Proceedings of the IASTED International Conference
on Artificial Intelligence and Soft Computing,
August 9-12, 1999, Honolulu, Hawaii USA

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
Purdue University, West Lafayette, IN 47907-1285
{harper,robot,helz,hockema}@ecn.purdue.edu

Abstract

MUSE CSP (*MU*ltiply *SE*gmented *C*onstraint *S*atisfaction *P*roblem) [1, 2] 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 Helzerman and Harper [2], 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 [3]. Recently, Bessière developed a new algorithm, AC-6 [4], which has the same worst-case running time as AC-4 but 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 Bessière's method to improve upon MUSE AC-1; and MUSE AC-3, which uses a fast evaluation method for managing the additional sets required by the MUSE approach. These new algorithms decrease the number of steps required to achieve arc consistency in artificial randomly generated and real MUSE CSP instances when compared to MUSE AC-1.
Keywords Constraint Satisfaction, MUSE Arc Consistency.

Introduction: MUSE AC-1

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 machine vision, belief maintenance, temporal reasoning, circuit design, diagnostic reasoning, and natural language processing.

Constraint Dependency Grammar (CDG) parsing, as introduced by Maruyama [5], 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, nor the simultaneous parsing of sentences contained in a word graph produced by a speech recognizer. Hence, Harper and Helzerman [6] adapted the parsing algorithm to support the simultaneous parsing of alternative sentences resulting from these types of ambiguity. From this work, the concept of a MUSE CSP (*MU*ltiply *SE*gmented *C*onstraint *S*atisfaction *P*roblem) [2] 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* variables *with* $|N| = n$,
$\Sigma \subseteq 2^N$ *is a set of* segments *with* $|\Sigma| = s$,
$L = \{a, b, \ldots\}$ *is the set of* labels, *with* $|L| = l$,
$L[i] = \{a |\ a \in L \wedge (i, a)$ *is admissible in at least one segment*$\}$,
$R1$ *is a unary constraint where* (i, a) *is admissible if* $R1(i, a)$,
$R2$ *is a binary constraint where* $(i, a) - (j, b)$ *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 (DAG), segments are defined as paths through the MUSE CSP from special purpose start to end nodes. Helzerman and Harper [2] also defined the concepts of MUSE node, arc, and path consistency and developed algorithms for MUSE arc and path consistency: MUSE AC-1 and MUSE PC-1. MUSE AC-1 was adapted from the CSP arc consistency (AC) algorithm AC-4 [3], which has a worst-case running time of $\Theta(el^2)$ (where e is the number of constraint arcs). Recently, Bessière developed a new algorithm, AC-6 [4], which has the same worst-case running time as AC-4 but is faster than AC-4 in practice. In this paper, we focus on providing faster MUSE AC algorithms by applying techniques similar to those developed by Bessière.

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

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

Definition 2 (MUSE AC) *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. Hence, there is more information that must be tracked for MUSE arc consistency.

MUSE AC is enforced by removing those labels in each $L[i]$ that violate the conditions of Definition 2. MUSE AC-1 [2] builds and maintains several data structures defined 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 $a \in L[i]$ has from the labels in $L[j]$ by counting the number that are compatible with a and storing the number in Counter$[(i,j),a]$. The algorithm also keeps track of the set of labels in $L[j]$ 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 $L[i]$ 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 AC-1, 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. To do this, the tuple $[(i,j),a]$ is placed on *List* (as in AC-4) and M$[(i,j),a]$ is set to 1. When the tuple is popped off *List*, the information is used to determine whether the label $a \in L[i]$ is illegal in other segments or is globally inadmissible, in which case it can be deleted.

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. For each variable i, variable j, $j \neq i$, and label $a \in L[i]$, MUSE AC-1 keeps track of two sets, Prev-Sup$[(i,j),a]$ and Next-Sup$[(i,j),a]$, which are formally defined in Figure 1. These sets keep track of those variables that precede or follow j and support at least one label in $L[j]$ and the label a in $L[i]$. In addition, two sets are maintained for each variable i and each label $a \in L[i]$, Local-Prev-Sup(i,a) and Local-Next-Sup(i,a), to keep track of the variables that precede or follow i that support the label $a \in L[i]$ (see definitions in Figure 1). If either local set becomes empty, a is no longer a part of any MUSE arc consistent instance and should be eliminated from $L[i]$ because it is globally inadmissible.

In MUSE AC-1, 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-Sup$[(i,k),a]$ (if j was a successor) or Prev-Sup$[(i,k),a]$ (if j was a predecessor) becomes empty after (i,j) is removed from the set. If either 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-Sup(i,a), and if j directly follows i, then (i,j) is removed from Local-Next-Sup(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 $a \in L[i]$.

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 in AC-6. We first review AC-6 and then develop two versions of MUSE AC that build on AC-6: MUSE AC-2, which directly applies Bessière's method to improve upon MUSE AC-1, and MUSE AC-3, which adds a fast evaluation method for keeping track of the Prev-Sup, Next-Sup, Local-Prev-Sup, and Local-Next-Sup sets.

AC-6

To improve the average-case performance while maintaining the same worst-case time complexity of AC-4, Bessière [4] developed AC-6 (see pseudocode in [4]), an algorithm conceptually similar to AC-4 but which avoids much of the work 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 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. **next**$(a, L[i])$ returns the smallest $b \in L[i]$ such that $a < b$ if $a \in (L[i] - \textbf{last}(L[i]))$.
4. **remove**$(a, L[i])$ removes a from $L[i]$.

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 admissibility of $a \in L[i]$ and $b \in L[j]$ given binary constraints.
Counter $[(i,j),a]$	The number of labels in $L[j]$ that are compatible with $a \in L[i]$ (MUSE AC-1 only).
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 been eliminated from) all segments containing i and j.
G	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[i]	The set of node pairs (i,j) for each directed edge $(i,j) \in G$ and (i,end) if there is no $x \in N$ such that $(i,x) \in G$.
Prev-Edge[i]	The set of node pairs (j,i) for each directed edge $(j,i) \in G$ and (start,i) if there is no $x \in N$ such that $(x,i) \in G$.
Local-Prev-Sup(i,a)	$\{(i,x) \mid (i,x) \in E \wedge (x,i) \in$ Prev-Edge$[i]\} \cup \{(i,\text{start}) \mid (\text{start},i) \in$ Prev-Edge$[i]\}$. After arc consistency, if $(i,j) \in$ Local-Prev-Sup(i,a) and $j \neq$ start, a must be compatible with at least one of j's labels.
Local-Next-Sup(i,a)	$\{(i,x) \mid (i,x) \in E \wedge (i,x) \in$ Next-Edge$[i]\} \cup \{(i,\text{end}) \mid (i,\text{end}) \in$ Next-Edge$[i]\}$. After arc consistency, if $(i,j) \in$ Local-Next-Sup(i,a) and $j \neq$ end, a must be compatible with at least one of j's labels.
Prev-Sup $[(i,j),a]$	$\{(i,x) \mid (i,x) \in E \wedge (x,j) \in$ Prev-Edge$[j]\} \cup \{(i,j) \mid (i,j) \in$ Prev-Edge$[j]\} \cup \{(i,\text{start}) \mid (\text{start},j) \in$ Prev-Edge$[j]\}$. After arc consistency, if $(i,k) \in$Prev-Sup$[(i,j),a]$ and $k \neq$ start, then $a \in L[i]$ is compatible with at least one of j's and k's labels.
Next-Sup $[(i,j),a]$	$\{(i,x) \mid (i,x) \in E \wedge (j,x) \in$ Next-Edge$[j]\} \cup \{(i,j) \mid (j,i) \in$ Next-Edge$[j]\} \cup \{(i,\text{end}) \mid (j,\text{end}) \in$ Next-Edge$[j]\}$. After arc consistency, if $(i,k) \in$Next-Sup$[(i,j),a]$ and $k \neq$ end, then $a \in L[i]$ is compatible with at least one of j's and k's labels.

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

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 $L[i]$.

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$[i,a]$ during initialization. In contrast, AC-6 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 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.

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, we have developed two MUSE AC algorithms that use mechanisms similar to those in AC-6 to improve the average-case running time. 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-Sup, Next-Sup, Local-Prev-Sup, and Local-Next-Sup) 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 2 shows the algorithm for initializing the data structures and for eliminating inconsistent labels from the domains. The routine to update the DAG support sets for MUSE AC-2 is the same as for MUSE AC-1 (see pseudocode in [2]).

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 (see Figure 3). Rather than initializing Local-Prev-Sup(i,a) (Local-Next-Sup(i,a)) to a set of all (i,j) pairs such that there is a directed edge from j to i (i to j) in the DAG, it sets Local-Prev-Sup(i,a) (Local-Next-Sup(i,a)) to (i,j) such that j is the smallest node that precedes (follows) i in the DAG. Note that start (end) is defined to be the smallest (largest) possible node. The Prev-Sup$[(i,j),a]$ (Next-Sup$[(i,j),a]$) set is initialized to (i,k) such that k defaults, whenever possible, to j or start (end) or is set to the smallest node that precedes (follows) j in the

```
procedure MUSE-AC-INITIALIZE {
 1. List := ∅;
 2. E := {(i,j)|∃σ ∈ Σ : i,j ∈ σ ∧ i ≠ j ∧ i,j ∈ N};
 3. for i ∈ N do
 4.     for a ∈ L[i] do {
 5.         for j ∈ N such that (i,j) ∈ E do {
 6.             S[(i,j),a] := ∅; M[(i,j),a] := 0;
 7.             INITIALIZE_SUP(i,j,a); }
 8.         INITIALIZE_LOCAL_SUP(i,a); }
 9. for i ∈ N do
10.     for a ∈ L[i] do {
11.         for j ∈ N such that (i,j) ∈ E do {
12.             b := first(L[j]);
13.             NEXTSUPPORT(i,j,a,b,emptysup);
14.             if emptysup then {
15.                List := List ∪ {[(i,j),a]};
16.                M[(i,j),a] := 1; }
17.             else S[(j,i),b] := S[(j,i),b] ∪ {(i,a)}; }}

procedureMUSE-AC (){
 1. MUSE-AC-INITIALIZE();
 2. while List ≠ ∅ do {
 3.     pop [(j,i),b] from List;
 4.     for (i,a) ∈ 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;
 8.            NEXTSUPPORT(i,j,a,c,emptysup);
 9.            if emptysup then {
10.               List := List ∪ {[(i,j),a]};
11.               M[(i,j),a]:=1; }
12.            else S[(j,i),c]:=S[(j,i),c] ∪ {(i,a)}; }}
13.     UPDATE_SUPPORT_SETS([(j,i),b]);
```

Figure 2: The routines to initialize the data structures and eliminate inconsistent labels from domains for MUSE AC-2 and MUSE AC-3. MUSE AC-2 and MUSE AC-3 differ only in the version of INITIALIZE_SUP, INITIALIZE_LOCAL_SUP, and UPDATE_SUPPORT_SETS used.

DAG. Setting k in this way makes dealing with the boundary cases easier and minimizes the need to update the entry until a is unsupported by all segments (in the case of start and end) or the tuple $[(i,j),a]$ is placed on $List$.

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 necessary constant time functions are defined below:

1. **first_prev**(i) returns the smallest node n such that $(n,i) \in$ Prev-Edge[i], 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 $(n,i) \in$ 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 $(i,n) \in$ Next-Edge[i], else returns 0.
5. **last_next**(i) returns the greatest node n such that $(i,n) \in$ Next-Edge[i], else returns 0.
6. If $k =$ **last_next**(i) and $(i,n) \in$ (Next-Edge[i] - (i,k)), **next_next**(i,j) returns the smallest n such that $(i,n) \in$ 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 in Figure 3. If, during this procedure, 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 because of the use of the initialization defaults for Prev-Sup$[(i,j),a]$ (Next-Sup$[(i,j),a]$), it is necessary, when updating these sets, to reset the (i,k) tuple so that k is the first prev (next) node that is not equal to j or start (end). The procedure next_local_prev_sup locates the next largest j preceding i to update Local-Prev-Sup(i,a); next_local_next_sup locates the next largest j following i to update Local-Next-Sup(i,a); next_prev_sup locates the next largest k preceding j to update Prev-Sup$[(i,j),a]$; and next_next_sup locates the next largest k following j to update Next-Sup$[(i,j),a]$. Each of these routines 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-1 [1, 2], and therefore is not given here.

Experiments and Results

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: *tree*, *random split*, and *lattice*. The *tree* topology 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 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. The *lattice* topology, is characterized by its *branching factor* and its *path length*. In addition to the topology, a MUSE CSP has three other defining parameters: the 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.

In Experiment 1, we ran all three versions of

```
procedure UPDATE_SUPPORT_SETS ([(j,i),b]) {
 1. if Prev-Sup[(j,i),b] ≠ NIL then {
 2.    emptyprev := false;
 3.    (j,x) := Prev-Sup[(j,i),b];
 4.    x := Get-Non-Default-Prev(i, j, x, emptyprev);
 5.    while (not emptyprev) do {
 6.       if (x ≠ i && x ≠ start) then {
 7.          if (j,i) ∈ Next-Sup[(j,x),b] then {
 8.             remove((j,i),Next-Sup[(j,x),b]);
 9.             new-next := i;
10.             next_next_sup(j,x,b,new-next,emptynext);
11.             if emptynext ∧ M[(j,x),b] = 0 then {
12.                List := List ∪ {[(j,x),b]};
13.                M[(j,x),b] := 1; }
14.             else Next-Sup[(j,x),b]:= (j,new-next); }}
15.       next_prev_sup(j,i,b,x,emptyprev); }}
16. if Next-Sup[(j,i),b] ≠ NIL then {
17.    emptynext := false;
18.    (j,x) := Next-Sup[(j,i),b];
19.    x := Get-Non-Default-Next(i, j, x, emptynext);
20.    while (not emptynext) do {
21.       if (x ≠ i && x ≠ end) then {
22.          if (j,i) ∈ Prev-Sup[(j,x),b] then {
23.             remove((j,i),Prev-Sup[(j,x),b]);
24.             new-prev := i,
25.             next_prev_sup(j,x,b,new-prev,emptyprev);
26.             if emptyprev ∧ M[(j,x),b] = 0 then {
27.                List := List ∪ {[(j,x),b]};
28.                M[(j,x),b] := 1; }
29.             else Prev-Sup[(j,x),b] := (j,new-prev); }}
30.       next_next_sup(j,i,b,x,emptynext); }}
31. if (i,j) ∈ Prev-Edge[j] ∧
       (j,i) ∈ Local-Prev-Sup(j,b) then {
32.    remove((j,i),Local-Prev-Sup(j,b));
33.    newlocprev := i;
34.    next_local_prev_sup(j,b,newlocprev,emptylocprev);
35.    if emptylocprev then {
36.       remove(b, L[j]);
37.       for (j,x) ∈ Local-Next-Sup(j,b) do {
38.          emptylocnext := false;
39.          remove((j,x),Local-Next-Sup(j,b));
40.          if x = end then continue;
41.          while not emptylocnext do {
42.             if M[(j,x),b] = 0 then {
43.                List := List ∪ {[(j,x),b]};
44.                M[(j,x),b] := 1; }
45.             next_local_next_sup(j,b,x,
                                    emptylocnext); }}}
46.    else Local-Prev-Sup(j,b):= (j,newlocprev); }
47. if (j,i) ∈ Next-Edge[j] ∧
       (j,i) ∈ Local-Next-Sup(j,b) then {
48.    remove((j,i),Local-Next-Sup(j,b));
49.    newlocnext := i;
50.    next_local_next_sup(j,b,newlocnext,emptylocnext);
51.    if emptylocnext then {
52.          remove(b, L[j]);
53.       for (j,x) ∈ Local-Prev-Sup(j,b) do {
54.          emptylocprev := false;
55.          remove((j,x),Local-Prev-Sup(j,b));
56.          if x = start then continue;
57.          while not emptylocprev do {
58.             if M[(j,x),b] = 0 then {
59.                List := List ∪ {[(j,x),b]};
60.                M[(j,x),b] := 1; }
61.             next_local_prev_sup(j,b,x,
                                    emptylocprev); }}}
62.    else Local-Next-Sup(j,b):= (j,newlocnext); }}
```

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

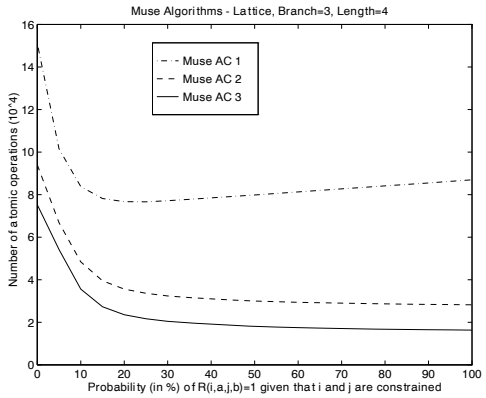

Figure 4: Number of operations performed by MUSE AC-1, MUSE AC-2, and MUSE AC-3 for a 3x4 lattice.

MUSE AC for all three topologies. We fixed the number of labels per variable to 16 and the probability of a constraint between two variables to 50% and varied the probability of $R2(i,a,j,b) = 1$ from 0% to 100% in steps of 5% (lower probability equals tighter constraints). Results for the lattice are displayed in Figure 4; results were similar for the other topologies. As can be seen, adapting the 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 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 produce a modest reduction in computational steps. Also evident in the figure is that the 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.

In Experiment 2, we investigated the influence of a lattice's shape on the number of operations required by MUSE AC-3. The parameters were the same as in the first experiment, except we generated lattices with varying path length and branching factors. As can be seen in Figure 5, increasing the number of nodes in a lattice increases the number of atomic operations performed. 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. 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.

In Experiment 3, MUSE AC-1, MUSE AC-2, and

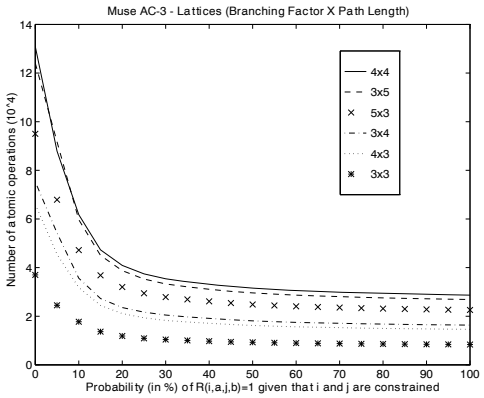

Figure 5: The number of operations performed by MUSE AC-3 for lattices of various sizes specified by branching factor x path length.

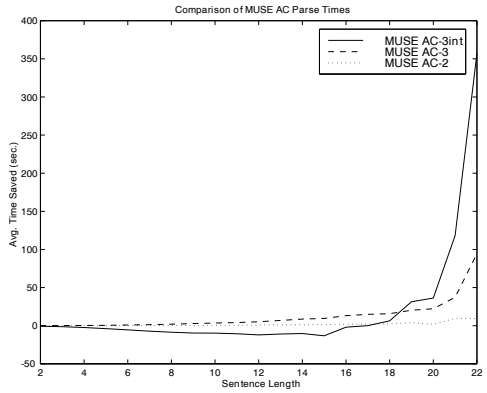

Figure 6: The Time saved by using MUSE AC-2, MUSE AC-3, and integrated MUSE AC-3 over MUSE AC-1.

MUSE AC-3 were incorporated into our CDG parser to perform arc consistency prior to extraction of legal parses. This parser uses methods developed by Harper and Helzerman [7, 8] to parse a sentence containing words with multiple lexical categories and multiple feature values. This method of parsing keeps the size of the MUSE network relatively small. For this experiment, we randomly parsed all of the sentences from the Resource Management Corpus [9] and compared the running times of MUSE AC-1, MUSE AC-2, and MUSE AC-3. We also compared the running time of MUSE AC-3 after it was integrated more tightly with the parsing algorithm [8]. As shown in Figure 6, MUSE AC-3 saves more time over MUSE AC-1 (30% reduction in filter time) than MUSE AC-2. The modest improvement of MUSE AC-2 over MUSE AC-1 (2% reduction in filter time) is likely due to the fact that our parsing algorithm applies the AC algorithm to moderately small constraint networks over multiple stages of parsing. Integrating MUSE AC-3 into the parser slows the parser down slightly for short sentences, suggesting that distributing the data structures across the CN incurs some overhead; however, that overhead is more than compensated for when parsing longer sentences.

Conclusion

We have shown that MUSE AC-2, simply by using the method of initializing and updating the S sets as in AC-6 [4], performs fewer operations than MUSE AC-1. MUSE AC-3 further improves upon MUSE AC-2 by using a fast method to initialize and update the DAG support sets. MUSE AC-3 is superior to MUSE AC-1 and MUSE AC-2 when they are used in CDG parsing. Tightly integrating MUSE AC-3 with the parser results in additional speedup. These runtime improvements have proven useful for speeding up spoken language understanding systems [6] and natural language front ends for multiple databases [7]. These algorithms should also be effective for other CSP problems that have segmental ambiguity, such as visual understanding and handwriting analysis.

References

[1] 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.

[2] 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.

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

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

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

[6] 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.

[7] 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.

[8] M. P. Harper, S. A. Hockema, and C. M. White. Enhanced constraint dependency grammar parsers. In *Proceedings of the IASTED International Conference on Artificial Intelligence and Soft Computing*, August 1999.

[9] 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.