2-1-1994

MUSE CSP: An Extension to the Constraint Satisfaction Problem

Randall A. Helzerman
Purdue University School of Electrical Engineering

Mary P. Harper
Purdue University School of Electrical Engineering

MUSE CSP: AN EXTENSION TO THE CONSTRAINT SATISFACTION PROBLEM

RANDALL A. HELZERMAN
MARY P. HARPER

TR-EE 94-8
FEBRUARY 1994

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

MUSE CSP: An Extension to the Constraint Satisfaction Problem *

Randall **A.** Helzerman and Mary P. Harper
School of Electrical Engineering
1285 Electrical Engineering Building
Purdue University
West Lafayette, IN 47907-1285
{helz,harper)@ecn.purdue.edu

'This work is supported in part by Purdue Research Foundation, NSF grant number IRI-9011179, and NSF Parallel Infrastructure Grant CDA-9015696.

Abstract

This paper describes an extension to the constraint satisfaction problem (CSP) approach called MUSE CSP (Multiply ***SE*gmented** Constraint Satisfaction Problem). This extension is especially useful for those problems which segment into multiple sets of partially shared variables. Such problems arise naturally in signal processing applications including computer vision, speech processing, and handwriting recognition. For these applications, it is often difficult to segment the data in only one way given the low-level information utilized by the segmentation algorithms. MUSE CSP can be used to efficiently represent and solve several similar instances of the constraint satisfaction problem simultaneously. If multiple instances of a CSP have some common variables which have the same domains and compatible constraints, then they can be combined into a single instance of a MUSE CSP, reducing the work required to enforce node and arc consistency.

1 Introduction

Constraint-satisfaction problems (CSP) have a rich history in Artificial Intelligence [2, 3, 4, 7, 8, 13, 14, 22] (see [12] for a survey of CSP). Constraint satisfaction provides a convenient way to represent and solve certain types of problems. In general, these are problems which 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, and diagnostic reasoning. When using a CSP approach (e.g., Figure 1), the variables are typically depicted as circles, where each circle is associated with a finite set of possible values, and the constraints imposed on the variables are depicted using arcs. An arc looping from a circle to itself represents a unary constraint (a relation involving a single variable), and an arc between two circles represents a binary constraint (a relation on two variables). A classic example of a CSP is the map coloring problem (e.g., Figure 1), where a color must be assigned to each country such that no two neighboring countries have the same color. A variable represents a country's color, and a constraint arc between two variables indicates that the two joined countries are adjacent and should not be assigned the same color.

Formally, a CSP [13, 18] is defined in Definition 1.

Definition 1 (Constraint Satisfaction Problem)
N = {i, j,...) is the set of nodes, with $|N| = n$,
L = {a, b,...) is the set of labels, with $|L| = 1$,
L; = $\{a|a \in$ L and (i, a) is admissible),
R1 is a unary relation, and (i, a) is admissible if R1(i, a),
R2 is a binary relation, (i, a) – (j, b) is admissible if *R2*(*i*, a, j, b).

A CSP network contains all n-tuples in L^n which satisfy R1 and *R2*. Since some of the labels associated with a variable may be incompatible with labels assigned to other variables, it is desirable to eliminate as many of these labels as possible by enforcing local consistency

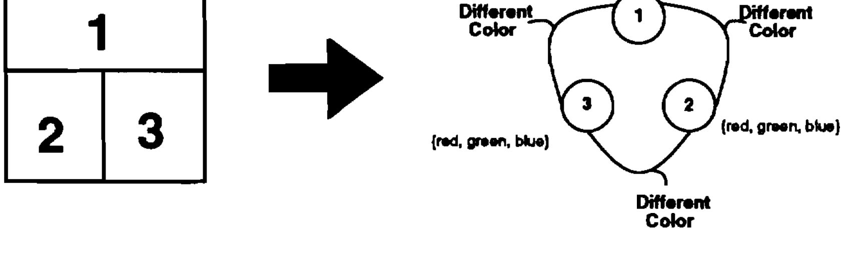


Figure 1: The map coloring problem as an example of CSP.

conditions (such as arc consistency) before a globally consistent solution is extracted [5]. Node and arc consistency are defined in Definitions 2 and 3 respectively.

Definition 2 (Node Consistency) *An instance of CSP is said to be* node consistent *if and only if each variable's domain contains only labels for which the unary relation $R1$ holds, i.e.:*

$$\forall i \in N : \forall a \in L_i : R1(i,a)$$

Definition 3 (Arc Consistency) *An instance of CSP is said to be* arc consistent *if and only if for every pair of nodes i and j, each element of L_i (the domain of i) has at least one element of L_j for which the binary relation $R2$ holds, i.e.:*

$$\forall i,j \in N : \forall a \in L_i : \exists b \in L_j : R2(i,a,j,b)$$

Node consistency is easily enforced by the operation $L_i = L_i \cap \{x|R1(i,x)\}$. Enforcing arc consistency is more complicated, but Mohr and Henderson [18] have designed an optimal algorithm (AC-4), which runs in $O(yl^2)$ time (where y is the number of pairs of nodes for which $R2$ does not hold).

There are many types of problems which can be solved by using this approach in a more or less direct fashion. There are also problems which might benefit from the CSP approach, but which are difficult to segment into a single set of variables. This is the class of problems our paper addresses. For example, suppose the map represented in Figure 1 were scanned by a noisy computer vision system, with a resulting uncertainty as to whether the line between regions 1 and 2 is really a border or an artifact of the noise. This situation would yield two CSP problems as depicted in Figure 2. A brute-force approach would be to solve both of the problems, which would be reasonable for scenes containing few ambiguous borders. However, as the number of ambiguous borders increases, the number of CSP networks would grow in a combinatorially explosive fashion. In the case of ambiguous segmentation, it might often be more efficient to merge the constraint networks into a single network which would compactly represent all of them simultaneously, as shown in Figure 3. In this paper, we develop an extension to CSP called MUSE CSP (Multiply *SE*gmented Constraint Satisfaction Problem) to represent and solve multiple instances of CSP problems.

The initial motivation for extending CSP came from our work in spoken language parsing [23, 11, 9]. The output of a hidden-Markov-model-based speech recognizer is often a list of the most likely sentence hypotheses (i.e., an N-best list) where parsing can be used to rule out the ungrammatical sentence hypotheses. Maruyama [15, 17, 16] has shown that parsing can be cast as a CSP with a finite domain, so constraints can be used to rule out syntactically

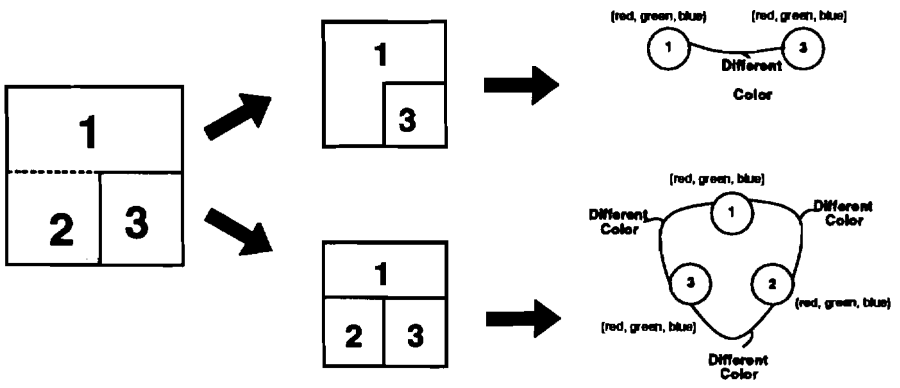


Figure 2: An ambiguous map yields two CSP problems.

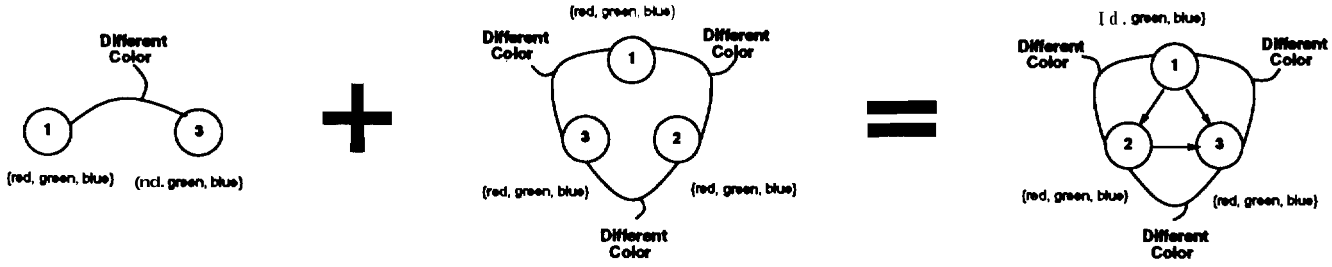


Figure 3: How the two CSP problems of Figure 2 can be captured by a single instance of MUSE CSP. The directed edges form a DAG such that the paths through the DAG correpond to instances of combined CSPs.

Its	hard	to	wreck	a	nice	beach
Its	hard	to	recognizes			speech

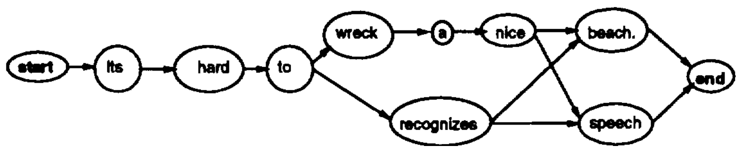


Figure 4: Multiple sentence hypotheses can be parsed simultaneously by propagating constraints over a 'word graph rather than individual sentences.

incorrect sentence hypotheses. However, individually processing each sentence hypothesis provided by a speech recognizer is inefficient since many sentence hypotheses are generated with a high degree of similarity. An alternative representation for a list of similar sentence hypotheses is a word graph or lattice of word candidates which contains information on the approximate beginning and end point of each word. Word graphs are typically more compact and more expressive than N-best sentence lists. In an experiment in [23], word graphs were constructed from three different lists of sentence hypotheses. The word graphs provided an 83% reduction in storage, and in all cases, they encoded more possible sentence hypotheses than were in the original list of hypotheses. In one case, 20 sentence hypotheses were converted into a word graph representing 432 sentence hypotheses. Figure 4 depicts a word graph containing eight sentence hypotheses which was constructed from two sentence hypotheses: ***Its hard to recognizes speech*** and ***It's hard to wreck a nice beach.*** If the spoken language parsing problem is structured as a MUSE CSP problem, then the constraints used to parse individual sentences would be applied to a word graph of sentence hypotheses, eliminating from further consideration all those hypotheses that are ungrammatical.

From CSP to MUSE CSP

If there are multiple, similar instances of a CSP which need to be solved, then separately enforcing node and arc consistency on each instance can often result in much duplicated work. To avoid this duplication, we have provided a way to combine the multiple instances of CSP into a shared constraint network and revised the node and arc consistency algorithms to support this representation.

Formally, we define MUSE CSP as follows:

Definition 4 (MUSE CSP)
$N = \{i, j, \ldots\}$ is the set of nodes, 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 $|L| = l$,
$L_i = \{a | a \in L$ and (i,a) is admissible in at least one segment$\}$,
R1 is a unary relation, and (i,a) is admissible if $R1(i,a)$,
R2 is a binary relation, $(i,a) - (j,b)$ is admissible if $R2(i,a,j,b)$.

The segments in C are the different sets of nodes representing instances of CSP which are combined to form a MUSE CSP. We also define $L_{(i,\sigma)}$ to be the set of all labels $a \in L_i$ that are admissible for some $a \in C$.

Because the number of segments in the set C can be exponential in n, it is important to define methods for combining instances of CSP into a single, compact MUSE CSP. To create a MUSE CSP, we must be able to determine when variables across several instances of CSP can be combined into a single shared variable, and we must also be able to determine which subsets of variables in the MUSE CSP correspond to individual CSPs. A word graph of sentence hypotheses is an excellent representation for a MUSE CSP based constraint parser for spoken sentences. Words that occur in more than one sentence hypothesis over the same time interval are represented using a single variable. The edges between the word nodes, which indicate temporal ordering among the words in the sentence, provide links between words in a sentence hypothesis. A sentence hypothesis, which corresponds to an individual CSP, is simply a path through the word nodes in the word graph going from a start node to an end node[1].

The concepts used to create a MUSE CSP network for spoken language can be adapted to other CSP problems. In particular, it is desirable to represent a MUSE CSP as a directed acyclic graph (DAG) where the paths through the DAG correspond to instances of CSP problems. In addition, CSP networks should be combined only if they satisfy the conditions below:

Definition 5 (MUSE CSP Combinability) p instances of CSP $C_1, \ldots, C_p$ *are* said to be MUSE Combinable iff the following conditions hold:

1. *If* $\{\sigma_1, \sigma_2, \ldots, \sigma_q\} \subseteq \{N_1, \ldots, N_p\} \wedge (i \in \sigma_1 \wedge i \in \sigma_2 \wedge \ldots \wedge i \in \sigma_q) \wedge (a \in L_{(i,\sigma_1)} \wedge a \in L_{(i,\sigma_2)} \wedge \ldots \wedge a \in L_{(i,\sigma_q)})$, then $R1_{\sigma_1}(i,a) = R1_{\sigma_2}(i,a) = \ldots = R1_{\sigma_q}(i,a)$.

[1]Note **that there may be more paths through the MUSE CSP than the number of original sentences.**

2. *If* $\{\sigma_1, \sigma_2, \ldots, \sigma_q\} \subseteq \{N_1, \ldots, N_p\} \wedge (i, j \in \sigma_1 \wedge i, j \in \sigma_2 \wedge \ldots \wedge i, j \in \sigma_q) \wedge (a \in L_{(i,\sigma_1)} \wedge a \in L_{(i,\sigma_2)} \wedge \ldots \wedge a \in L_{(i,\sigma_q)}) \wedge (b \in L_{(j,\sigma_1)} \wedge b \in L_{(j,\sigma_2)} \wedge \ldots \wedge b \in L_{(j,\sigma_q)})$, *then* $R2_{\sigma_1}(i,a,j,b) = R2_{\sigma_2}(i,a,j,b) = \ldots = R2_{\sigma_q}(i,a,j,b)$.

These conditions are not overly restrictive, requiring only that the labels for each variable i must be consistently admissible or inadmissible for all instances of CSP which are combined. These conditions do not uniquely determine which variables should be shared across CSP instances for a particular problem type. We define an operator $\oplus$ which combines instances of CSP into an instance of MUSE CSP in Definition 6.

Definition 6 ($\oplus$, the MUSE CSP Combining Operator) *If* $C_1, \ldots, C_p$ *are* MUSE *combinable instances of CSP, then* C $= C_1 \oplus \ldots \oplus C_p$ *will* be *an instance of* MUSE CSP *such that:*

$$
\begin{aligned}
&N = N_1 \cup \ldots \cup N_p \\
&\Sigma = \{N_1, \ldots, N_p\} \\
&L = L_1 \cup \ldots \cup L_p \\
&L_i = L_{(i,N_1)} \cup L_{(i,N_2)} \cup \ldots \cup L_{(i,N_p)} \\
&R1(i, a) = \bigvee_{\sigma=N_1}^{N_p} (R1_\sigma(i, a) \wedge a \in L_{(i,\sigma)}) \\
&R2(i,a,j,b) = \bigvee_{\sigma=N_1}^{N_p} (R2_\sigma(i,a,j,b) \wedge (a \in L_{(i,\sigma)} \wedge b \in L_{(j,\sigma)}))
\end{aligned}
$$

As previously mentioned, a DAG is an excellent representation for C, where its nodes are the elements of N, and its edges are arranged such that every a in C maps onto a path through the DAG. In some applications, such as speech recognition [23, 11, 9], the DAG will already be available to us. In applications where the DAG is not available, the user must determine the best way to combine instances of CSP to maximize sharing. We have provided an algorithm (shown in Figure 5) which automatically constructs a single instance of a MUSE CSP from multiple CSP instances in $O(sn \log n)$ time, where s is the number of CSP instances to combine, and n is the number of nodes in the MUSE CSP. This algorithm requires the user to assign numbers to variables in the CSPs such that variables that can be shared are assigned the same number. As shown by the examples in Figure 5, for a given set of CSPs, the greater the intersection between the sets of node numbers across CSPs, the more compact the MUSE CSP.

A MUSE CSP can be solved with a modified backtracking algorithm which finds all possible consistent label assignments, where the search space is pruned hy enforcing local consistency conditions, such as node and arc consistency. However, to gain the efficiency

```
1.  Assign each element i of N some number v in the range from 1 to n by using ord(i) = v.
2.  for each a ∈ Σ do
3.      begin
4.          Add to a two distinguished nodes called start and end and
               let ord(start)=0 and ord(end)=∞.
5.          Sort the elements of a by their ordinal numbers.
6.          for i, j ∈ a such that i's position immediatly preceeds j's position
7.              in the sorted a do
8.              begin
9.                  Next-edge_i :=Next-edge_i ∪{(i,j)}
10.                 Prev-edge_j :=Prev-edge_j ∪{(i, j)}
11.             end
12.     end
```

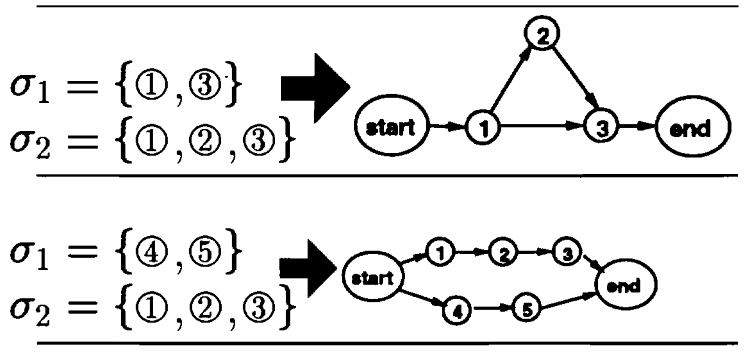


Figure 5: The algorithm to create a DAG to represent the set Σ, and examples of its action.

resulting from enforcing local consistency conditions before backtracking, node and arc consistency must be modified for MUSE CSP.

Definition 7 (MUSE Node Consistency) *An instance of MUSE CSP is said to be* node consistent *if and only if each variable's domain L_i contains only labels for which the unary relation $R1$ holds, i.e.:*

$$\forall i \in N : \forall a \in L_i : R1(i,a)$$

Definition 8 (MUSE Arc Consistency) *An instance of MUSE CSP is said to be* arc consistent *if and only if for every label a in each domain L_i there is at least one segment a whose nodes'domains contain at least one label b for which the binary relation $R2$ holds, i.e.:*

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

A MUSE CSP is node consistent if all of its segments are node consistent. Unfortunately, arc consistency in a MUSE CSP requires more attention because even though a binary constraint might disallow a label in one segment, it might allow it in another segment. 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 which together with a satisfy the binary constraints. However, in a MUSE CSP, before a label can be eliminated from a node, it must fail to satisfy the binary constraints in all the segments in which it appears. Therefore,

Notation	Meaning
(i,j)	An ordered pair of nodes.
$[(i,j),a]$	An ordered pair of a node pair (i, j) and a label $a \in L_i$.
$M[(i,j),a]$	$M[(i,j),a] = 1$ indicates that the label a is not admissable for (and has already been eliminated from) all segments containing i and j.
E	All node pairs (i, j) such that there exists a segment which contains both i and j.
$S[(i,j),a]$	$[(j,i),b] \in S[(i,j),a]$ means that label a at node i and b at j are simultaneously admissible.
Next-edge;	If a directed edge from i to j exists in E then (i,j) is a member of this set.
Prev-edge;	If a directed edge from j to i exists in E then (j, i) is a member of this set.
Counter$[(i,j),a]$	The number of labels in L_j which are compatible with a in L_i.
Prev-Support$[(i,j),a]$	$(i,k) \in$ Prev-Support$[(i,j),a]$ means that a is admissible in every segment which contains i, j, and k.
Next-Support$[(i,j),a]$	$(i,k) \in$ Next-Support$[(i,j),a]$ means that a is admissible in every segment which contains i, j, and k.
Local-Prev-Support(i,a)	A set of elements (i,j) such that $(j,i) \in$ Prev-edge; and a is compatible with at least one of j's labels.
Local-Next-Support(i,a)	A set of elements (i, j) such that $(i,j) \in$ Next-edge$_i$ and a is compatible with at least one of j's labels.
List	A queue of arc support to be deleted.

Figure 6: Data structures and notation for the MUSE CSP arc consistency algorithm.

the definition of MUSE arc consistency is modified as shown in Definition 8. Notice that Definition 8 reduces to Definition 3 when the number of segments is one.

3 The MUSE CSP Arc Consistency Algorithm

MUSE arc consistency[2] is enforced by removing from the domains those labels in L_i which violate the conditions of Definition 8. MUSE AC-1 builds and maintains several data structures, described in Figure 6, to allow it to efficiently perform this operation. Figure 9 shows the code for initializing the data structures, and Figure 10 contains the algorithm for eliminating inconsistent labels from the domains.

If label a at node i is compatible with label b at node j, then a ***supports b.*** To keep track of how much support each label a has, the number of labels in L_j which are compatible

[2]We purposely keep our notation and presentation as close as possible to that of Mohr and Henderson in [18], in order to aid understanding for those already familiar with the literature on arc consistency.

with a in L_i are counted, and the total is stored in **Counter**[(i, j),a]. The algorithm must also keep track of which labels that label a supports by using S[(i, j),a], which is a set of arc-label pairs. For example, S[(i, j),a] = {[(j,i), b], [(j,i),c]} means that a in L_i supports b and c in L_j. If a is ever invalid for L_i then b and c will loose some of their support. This is accomplished by decrementing **Counter**[(j, i),b] and **Counter**[(j, i),c]. For regular CSP arc consistency, if **Counter**[(i, j),a] becomes zero, a would automatically be removed from L_i, because that would mean that a was incompatible with every label for j. However, in MUSE arc consistency, this is not the case, because even though a does not participate in a solution for any of the segments which 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.

By representing Σ as a DAG, the algorithm is able to use the properties of the DAG to identify local (and hence efficiently computable) conditions under which labels become globally inadmissible. Consider Figure 7, which shows the nodes which are adjacent to node i in the DAG. Because every segment in the DAG which contains node i is represented as a path in the DAG going through node i, either node j or node k must be in every segment containing i. Hence, if the label a is to remain in L_i, it must be compatible with at least one label in either L_j or L_k. Also, because either n or m must be contained in every segment containing i, if label a is to remain in L_i, it must also be compatible with at least one label in either L_n or L_m.

In order to track this dependency, two sets are maintained for each label a at node i, **Local-Next-Support**$(i,$a) and **Local-Prev-Support**$(i,$a). **Local-Next-Support**$(i,$a) is a set of ordered node pairs (i,j) such that (i,j) $\in$ Next-edge$_i$, and there is at least one label b $\in L_j$ which is compatible with a. **Local-Prev-Support**(i, a) is a set of ordered pairs (i, j) such that (j,i) $\in$ Prev-edge$_i$ and there is at least one label b $\in L_j$ which is compatible with a. Whenever one of i's adjacent nodes, j, no longer has any labels b in its domain which are compatible with a, then $(i,$j) should be removed from **Local-Prev-Support**(i, a) or **Local-Next-Support**$(i,$a), depending on whether the edge is from j to i or from i to j, respectively. If either Local-Prev-Support(i,a) or Local-Next-Support(i,a) becomes the empty set, then a is no longer a part of any solution, and may be eliminated from L_i. In Figure 7, the label a is admissible for the segment containing i and j, but not for the segment containing i and

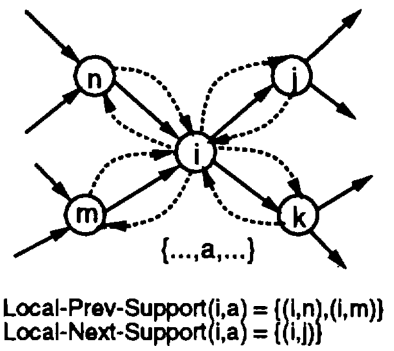

n
j
i
m
k
{...,a,...}
Local-Prev-Support(i,a) = {(i,n),(i,m)}
Local-Next-Support(i,a) = {(i,j)}

Figure 7: Local-Prev-Support and Local-Next-Support for an example DAG. The sets indicate that the label a is allowed for every segment which contains n, m, and j, but is disallowed for every segment which contains k.

k. If because of constraints, the labels in j become inconsistent with a on i, (i,j) would be eliminated from Local-Next-Support(a, i), leaving an empty set. In that case, a would no longer be supported by any segment.

The algorithm can utilize similar conditions for nodes which may not be directly connected to i by Next-edge; or Prev-edge;. Consider Figure 8. Suppose that the label a at node i is compatible with a label in L_j, but it is incompatible the labels in L_x and L,, then it is reasonable to eliminate a for all segments containing both i and j, because those segments would have to include either node x or y. To determine whether a label is admissible for a set of segments containing i and j, we calculate **Prev-Support**$[(i,$j),a] and **Next-Support**$[(i,$j),a] sets. **Next-Support**$[(i,$j),a] includes all (i,k) arcs which support a in i given that there is a directed edge between j and k, and (i,j) supports a. **Prev-Support**$[(i,$j),a] includes all (i,k) arcs which support a in i given that there is a directed edge between k and j, and (i,j) supports a. Note that **Prev-Support**$[(i,$j),a] will contain an ordered pair (i,j) if (i,j) $\in$ **Prev-edge**$_j$, and **Next-Support**$[(i,j),a]$ will contain an ordered pair (i,j) if (j,i) $\in$ Next-edge,. These elements are included because the edge between nodes i and j is sufficient to allow j's labels to support a in the segment containing i and j. Dummy ordered pairs are also created to handle cases where a node is at the beginning or end of a network: when (start,j) $\in$ **Prev-edge**$_j$, (i,start) is added to Prev-support[(i,j),a], and when (j,end) $\in$ **Next-edge**$_j$, (i,end) is added to **Next-support**$[(i,$j),a]. This is to prevent a label from being ruled out because no nodes precede or follow it in the DAG. Figure 9 shows the Prev-Support, Next-Support, Local-Next-Support, and Local-Prev-Support sets

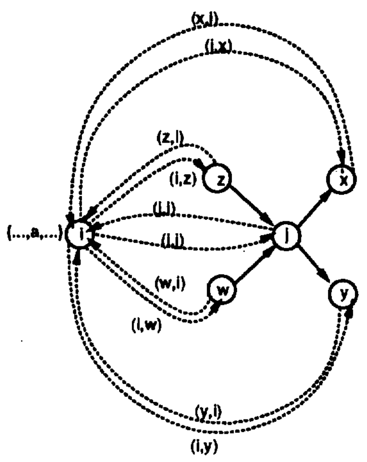
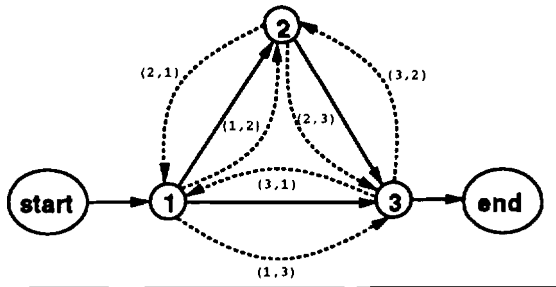


Figure 8: If Next-edge$_j$ = {(j, x),(j,y)) and S[$(i$,x),a] = ϕ and S[$(i$,y),a] = ϕ, then a is inadmissible for every segment containing both i and j.

that the initialization algorithm creates for the label a in the simple example DAG.

To illustrate how these data structures are used in MUSE AC-1 (see Figure 10), consider what happens if initially [(1,3),a] $\in$ List for the MUSE CSP in Figure 9. First, it is necessary to remove [(1,3),a]'s support from all S[(3,1),x] such that [(3,1),x] $\in$ S[(1,3),a] by decrementing for each x, **Counter**[(3,1),x] by one. If the counter for any [(3,1),x] becomes 0, and the value has not already been placed on the List, then it is added for future processing. Once this is done, it is necessary to remove [(1,3),a]'s influence on the DAG. To handle this, we examine the two sets Prev-Support[(1,3),a] = {(1,2),(1,3)} and Next-**Support**[(1,3),a] = {(1,end)). Note that the value (1,end) in **Next-Support**[(1,3),a] and the value (1,3) in Prev-Support[(1,3),a], once eliminated from those sets, require no further action because they are dummy values. However, the value (1,2) in Prev-Support[(1,3),a] indicates that (1,3) is a member of Next-Support[(1,2),a], and since a is not admissible for (1,3), (1,3) should be removed from Next-Support[(1,2),a], leaving an empty set. Note that because **Next-Support**[(1,2),a] is empty and assuming that M[(1,2),a] = 0, [(1,2),a] is added to List for further processing. Next, (1,3) is removed from **Local-Next-Support**(1,a), but that set is non-empty. During the next iteration of the while loop [(1,2),a] is popped from List. When **Prev-Support**[(1,2),a] and Next-Support[(1,2),a] are processed, Next-Support[(1,2),a] = ϕ and Prev-Support[(1,2),a] contains only a dummy, which is removed. When (1,2) is removed from **Local-Next-Support**(1,a), the set becomes empty, so a is no

```
1.  List:=$\phi$;
2.  E := {(i, $j$)|$\exists\sigma \in \Sigma$ : $i$, j $\in$ a A $i \neq j$ A $i, j \in$ N);
3.  for ($i$, j) $\in$ E do
4.      for a $\in L_i$ do
5.          begin
6.              $M$[($i$, j), a] := 0;
7.              Prev-Support[($i$, j), a] := 4; Next-Support[(;, j), a] := 4;
8.              Local-Prev-Support($i$, a) := 4; Local-Next-Support(;, a) := 4;
9.              $S$[($i, j$), $a$] := 4;
10.         end
11. for (i, j) $\in$ E do
12.     for a $\in L_i$ do
13.         begin
14.             Total:=0;
15.             for b$\in L_j$ do
16.                 if $R2$($i$, a, j, b) then .
17.                   begin
18.                       Total:=Total+1;
19.                       $S$[($j, i$), $b$] := $S$[($j, i$), $b$] $\cup$ {[($i, j$), $a$]};
20.                   end
21.             if Total=0 then
22.                 begin
23.                     $M$[($i, j$), a] := 1;
24.                     List:=List $\cup${[($i$, j), a]);
25.                 end
26.             Counter[($i$, j), $a$]:=Total;
27.             Prev-Support[($i$, j), a] := {(i, $x$)|($i, x$) $\in$ E A (z, j) $\in$ Prev-edge$_j$)
                        $\cup${($i, j$)|($i$, j) $\in$ Prev-edge$_j$) $\cup$ {($i$, start)|(start, $j$) $\in$ Prev-edge$_j$);
28.             Next-Support[(;, j), a] := {(i, $x$)|($i, x$) $\in$ E $\wedge$ (j, z) $\in$ Next-edge$_j$)
                        $\cup${($i, j$)|($j, i$) $\in$ Next-edge$_j$) $\cup${(i, end)|($j$, end) $\in$ Next-edge$_j$);
29.             if (i, j) $\in$ Next-edge; then
30.                 Local-Next-Support($i, a$):=Local-Next-Support($i$, a) $\cup${($i$, j));
31.             if (j, i) $\in$ Prev-edge; then
32.                 Local-Prev-Support($i, a$):=Local-Prev-Support($i$, a) $\cup${(i, j));
33.         end
```



Prev-Support[(1,2), a] = {(1,2)}	Prev-Support[(1,3), a] = {(1,2),(1,3)}	Prev-Support[(2,3), a] = {(2,3),(2,1)}
Next-Support[(1,2), a] = {(1,3)}	Next-Support[(1,3), a] = {(1,end)}	Next-Support[(2,3), a] = {(2,end)}
Prev-Support[(2,1), a] = {(2,start)}	Prev-Support[(3,1), a] = {(3,start)}	Prev-Support[(3,2), a] = {(3,1)}
Next-Support[(2,1), a] = {(2,1),(2,3)}	Next-Support[(3,1), a] = {(3,1),(3,2)}	Next-Support[(3,2), a] = {(3,2)}
Local-Prev-Support(1, a) = {(1,start)}	Local-Prev-Support(2, a) = {(2,1)}	Local-Prev-Support(3, a) = {(3,1),(3,2)}
Local-Next-Support(1, a) = {(1,2),(1,3}	Local-Next-Support(2, a) = {(2,3)}	Local-Next-Support(3, a) = {(3,end)}

Figure 9: Algorithm for initializing the MUSE CSP data structures along with a simple example. The dotted lines are members of the set E.

```
while $List \neq \phi$ do
    begin
        choose $[(j,i),b]$ from List and remove it from List;
        for $[(i,j),a] \in S[(j,i),b]$ do
            begin
                Counter$[(i,j),a]$:=Counter$[(i,j),a] - 1$;
                if Counter$[(i,j),a] = 0 \wedge M[(i,j),a] = 0$ then
                    begin
                        List:=List $\cup \{[(i,j),a]\}$;
                        $M[(i,j),a] := 1$;
                    end
            end
        for $(j,z) \in$ Next-Support$[(j,i),b]$ do
            begin
                Prev-Support$[(j,z),b]$:=Prev-Support$[(j,z),b] - \{(j,i)\}$;
                if Prev-Support$[(j,z),b] = \phi \wedge M[(j,z),b] = 0$ then
                    begin
                        List:=List $\cup \{[(j,z),b]\}$;
                        $M[(j,z),b] := 1$;
                    end
            end
        for $(j,z) \in$ Prev-Support$[(j,i),b]$ do
            begin
                Next-Support$[(j,z),b]$:=Next-Support$[(j,z),b] - \{(j,i)\}$;
                if Next-Support$[(j,z),b] = \phi \wedge M[(j,z),b] = 0$ then
                    begin
                        List:=List $\cup \{[(j,z),b]\}$;
                        $M[(j,z),b] := 1$;
                    end
            end
        if $(j,i) \in$ Next-edge$_j$ then
            Local-Next-Support$(j,b)$:=Local-Next-Support$(j,b) - \{(j,i)\}$;
        if Local-Next-Support$(j,b) = \phi$ then
            begin
                $L_j := L_j - \{b\}$;
                for $(j,z) \in$ Local-Prev-Support$(j,b)$ do
                    if $M[(j,z),b] = 0$ then
                        begin
                            List:=List $\cup \{[(j,z),b]\}$;
                            $M[(j,z),b] := 1$;
                        end
            end
        if $(i,j) \in$ Prev-edge$_j$ then
            Local-Prev-Support$(j,b)$:=Local-Prev-Support$(j,b) - \{(j,i)\}$;
        if Local-Prev-Support$(j,b) = \phi$ then
            begin
                $L_j := L_j - \{b\}$;
                for $(j,z) \in$ Local-Next-Support$(j,b)$ do
                    if $M[(j,z),b] = 0$ then
                        begin
                            List:=List $\cup \{[(j,z),b]\}$;
                            $M[(j,z),b] := 1$;
                        end
            end
    end
```

Figure 10: Algorithm to enforce MUSE CSP arc consistency.

longer compatible with any segment containing 1 and can be eliminated from further consideration as a possible label for node 1. Once a is eliminated from **node** 1, it is also necessary to remove the support of a $\in L_1$ from all labels on nodes that precede node 1, that is for all nodes x such that $(1, x) \in$ **Local-Prev-Support**$(1, a)$. Since Local-Prev-Support$(1, a) = \{(1, \textbf{start})\}$, and start is a dummy node, there is no more work to be done.

In contrast, consider what happens if initially $[(1,2), a] \in$ List for the MUSE CSP in Figure 9. In this case, Prev-Support$[(1,2), a]$ contains $(1,2)$ which requires no additional work; whereas, Next-Support$[(1,2), a]$ contains $(1,3)$, indicating that $(1,2)$ must be removed from Prev-Support$[(1,3), a]$'s set. After the removal, Prev-Support$[(1,3), a]$ is non-empty, so the segment containing nodes 1 and 3 still supports the label a on 1. The reason that these two cases provide different results is that nodes 1 and 3 are in every segment; whereas, nodes 1 and 2 are only in one of them.

3.1 The Running Time of MUSE AC-1

The running time of the routine to initialize the MUSE CSP data structures (in Figure 9) is $O(n^2l^2 + n^3l + n^2l)$, where n is the number of nodes in a MUSE CSP and l is the number of labels. Given that the number of (i, j) elements in E is $O(n^2)$ and the number of labels in L_i and L_j is $O(l)$, there are $O(n^2l)$ counters and S sets to calculate values for. To determine the number of supporters for a given arc-label pair requires $O(l)$ work; hence, the initializing all of the counters and S sets requires $O(n^2l^2)$ time. However, the determination of each Prev-Support$[(i,j),a]$ and Next-Support$[(i,j),a]$ requires $O(n)$ time, so the time required to calculate all Prev-Support and Next-Support sets is $O(n^3l)$. Finally, the time needed to calculate all Local-Next-Support and Local-Prev-Support sets is $O(n^2l)$ because there are $O(nl)$ sets with up to $O(n)$ elements per set.

The running time for the algorithm which prunes labels that are not arc consistent (in Figure 10) also operates in $O(n^2l^2 + n^3l + n^2l)$ time. Clearly there are only $O(n^2l)$ counters to keep track of in the algorithm. Each counter can be at most l in magnitude, and, it can never become negative, so the maximum running time for line 6 in the algorithm (given that elements, because of M, appear on the list only once) is $O(n^2l^2)$. Because there are $O(n^2l)$ Prev-Support and Next-Support Lists, each up to $O(n)$ in size, the running time required for lines 15 and 24 is O($n^3$1). Finally, since there are $O(nl)$ Local-Next-Support and Local-Prev-Support sets to eliminate $O(n)$ elements from, the running time of lines 32 and 44 is O($n^2$1). Hence, the running time of the MUSE CSP arc consistency algorithm is $O(n^2l^2 + n^3l +$ $n^2$1). By comparison, the running time for CSP arc consistency is (n^2l^2), assuming that there are n^2 constraint arcs. Note that for applications where 1 = n, the running times of the algorithms are the same (this is true for parsing spoken language with a MUSE CSP). Also, if Σ is representable as planar DAG (in terms of Prev-edge and Next-Edge, not E), then the running time of the algorithms is the same because the average number of values in Prev-Support and Next-Support would be a constant. In the general case, the increase in the running time for arc consistency of a MUSE CSP is reasonable considering that it is possible to combine a large number of CSP instances (possibly exponential) into a compact graph with a small number of nodes.

3.2 Correctness of MUSE AC-1

Next we prove the correctness of MUSE AC-4. A label is eliminated from a domain by MUSE AC-4 only if its Local-Prev-Support or its Local-Next-Support becomes empty. To prove this, we must show that a label's local support sets become **empty** if and only if that label cannot participate in a MUSE arc consistent solution. This is proven for Local-Next-Support (Local-Prev-Support follows by symmetry.)' Observe that if $a \in$ L;, and it is incompatible with all of the nodes which immediately follow L_i in the DAG, then it cannot participate in a MUSE arc consistent solution. In line 32 in Figure 10, (i,j) is removed from Local-Next-Support(i,a) set only if [(i,j),a] has been popped off List.

Therefore, it must be shown that [(i,j),a] is put on List only if $a \in L_i$ is incompatible with every segment which contains i and **j.** This is proven by induction on the number of iterations of the while loop in Figure 10.

Base case: The initialization routine only puts [(i,j),a]on List if $a \in$ L; is incompatible with every label in L_j (line 24 of Figure 9). Therefore, $a \in L_i$ is in no solution for any segments which contain i and j.

Induction step: Assume that at the start of the kth iteration of the while loop all [(x,y),c] which have ever been put on List indicate that $c \in L_x$ is incompatible with every segment which contains x and y. It remains to show that during the kth iteration, if [(i,j),a]

is put on List, then $a \in L_i$ is incompatible with every segment which contains i and **j**. There are four ways in which a new [(i,j),a] can be put on List:

1. All labels in L_j which were once compatible with $a \in L_i$ have been eliminated. This item could have been placed on List either during initialization (see line 24 in Figure 9) or during a previous iteration of the while loop (see line 9 in Figure 10)), just as in the CSP AC-4 algorithm. It is obvious that, in this case, $a \in L_i$ is incompatible with every segment containing i and **j**.

2. **Prev-Support**$[(i,j),a] = \phi$ (see line 16 in Figure 10) indicating that $a \in L_i$ is incompatible with all nodes k for $(k,j) \in$ Prev-Edge$_j$. The only way for $[(i,j),a]$ to be placed on List for this reason (at line 18) is because all tuples of the form [(i,k),a] (where $(k,j) \in$ **Prev-edge**$_j$) were already put on List. By the induction hypothesis, these [(i,k),a] items were placed on the List because $a \in L_i$ is incompatible with all segments containing i and k in the DAG. But if a is incompatible with every node which immediately precedes **j** in the DAG, then a is incompatible with every segment which contains j. Therefore, it is correct to put [(i,j),a] on List.

3. **Next-Support**$[(i,j),a] = \phi$ (see line 25 in Figure 10) indicating that $a \in L_i$ is **incom**patible with all nodes k for $(j,k) \in$ Next-Edge$_j$. The only way for $[(i,j),a]$ to be placed on List (at line 27) for this reason is because all tuples of the form [(i,k),a] (where $(j,k) \in$ Next-edge$_j$) were already put on List. By the induction hypothesis, these [(i,k),a] items were placed on the List because $a \in L_i$ is incompatible with all segments containing i and k in the DAG. But if a is incompatible with every node which immediately follows **j** in the DAG, then a is incompatible with every segment which contains **j**. Therefore, it is correct to put [(i,j),a] on List.

4. **Local-Next-Support**$(i,a) = \phi$ (see line 33 in Figure 10) indicating **that** $a \in L_i$ is no longer compatible with all nodes k such that $(i,k) \in$ Next-Edge$_i$. The only way for [(i,j),a] to be placed on List (at line 39) for this reason is because no node which follows i in the DAG supports a, and so all pairs (i,k) have been legally removed from **Local-Next-Support**(i,a) during previous iterations. Because there is no segment containing i which supports a, it follows that no segment containing i and j supports that label.

At the beginning of the (k + 1)th iteration of the while loop, every [(x, y), c] on List implies that c is incompatible with every segment which contains x and y. Therefore, by induction, it is true for all iterations of the while loop in Figure 10.

4 MUSE CSP Path Consistency

Path consistency ensures that any pair of labelings $(i,b) - (j,c)$ allowed by the (i,j) arc directly are also allowed by all paths from i to j. Montanari [19] has proven that to ensure path consistency for a complete graph, it suffices to check every path of length two. A definition of path consistency for a CSP is shown in Definition 9. In that definition, we use the predicate $Path(i,k,j)$ to indicate that there is a path of arcs in E between i and j which goes through k.

Definition 9 (Path Consistency) An instance of *CSP* is said to be path consistent if and only if:

$\forall i,j \in N : i \neq j \Rightarrow (\forall a \in L_i : \forall b \in L_j : \forall k \in N : k \neq i \wedge k \neq j \wedge Path(i,k,j) \Rightarrow$
$(R2(i,a,j,b) \Rightarrow \exists c \in L_k : R2(i,a,k,c) \wedge R2(k,c,j,b)))$

Path consistency can also be ensured for MUSE CSP problems. The definition of MUSE path consistency is shown below:

Definition 10 (MUSE Path Consistency) An instance of *MUSE CSP* is said to be path consistent if and only if:

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

MUSE path consistency is enforced by removing from the domains those labels in L_i and L_j which violate the conditions of Definition 10. MUSE PC-1 builds and maintains several data structures comparable to the data structures defined for MUSE AC-1. Figure 11 shows the code for initializing the data structures, and Figure 12 contains the algorithm for eliminating inconsistent labels from the domains.

MUSE PC-1 must keep track of which labels in L_k concurrently support, label a at node i and label b at node j. To keep track of how much path support each arc-label triple $[(i,j),a,b]$ has, the number of labels in L_k which satisfy the relation $R2(i,a,k,c) \wedge R2(k,c,j,b)$ are counted using Counter$[(i,j),k,a,b]$. Additionally, the algorithm must keep track of which arc-label triples are supported by the label $c \in L_k$. In the case that $c \in L_k$ supports $[(i,j),a,b]$, we add the value $[(i,j),a,b]$ to the set $S[(k,i),c]$. In the case that $c \in L_k$

supports $[(j,i),b,a]$, we add the value $[(j,i),b,a]$ to the set $S[(k,j),c]$. MUSE PC-1 also uses the Local-Next-Support, Local-Prev-Support, Prev-Support, and Next-Support sets just as in MUSE AC-1. The algorithm for setting up these data structures for an instance of a MUSE CSP is shown in Figure 11.

To prune the graph of all labels that are path inconsistent in a MUSE CSP, it is important to ensure that one segment does not disallow a path consistent solution in another segment. Hence, as soon as one path inconsistency is found, we must remove support for the values on other previously consistent paths by using the properties of the DAG, as in arc consistency. If there is no segment $\sigma \in \Sigma$ which contains $Path(i,k,j)$ such that there exists $c \in L_k$ such that $R2(i,a,k,c)$ and $R2(k,c,j,b)$, then a can be eliminated from L_i, and b can be eliminated from L_j. The routine for eliminating inconsistent labels from the domains is shown in Figure 12. If $[(k,x),c]$ appears on List, then $c \in L_k$ no longer participates in a path which contains both k and x.

4.1 The Running Time of MUSE PC-1

The running time of the routine to initialize the MUSE CSP data structures (in Figure 11) is $O(n^3l^3 + n^3l + n^2l)$, where n is the number of nodes in a MUSE CSP and l is the number of labels. Given that the number of (i,j) elements in E is $O(n^2)$ and the number of labels in L_i and L_j is $O(l)$, there are $O(n^3l^2)$ $\textbf{Counter}[(i,j),k,a,b]$s to calculate values for. To determine the number of supporters for a given counter requires $O(l)$ work; hence, the initializing all of the counters requires $O(n^3l^3)$ time. Additionally, there are $O(n^2l)$ $S[(i,k),a]$ sets to determine. Each support set can have up to $O(nl^2)$ values, so the time required to initialize the support sets. is $O(n^3l^3)$. Determining each $\textbf{Prev-Support}[(i,j),a]$ and $\textbf{Next-Support}[(i,j),a]$ requires $O(n)$ time, so the time required to calculate all Prev-Support and Next-Support sets is $O(n^3l)$. Finally, the time needed to calculate all Local-Next-Support and Local-Prev-Support sets is $O(n^2l)$ because there are $O(nl)$ sets with up to $O(n)$ elements per set.

The running time for the algorithm which prunes labels that are not path consistent (in Figure 12) also operates in $O(n^3l^3 + n^3l + n^2l)$ time. Clearly there are $O(n^3l^2)$ counters to keep track of in the algorithm. Each counter can be at most l in magnitude, and, it can never become negative, so the maximum running time for line 6 in the algorithm (given that

```
1. List:=$\phi$;
2. E :=$\{(i, j) | \exists\sigma \in \Sigma : i, j \in a \wedge i \neq j \wedge i, j \in N\}$;
3. for $(i,j) \in E$ do
4.     for $a \in L_i$ do
5.         begin
6.             $M[(i,j),a] := 0$;
7.             Prev-Support$[(i,j),a] := \phi$; Next-Support$[(i,j),a] := \phi$;
8.             Local-Prev-Support$(i,a) := \phi$; Local-Next-Support$(i,a) := \phi$;
9.             for $k \in N$ such that $k \neq i$, $k \neq j$ do
10.                for $c \in L_k$ do
11.                    $S[(k,i),c] := \phi$;
12.        end
13. for $(i,j) \in E$ do
14.     for $a \in L_i$ do
15.         begin
16.             for $b \in L_j$ such that $R2(i,a,j,b)$ do
17.                 for $k \in N$ such that $k \neq i$, $k \neq j$ do
18.                     begin
19.                         Total:=0;
20.                         for $c \in L_k$ do
21.                             if $R2(i,a,k,c)$ and $R2(k,c,j,b)$ then
22.                                 begin
23.                                     Total:=Total+1;
24.                                     $S[(k,i),c] := S[(k,i),c] \cup \{[(i,j),a,b]\}$;
25.                                 end
26.                         if Total=0 then
27.                             begin
28.                                 $M[(i,k),a] := 1$; $M[(j,k),b] := 1$;
29.                                 List:=List $\cup\{[(i,k),a],[(j,k),b]\}$;
30.                             end
31.                         Counter$[(i,j),k,a,b]$:=Total;
                        end
                Prev-Support$[(i,j),a] := \{(i,x) | (i,z) \in E \wedge (z,j) \in$ Prev-edge$_j\}$
                        $\cup\{(i,j) | (i,j) \in$ Prev-edge$_j\} \cup \{(i,$start$) | ($start$,j) \in$ Prev-edge$_j\}$;
                Next-Support$[(i,j),a] := \{(i,z) | (i,z) \in E \wedge (j,z) \in$ Next-edge$_j\}$
                        $\cup\{(i,j) | (j,i) \in$ Next-edge$_j\} \cup \{(i,$end$) | (j,$end$) \in$ Next-edge$_j\}$;
35.             if $(i,j) \in$ Next-edge$_i$ then
36.                 Local-Next-Support$(i,a)$:=Local-Next-Support$(i,a) \cup \{(i,j)\}$;
37.             if $(j,i) \in$ Prev-edge$_i$ then
38.                 Local-Prev-Support$(i,a)$:=Local-Prev-Support$(i,a) \cup \{(i,j)\}$;
39.         end
```

Figure 11: Algorithm for initializing the MUSE CSP data structures for path consistency.

```
while $List \neq \phi$ do
    begin
        choose $[(k,z),c]$ from List and remove it from List;
        for $[(x,y),a,b] \in S[(k,x),c]$ do
            begin
                Counter$[(x,y),k,a,b]$:=Counter$[(x,y),k,a,b] - 1$;
                if Counter$[(x,y),k,a,b] = 0$ then
                  begin
                      if $M[(z,k),a] = 0$ then
                        begin
                            List:=List $\cup\{[(x,k),a]\}$;
                            $M[(x,k),a] := 1$;
                        end
                      if $M[(y,k),b] = 0$ then
                        begin
                            List:=List $\cup\{[(y,k),b]\}$;
                            $M[(y,k),b] := 1$;
                        end
                  end
            end
        for $(k,z) \in$ Next-Support$[((),z),c]$ do
            begin
                Prev-Support$[(k,z),c]$:=Prev-Support$[(k,z),c] - \{(k,z)\}$;
                if Prev-Support$[(k,z),c] = \phi \wedge M[(k,z),c] = 0$ then
                  begin
                      List:=List $\cup\{[(k,z),c]\}$;
                      $M[(k,z),c] := 1$;
                  end
            end
        for $(k,z) \in$ Prev-Support$[(k,z),c]$ do
            begin
                Next-Support$[(k,z),c]$:=Next-Support$[(k,z),c] - \{(k,z)\}$;
                if Next-Support$[(k,z),c] = \phi \wedge M[(k,z),c] = 0$ then
                  begin
                      List:=List $\cup\{[(k,z),c]\}$;
                      $M[(k,z),c] := 1$;
                  end
            end
        if $(k,z) \in$ Next-edge$_k$ then
          Local-Next-Support$(k,c)$:=Local-Next-Support$((),c) - \{(k,z)\}$;
        if Local-Next-Support$(k,c) = \phi$ then
          begin
              $L_k := L_k - \{c\}$;
              for $(k,z) \in$ Local-Prev-Support$(k,c)$ do
                  if $M[(k,z),c] = 0$ then
                    begin
                        List:=List $\cup\{[(k,z),c]\}$;
                        $M[(k,z),c] := 1$;
                    end
          end
        if $(x,k) \in$ Prev-edge$_k$ then
          Local-Prev-Support$(k,c)$:=Local-Prev-Support$(k,c) - \{(k,x)\}$;
        if Local-Prev-Support$(k,c) = \phi$ then
          begin
              $L_k := L_k - \{c\}$;
              for $(k,z) \in$ Local-Next-Support$(k,c)$ do
                  if $M[(k,z),c] = 0$ then
                    begin
                        List:=List $\cup\{[(k,z),c]\}$;
                        $M[(k,z),c] := 1$;
                    end
          end
    end
```

Figure 12: Algorithm to enforce MUSE CSP path consistency.

elements, because of M, appear on the list only once) is $O(n^3l^3)$. Because there are $O(n^2l)$ Prev-Support and Next-Support Lists, each up to $O(n)$ in size, the running time required for line 23 and 32 is $O(n^3l)$. Finally, since there are $O(nl)$ Local-Next-Support; and Local-Prev-Support sets to eliminate $O(n)$ elements from, the running time of lines 40 and 52 is $O(n^2l)$. Hence, the running time of the MUSE CSP path consistency algorithm is $O(n^3P+n^3l+n^2l)$. By comparison, the running time for CSP path consistency is (n^3l^3).

The proof of correctness for MUSE PC-1 is similar to our proof for MUSE AC-1. Induction on the iterations of the while loop is sufficient to demonstrate the correctness of the algorithm in Figure 12. The extension of our algorithm for general k-consistency [1] should now be apparent to the reader.

5 Conclusion

In conclusion, MUSE CSP can be used to efficiently represent and solve several similar instances of the constraint satisfaction problem simultaneously. If multiple instances of a CSP have some common variables which have the same domains and compatible constraints, then they can be combined into a single instance of a MUSE CSP, and much of the work required to enforce node, arc, and path consistency need not be duplicated across the multiple instances.

We have already developed a MUSE CSP constraint-based parser, PARSEC [10, 23, 11, 9], which is capable of parsing word graphs containing multiple sentence hypotheses. We have developed syntactic and semantic constraints for parsing single sentences, which when applied to a word graph, eliminate those hypotheses that are syntactically or semantically incorrect. Speech processing is not the only area where segmenting the signal into higher-level chunks is problematic. Vision systems and handwriting analysis systems have comparable problems. Additionally, Pearl [20] has pointed out how a large number of instances of CSP can be used to compute Dempster's rule of combination [6, 21]. These instances can easily be replaced by a single instance of MUSE CSP.

References

[1] M.C. Cooper. An optimal k-consistency algorithm. Artificial Intelligence, 41:89–95,

1989.

[2] A.L. Davis and A. Rosenfeld. Cooperating processes for low-level vision: A survey. Artificial Intelligence, 17:245–263, 1981.

[3] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. Artificial Intelligence, 34:1–38, 1988.

[4] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. Artificial Intelligence, 34:1–38, 1988.

[5] Rina Dechter. From local to global consistency. Artificial Intelligence, 55:87–107, 1992.

[6] A. P. Dempster. Upper and lower probabilities induced by a multivalued mapping. Annals of Mathematical Statistics, 38:325–339, 1967.

[7] E. Freuder. Partial constraint satisfaction. In Proceedings of the *International* Joint Conference on Artificial Intelligence, pages 278–283, 1989.

[8] E. Freuder. Complexity of K-tree-structured constraint-satisfaction problems. In Proceedings of the Eighth National Conference on Artificial Intelligence, pages 4–9, 1990.

[9] M. P. Harper and R. A. Helzerman. PARSEC: A constraint-based parser for spoken language parsing. Technical Report EE-93-28, Purdue University, School of Electrical Engineering, West Lafayette, IN, 1993.

[10] M.P. Harper, R.A. Helzerman, and C.B. Zoltowski. Constraint parsing: A powerful framework for text-based and spoken language processing. Technical Report EE-91-34, Purdue University, School of Electrical Engineering, West Lafayette, IN, 1991.

[11] M.P. Harper, L.H. Jamieson, C.B. Zoltowski, and R. Helzerman. Semantics and constraint parsing of word graphs. In Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, pages II–63–II–66, April 1992.

[12] V. Kumar. Algorithms for constraint-satisfaction problems: A survey. AI Magazine, 13(1):32–44, 1992.

[13] A.K. Mackworth. Consistency in networks of relations. Artificial Intelligence, 8(1):99–118, 1977.

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

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

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

[17] H. Maruyama. Structural disambiguation with constraint propagation. In The Proceedings of the Annual Meeting of ACL, 1990.

[18] Roger Mohr and Thomas C. Henderson. Arc and path consistency revisited. Artificial Intelligence, 28:225–233, 1986.

[19] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. Information Science, 7:95–132, 1974.

[20] Judea Pearl. A constraint-propagation approach to probabilistic reasoning. In Laveen N. Kanal, John F. Lemmer, and A. Rosenfeld, editors, Uncertainty in Artificial Intelligence, volume 4 : Machine Intelligence and Pattern Recognition. North-Holland, Amsterdam, 1986.

[21] G. Shafer. A Mathematical Theory of Evidence. Princeton'University Press, Princeton, NJ, 1976.

[22] M. Villain and H. Kautz. Constraint-propagation algorithms for temporal reasoning. In Proceedings of the Fifth National Conference on Artificial Intelligence, pages 377–382, 1986.

[23] C.B. Zoltowski, M.P. Harper, L.H. Jamieson, and R. Helzerman. PARSEC: A constraint-based framework for spoken language understanding. In Proceedings of the International Conference on Spoken Language Understanding, October 1992.