithm. *Artificial Intelligence*, 36:125–130, 1988.

[3] M. P. Harper and R. A. Helzerman. Extensions to constraint dependency parsing for spoken language processing. *Computer Speech and Language*, 9(3):187–234, 1995.

[4] M. P. Harper and R. A. Helzerman. Managing multiple knowledge sources in constraint-based parsing of spoken language. *Fundamenta Informaticae*, 23(2,3,4):303–353, 1995.

[5] Randall A. Helzerman and Mary P. Harper. An approach to multiply-segmented constraint satisfaction problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 350–355, 1994.

[6] Randall A. Helzerman and Mary P. Harper. MUSE CSP: An extension to the constraint satisfaction problem. *Journal of Artificial Intelligence Research*, 5:239–288, 1996.

[7] A. K. Mackworth and E. Freuder. The complexity of some polynomial network-consistency algorithms for constraint-satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.

[8] H. Maruyama. Constraint dependency grammar. Technical Report #RT0044, IBM, Tokyo, Japan, 1990.

[9] H. Maruyama. Constraint dependency grammar and its weak generative capacity. *Computer Software*, 1990.

[10] H. Maruyama. Structural disambiguation with constraint propagation. In *The Proceedings of the Annual Meeting of ACL*, pages 31–38, 1990.

[11] R. Mohr and T. C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[12] P. J. Price, W. Fischer, J. Bernstein, and D. Pallett. A database for continuous speech recognition in a 1000-word domain. In *Proceedings of the International Conference on Acoustics, Speech,, and Signal Processing*, pages 651–654, 1988.