Multiagent Constraint Satisfaction with Multiply Sectioned Constraint Networks

Y. Xiang and W. Zhang

University of Guelph, Canada

Abstract. Variables and constraints in problem domains are often distributed. These distributed constraint satisfaction problems (DCSPs) lend themselves to multiagent solutions. Most existing algorithms for DCSPs are extensions of centralized backtracking or iterative improvement with breakout. Their worst case complexity is exponential. On the other hand, directional consistency based algorithms solve centralized CSPs efficiently if primal graph density is bounded. No known multiagent algorithms solve DCSPs with the same efficiency. We propose the first such algorithm and show that it is sound and complete.

1 Introduction

Many practical problems can be solved as constraint satisfaction problems (CSPs). Often, the variables and constraints in the problem domain are naturally distributed, spatially, cognitively, or otherwise. These distributed CSPs (DCSPs) [13] lend themselves naturally to solutions using multiagent systems.

Most existing algorithms for solving DCSPs are extensions of centralized algorithms based on backtracking or iterative improvement with breakout [13, 11, 14, 5, 7, 9, 8]. Their worst case complexity is exponential. Another class of algorithms [12] is based on truth maintenance, e.g., DATMS [4]. The complexity of truth maintenance problem is at least NP-hard [6].

On the other hand, directional consistency based algorithms [2, 3] solve centralized CSPs efficiently if the density of the primal graph (measured by *tree width*) is upper-bounded. To the best of our knowledge, no existing multiagent algorithms solve DCSPs with the same efficiency. In this work, we propose the first such algorithm. We present formally an multiagent representation of DCSPs. We prove soundness and completeness of the algorithm and illustrate with a detailed example. Due to space limitations, however, we omit proofs.

2 Background

CSPs are formally modeled as constraint networks. A *constraint network* (CN) $\mathcal{R}$ is a pair $\mathcal{R} = (V, \Lambda)$. V is an non-empty set of discrete variables, called *domain*. Each variable $v \in V$ has a finite space D_v. The space of a subset $X \subset V$ is the Cartesian product of spaces of variables in X and is denoted by D_X. Each $\underline{x} \in D_X$ is a configuration of X. Λ is an non-empty set of constraints. Each constraint is a relation $R_X \subseteq D_X$, where $X \subset V$ is the *scope* of the constraint. The union of scopes of all constraints covers the domain, i.e., $\cup_{R_X \in \Lambda} X = V$.

A configuration $\underline{x} \in D_X$ satisfies a constraint R_X if $\underline{x} \in R_X$. Otherwise, it *violates* the constraint. The *projection* of configuration $\underline{x}$ to $Y \subset X$ is denoted by $\pi_Y(\underline{x})$ and the projection of relation R_X to $Y \subset X$ is denoted by $\pi_Y(R_X)$. Configuration $\underline{x}$ is *consistent* or *legal* if it satisfies every constraint R_Y such that

$Y \subseteq X$. A *solution* to CN $\mathcal{R}$ is a consistent configuration over V. Formally, the set of all solutions, called the *solution set*, of $\mathcal{R}$ is the relation $\bowtie_{R \in \Lambda} R$, where $\bowtie$ refers to relational operator *natural join*. $\mathcal{R}$ is *consistent* iff $\bowtie_{R \in \Lambda} R \neq \emptyset$.

The dependence structure of $\mathcal{R}$ can be depicted by a *primal graph* $G = (V, E)$, where each node is labeled by a variable $v \in V$ and each link $\langle u, v \rangle \in E$ connects nodes u, v if there exists a constraint $R_X \in \Lambda$ such that $u, v \in X$. $\mathcal{R}$ can be solved through an alternative dependence structure compiled from its primal graph. A *cluster* is a subset of V. A *cluster tree* connects a set of clusters into a tree. Each link, called a *separator*, connects two clusters whose intersection $S \neq \emptyset$, and is labeled by S. A cluster tree is a *junction tree* (JT) if the intersection of each pair of clusters is a subset of every separator on the path between them. Details on how to compile a graph into a JT can be found in [10].

For DCSP, we assume that variables and constraints are distributed among multiple agents such that each agent is in charge of a CN. We introduce concepts for description of primal graphs from multiple CNs to be used later. Let $G_i = (V_i, E_i)$ $(i = 0, 1)$ be two graphs. G_0 and G_1 are *graph-consistent* if subgraphs of G_0 and G_1 spanned by $V_0 \cap V_1$ are identical. Given two graph-consistent graphs $G_i = (V_i, E_i)$ $(i = 0, 1)$, the graph $G = (V_0 \cup V_1, E_0 \cup E_1)$ is the *union* of G_0 and G_1, denoted by $G = G_0 \cup G_1$. Given a graph $G = (V, E)$, a partition of V into V_0 and V_1 such that $V_0 \cup V_1 = V$ and $V_0 \cap V_1 \neq \emptyset$, and subgraphs G_i $(i = 0, 1)$ of G spanned by V_i, G is said to be *sectioned* into G_0 and G_1.

3 Solving CSP With Junction Tree Representation

The method for solving centralized CSPs is attributed to Dechter and Pearl [2, 3, 1]. Our work extends theirs to multiagent systems. We review the method so that its components can be directly referenced later in presenting our extension. Our formulation, however, is not necessarily identical to that in the references.

Given CN $\mathcal{R} = (V, \Lambda)$ and its primal graph G, first, compile G into a JT T. Second, for each constraint R_X in Λ, assign R_X to a cluster Q in T such that $X \subseteq Q$. Third, for each cluster Q in T, replace the set Λ_Q of constraints assigned to it by a single constraint $R_Q = U_Q \bowtie_{R \in \Lambda_Q} R$, where U_Q is a universal relation over Q (containing every configuration of Q). Let Λ' denotes the set of new constraints one per cluster of T. Note that Λ' is simply a grouping of Λ. Finally, let each cluster in T be a variable and its space be configurations in the relation associated with the cluster. For each pair of adjacent clusters Q and C with separator S, impose the *implicit constraint* between Q and C: $\pi_S(\underline{q}) = \pi_S(\underline{c})$, where $\underline{q}$ is a configuration of Q and $\underline{c}$ is a configuration of C. The triple (V, T, Λ') is the *JT representation* of $\mathcal{R}$ and its solution set is $\bowtie_{R \in \Lambda'} R$. Note that Λ' does not include implicit constraints since they simply allows $\bowtie$ operation to be well defined. Note also that (V, T, Λ') is equivalent to a binary CN. Proposition 1 below establishes the key property of the JT representation and plays an important role in analysis of our method.

Proposition 1 *Let (V, Λ) be a CN and (V, T, Λ') be its JT representation. The solution set of (V, Λ) and that of (V, T, Λ') are identical.*

The complexity of the compilation is $O(|\Lambda|\ k^q)$, where k binds the space for variables in V and q binds the size for clusters in T. (V, T, Λ') can be solved based on directional arc-consistency. Given two clusters Q and C with $S = Q \cap C$, configurations $\underline{q}$ and $\underline{c}$ are *consistent* if $\pi_S(\underline{q}) = \pi_S(\underline{c})$. A cluster Q in T is *consistent relative to* an adjacent cluster C if, for each configuration in R_Q, there exists a consistent configuration in R_C. Let Q^* be any cluster in T. Given Q^*, T can be viewed as a tree rooted at Q^* and each two adjacent clusters form a parent-child pair. (V, T, Λ') is *directional arc-consistent* relative to a root cluster Q^* if for every pair of clusters Q and C, where Q is the parent of C, Q is consistent relative to C.

The following object oriented algorithm is activated at each cluster in T by a *caller*, which is either an adjacent cluster or the object T. After it is called in Q^* by T, (V, T, Λ') is directional arc-consistent relative to Q^*.

Algorithm 1 (CollectSeparatorConstraint) *When caller calls in cluster Q, it does the following:*

Q calls CollectSeparatorConstraint in each adjacent cluster C except caller;
for each cluster C (whose separator with Q is S),
    Q receives from C a constraint R_S;
    if $R_S = \emptyset$, Q sends $\emptyset$ to caller and halts;
    Q assigns $R_Q = R_Q \bowtie R_S$;
if caller is a cluster (whose separator with Q is S'), Q sends $\pi_{S'}(R_Q)$ to caller;

The complexity of CollectSeparatorConstraint is $O(t\ l^2)$, where t is the number of clusters in T and l binds the size of relation in each cluster. After CollectSeparatorConstraint is called in Q^*, if $\emptyset$ is returned, the CN is inconsistent. Otherwise, (V, T, Λ') can be solved by T calling the following algorithm in Q^*. It will then be called recursively at each cluster.

Algorithm 2 (DistributeSeparatorSolution) *When caller calls in cluster Q, it does the following:*

if caller is a cluster (whose separator with Q is S),
    Q receives from caller a constraint R_S of a single configuration;
    Q assigns $R_Q = \{\underline{q}\}$, where $\underline{q} \in R_Q \bowtie R_S$;
else Q removes all configurations in R_Q except one;
for each adjacent cluster C (whose separator with Q is S') except caller;
    Q calls DistributeSeparatorSolution in C with $\pi_{S'}(R_Q)$;

After DistributeSeparatorSolution is called in Q^*, the solution to (V, T, Λ') can be obtained by retrieving R_Q from each cluster Q and joining them together.

CollectSeparatorConstraint above only achieves directional arc-consistency. A parent cluster Q (relative to a root) is consistent relative to a child cluster C, but C may not be consistent relative to Q. This is possible because the constraint R_S sent from C to Q during CollectSeparatorConstraint may contain a configuration $\underline{s}$ such that no configuration $\underline{q}$ in R_Q satisfies $\pi_S(\underline{q}) = \underline{s}$. Adjacent clusters Q and C are *consistent* if Q is consistent relative to C and vice versa.

(V, T, Λ') is *full arc-consistent* if every pair of adjacent clusters is consistent. Full full arc-consistency is not needed to solve (V, T, Λ') as shown above. However, it is necessary for solving DCSPs as will be seen.

The following object oriented algorithm can be performed after CollectSeparatorConstraint to make a JT full arc-consistent.

Algorithm 3 (DistributeSeparatorConstraint) *When caller calls in cluster C, it does the following:*

if caller is a cluster (whose separator with C is S),
C receives from caller a constraint R_S;
C assigns $R_C = R_C \bowtie R_S$;
for each adjacent cluster Q (whose separator with C is S') except caller,
C calls DistributeSeparatorConstraint in Q with $\pi_{S'}(R_C)$;

The following algorithm combines CollectSeparatorConstraint and DistributeSeparatorConstraint.

Algorithm 4 (UnifyConstraint) *Choose a cluster Q^* arbitrarily and call CollectSeparatorConstraint in Q^*. If Q^* returns $\emptyset$, return false. Otherwise, call DistributeSeparatorConstraint in Q^* and return true upon completion.*

After UnifyConstraint, a JT is full arc-consistent as summarized below.

Proposition 2 *Let (V, T, Λ') be the JT representation of a CN. The CN is inconsistent iff UnifyConstraint returns false. Otherwise, UnifyConstraint returns true and the JT is full arc-consistent.*

4 Multiply Sectioned Constraint Network

A DCSP involves a large problem domain where variables and constraints are distributed. We solve a DCSP with a multiagent system, where each agent is in charge of a subset of variables and constraints. To ensure that computation is sound and complete as well as efficient, partition of variables and constraints among agents needs to satisfy certain conditions. We model a DCSP as an multiply sectioned constraint network (MSCN) which specifies these conditions formally.

Definition 1 *From a set of CNs $\{\mathcal{R}_i = (V_i, \Lambda_i)\}$ (each called a* `subnet`*), an* `MSCN` *$\mathcal{R}$ is defined as a pair $\mathcal{R} = (V, \Lambda)$, where $V = \bigcup_i V_i$ is the* `domain` *(with each V_i called a* `subdomain`*) and $\Lambda = \bigcup_i \Lambda_i$ is the set of constraints, such that the following holds: (1) A JT exists with $\{V_i\}$ as the set of clusters. (2) For any two subnets $\mathcal{R}_i$ and $\mathcal{R}_j$ such that $V_i \cap V_j \neq \emptyset$, their primal graphs are graph-consistent. The* solution set *of $\mathcal{R}$ is $\bowtie_i (\bowtie_{R \in \Lambda_i} R)$.*

This concise definition has a number of implications: First of all, although there is no mention of agents in the definition, we assume that each subnet $\mathcal{R}_i$ is embodied by a unique agent A_i who is in charge of subdomain V_i. Hence,

a variable shared by two subnets are *public* to the corresponding agents and a variable unique in a subnet is *private*.

Second, domain partition is required to satisfy the connectivity condition (a JT is connected). That is, for any two subdomains V_i and V_j, there exists a sequence of subdomains such that every two adjacent in the sequence share some variables. This restriction implies that each subnet is relevant to the partial solution in each other subnet.

Third, domain partition is required to satisfy the JT condition. Although a natural domain partition may not satisfy this condition, it can be enforced by making limited private variables public. Agents A_i and A_j are said to be *adjacent* if V_i and V_j are adjacent in the JT.

Fourth, primal graphs are required to be graph-consistent. This means that every constraint over public variables in one subnet must be contained in every other subnet that share these variables. We assume that this condition is enforced by communicating any constraint over public variables to other agents in a pre-processing. Similarly, if a constraint R_Z has a scope $Z = X \cup Y$, where $X \cap Y = \emptyset$, X is public, and Y is private, we assume that the constraint $\pi_X(R_Z)$ has been communicated to the other agent. The condition essentially ties variable sharing between subnets with constraint sharing.

Fifth, as each subnet uniquely defines its primal graph and these primal graphs are graph-consistent, the collection of primal graphs from all subnets defines a multiply sectioned primal graph over the domain, and hence the name MSCN.

Sixth, although an MSCN may admit multiple JTs (condition (1)), one of them, referred to as the *hypertree*, is agreed upon by all agents and governs agent communication. That is, if A_i and A_j are adjacent in the hypertree, then they can communicate directly. We refer to each cluster V_i in the hypertree as a *hypernode*, and associate the hypernode with subnet $\mathcal{R}_i$ and agent A_i. Hence, the hypertree is the agent organization. If A_i and A_j are adjacent in the organization, we refer to $V_i \cap V_j$ as their *agent interface*.

Finally, joining a relation multiple times to another relation has the effect of exactly once. Hence, communicating constraints over public variables, as mentioned above, has no impact on the solution set.

Fig. 1 shows a distributed map coloring problem as an example MSCN. The primal graphs of subnets are shown in (b) and the hypertree is shown in (a). The space of each variable contains three colors which we denote simply by $\{0, 1, 2\}$.

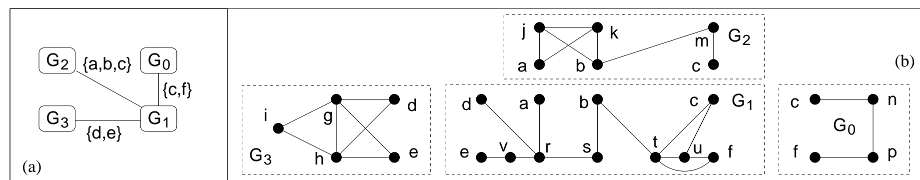

Fig. 1. The hypertree (a) and primal graphs (b) of an MSCN. Each link in (b) represents a $\neq$ constraint.

5 Linked Junction Forest Representation Of MSCN

To extend JT based solution of CNs to MSCNs, we compile MSCNs to a runtime representation. Exploring structural similarity between constraint reasoning and probabilistic reasoning, we adopt key steps in structure compilation of multiply sectioned Bayesian networks (MSBNs)[10]: cooperative triangulation, local JT construction, and linkage tree (LT) construction. Formal specification in the context of MSBNs can be found in the reference. Outcome of structure compilation for the MSCN in Fig. 1 is shown in Fig. 2.

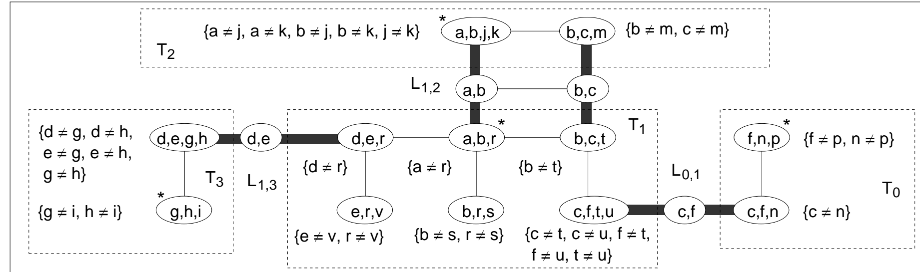

Fig. 2. The linked junction forest compiled from MSCN in Fig. 1. The constraint assigned to each cluster is shown in {}.

Each subnet is compiled into a JT, e.g., subnet G_1 is compiled into T_1. Each agent interface is compiled into a LT, e.g., the agent interface between A_1 and A_2 is compiled into LT $L_{1,2}$ which consists of two clusters. Each cluster in $L_{1,2}$ is referred to as a *linkage*, e.g., $\{b, c\}$. Each linkage has two *host clusters* one in each JT it links. For instance, linkage $\{b, c\}$ has host cluster $\{b, c, t\}$ in T_1 and host cluster $\{b, c, m\}$ in T_2.

After the structure compilation, each agent A_i assigns constraints in Λ_i to clusters in T_i as follows: For each constraint R_X in Λ_i, assign R_X to a cluster Q in T_i such that $X \subseteq Q$. After assignment, for each cluster Q in T_i, A_i replaces the set Λ_Q of constraints assigned to it by a single constraint $R_Q = U_Q \bowtie_{R \in \Lambda_Q} R$, where U_Q is the universal relation over Q.

Let each cluster in T_i be a variable and its space be configurations in the relation associated with the cluster. For each pair of adjacent clusters Q and C with separator S, let the *implicit constraint* between Q and C be $\pi_S(\underline{q}) = \pi_S(\underline{c})$, where $\underline{q}$ is a configuration of Q and $\underline{c}$ is a configuration of C. For instance, constraint between clusters $\{c, f, t, u\}$ and $\{b, c, t\}$ in T_1 requires their configurations to have the same value over c and t. The similar implicit constraint is imposed relative to each linkage S and its two linkage hosts Q and C. For instance, constraint between linkage hosts $\{b, c, t\}$ in T_1 and $\{b, c, m\}$ in T_2 requires their configurations to have the same value over b and c.

Given an MSCN $\mathcal{R} = (V = \bigcup_i V_i, \Lambda = \bigcup_i \Lambda_i)$, the outcome of compilation is a tuple (V, T, L, Λ'), where $T = \{T_i\}$ is a set of JTs each compiled from a subnet of $\mathcal{R}$, and $L = \{L_{i,j}\}$ is a set of LTs one compiled from each pair of adjacent subnets on hypertree. $\Lambda' = \{\Lambda'_i\}$ is a collection of sets. Each Λ'_i is a set of constraints one per cluster of T_i. We refer to (V, T, L, Λ') as the

linked junction forest representation (LJF) of the MSCN. Again, we assume that agents are attached to LJF such that each T_i is embodied by A_i. The solution set of (V, T, L, Λ') is $\bowtie_i (\bowtie_{R \in \Lambda'_i} R)$. Note that Λ' does not include implicit constraints since they simply allow $\bowtie$ operation to be well defined. The following theorem establishes an important property of the LJF. It follows from definitions of solution sets for MSCN and its LJF, as well as the composition of Λ'.

Theorem 1 *Let $\mathcal{R} = (\bigcup_i V_i, \bigcup_i \Lambda_i)$ be an MSCN and $\mathcal{F} = (V, T, L, \Lambda')$ be its LJF representation. Then, $\mathcal{R}$ and $\mathcal{F}$ have the same solution set.*

The compilation computation is dominated by the cooperative triangulation and local JT construction. The complexity is $O(n\ \lambda\ k^q)$, where n is the number of agents, λ bounds $|\Lambda_i|$, k binds the space for variables in V and q binds the size for clusters in JTs in T.

6 Solving MSCN with LJF

To solve MSCN using its LJF, we extend directional arc-consistency to LJF. An agent A_i is *interface-consistent* relative to an adjacent agent A_j if, for each configuration R_i associated with A_i ($R_i \in \ \bowtie_{R \in \Lambda'_i} R$), there exists a consistent configuration associated with A_j. A LJF is *directional interface-consistent* relative to a root agent if, for every two agents A_i and A_j where A_i is the parent of A_j relative to the root, A_i is interface-consistent relative to A_j.

The following two algorithms achieve directional interface-consistency in a LJF. The first below is used by agent A_i to update constraints in its linkage hosts based on constraint message from an adjacent agent A_j.

Algorithm 5 (AbsorbInterfaceConstraint) *When agent A_i performs AbsorbInterfaceConstraint relative to agent A_j with a set $\Omega = \{R_X\}$, where each R_X is a constraint over a linkage X with agent A_j, A_i does the following:*

for each linkage C with A_j with linkage host Q at A_i,
  assign $R_Q = R_Q \bowtie R_C$, where $R_C \in \Omega$;
  if $R_Q = \emptyset$, return false;
return true;

The second algorithm below recursively propagates constraint messages inwards along the hypertree. The agent executing the algorithm is referred to as A_0 with local JT T_0. The execution is activated by a *caller* agent, who is either an adjacent agent, denoted by A_c, or the coordinator. Additional adjacent agents of A_0 are denoted by $A_1, \ldots, A_m$, if any.

Algorithm 6 (CollectInterfaceConstraint) *When caller calls A_0 to CollectInterfaceConstraint, it does the following:*

1 for each agent A_i ($i = 1, \ldots, m$),
2   call CollectInterfaceConstraint on A_i;
3   if A_i returns $\emptyset$, return $\emptyset$;
4   receive $\Omega_i = \{R_C\}$ where R_C is a constraint over a linkage C with A_i;

5 perform AbsorbInterfaceConstraint relative to A_i with Ω_i;
6 if false is returned, return $\emptyset$;
7 perform UnifyConstraint;
8 if false is returned, return $\emptyset$;
9 if A_c is an adjacent agent,
10 initialize $\Omega_c = \emptyset$;
11 for each linkage S with A_c of linkage host Q at A_0,
12 compute $R_S = \pi_S(R_Q)$;
13 if $R_S = \emptyset$, return $\emptyset$;
14 else add R_S to Ω_c;
15 return Ω_c to A_c;
16 else return a special set ∇ to coordinator signifying successful completion;

Theorem 2 shows the consistency properties achieved by the above algorithm.

Theorem 2 *Let $\mathcal{F} = (V, T, L, \Lambda')$ be the LJF representation of an MSCN populated by agents and CollectInterfaceConstraint is called on any agent A_0.*

$\mathcal{F}$ is inconsistent iff A_0 returns $\emptyset$. Otherwise, A_0 returns ∇ and the following holds:

1. *$\mathcal{F}$ is directional interface-consistent relative to A_0.*
2. *Each T_i is full arc-consistent.*
3. *Each linkage tree L_i is full arc-consistent.*

The following algorithm generates a (partial) solution for a subdomain constrained by a partial solution over the interface with the calling agent.

Algorithm 7 (GetLocalSolution) *When agent A_0 performs GetLocalSolution with $\Omega = \{R_X\}$, where each R_X is a singleton constraint (consisting of a single configuration) over a linkage X with agent A_c, it does the following:*

if $\Omega = \emptyset$, call DistributeSeparatorSolution in any cluster in T_0;
else
for each linkage C with A_c (whose host cluster is Q),
assign $R_Q = R_Q \bowtie R_C$, where $R_C \in \Omega$;
call DistributeSeparatorSolution in the host of any linkage with A_c;

Note that after the assignment, R_Q is not necessarily a singleton. After DistributeSeparatorSolution is called, it is so. The following recursive algorithm propagates partial solutions over agent interfaces along the hypertree.

Algorithm 8 (DistributeSolution) *When caller calls A_0 to DistributeSolution, it does the following:*

1 if caller is an adjacent agent,
2 receive $\Omega = \{R_X\}$ where each R_X is a singleton constraint over linkage X with caller;
3 perform GetLocalSolution with Ω;
4 else perform GetLocalSolution with $\emptyset$;

5 for each agent A_i $(i = 1, \ldots, m)$,
6 initialize $\Omega' = \emptyset$;
7 for each linkage C with A_i (whose host cluster is Q), add $\pi_C(R_Q)$ to Ω';
8 call DistributeSolution on A_i with Ω';

The following algorithm is executed by the system coordinator.

Algorithm 9 (SolveDCSP) *Choose an agent A^* arbitrarily. Call CollectInterfaceConstraint in A^*. If A^* returns $\emptyset$, return failure. Otherwise, call DistributeSolution in A^*.*

Theorem 3 below establishes that SolveDCSP is sound and complete.

Theorem 3 *Let $\mathcal{F} = (V, T, L, \Lambda')$ be a LJF of an MSCN and SolveDCSP be executed. Then failure will be returned iff $\mathcal{F}$ is inconsistent. Otherwise, $R' = \bowtie_i (\bowtie_{Q \in T_i} R_Q)$ is a singleton such that $R' \subseteq R$, where R is the solution set of $\mathcal{F}$.*

Let n be the number of agents, t the maximum number of clusters in a local JT, q the maximum size of clusters, and k bind the space for variables in V. After CollectInterfaceConstraint completes, SolveDCSP is backtracking free. Hence, computation is dominated by UnifyConstraint during CollectInterfaceConstraint. UnifyConstraint has no more than twice the amount of computation of CollectSeparatorConstraint, whose complexity is $O(t\ l^2)$ (Section 3), where l binds the size of relation in each cluster. Instead, we use a conservative complexity estimation, $O(t\ k^{2q})$, by replacing l with k^q. Therefore, the complexity of SolveDCSP is $O(n\ t\ k^{2q})$. When q is upper bounded, SolveDCSP is efficient. Note that q characterizes the density of an MSCN and is equivalent to the tree width of a centralized CN.

Another advantage of our method is that private variables in each agent and constraints over them are kept private during compilation and solution.

7 Example

We illustrate solution process for the example MSCN. Its compiled LJF is shown in Fig. 2. Initial constraints for clusters are listed in Table 1, where relations of the 'same' set of configurations are listed only once. For instance, relations over clusters $\{g, h, i\}$ and $\{b, c, m\}$ are shown in the middle and will be referred to as R_2 over $\{g, h, i\}$ and R_2 over $\{b, c, m\}$, respectively.

Suppose SolveDCSP is executed with $A^* = A_0$. Then, CollectInterfaceConstraint is called in A_0. In turn, A_0 calls CollectInterfaceConstraint in A_1, which calls CollectInterfaceConstraint in A_2 and A_3.

A_3 performs UnifyConstraint by calling CollectSeparatorConstraint in cluster, say, $\{g, h, i\}$, which in turn calls CollectSeparatorConstraint in cluster $\{d, e, g, h\}$. In response, $\{d, e, g, h\}$ sends relation R_4 (Table 2) over $\{g, h\}$ to $\{g, h, i\}$, which causes modification of the constraint at $\{g, h, i\}$ to R_5 (Table 2).

Next, A_3 calls DistributeSeparatorConstraint in $\{g, h, i\}$, which in turn calls DistributeSeparatorConstraint in $\{d, e, g, h\}$ with R_4 (Table 2). This results in no change in the constraint at $\{d, e, g, h\}$. UnifyConstraint at A_3 returns with

Table 1. Initial constraints associated with clusters. A single line separates variables of relations with identical set of configurations which are enclosed by double lines.

R_1	
d e g h	1 1 0 2
a b j k	1 1 2 0
c f t u	2 2 0 1
0 0 1 2	2 2 1 0

(The first row of R_1's configuration column is 0 0 2 1.)

R_2	f n p	1 0 2	2 1 0
g h i	0 0 1	1 1 0	2 2 0
b c m	0 0 2	1 1 2	2 2 1
e r v	0 1 2	1 2 0	
b r s	0 2 1	2 0 1	

R_3	0 0 1	0 2 2	1 2 0	2 1 1
d e r	0 0 2	1 0 0	1 2 2	2 2 0
a b r	0 1 1	1 0 2	2 0 0	2 2 1
b c t	0 1 2	1 1 0	2 0 1	
c f n	0 2 1	1 1 2	2 1 0	

Table 2. Constraints as messages between clusters or newly assigned to clusters.

R_4	g h	2 0
b r	0 1	2 1
c t	0 2	
e r	1 0	
f n	1 2	

R_5	0 1 2	2 1 0
b r s	0 2 1	
e r v	1 0 2	
f n p	1 2 0	
g h i	2 0 1	

R_6	1 1
a b	2 2
c f	
d e	
0 0	

R_7	0 2 1	2 0 1
b c t	1 0 2	2 1 0
0 0 1	1 1 0	2 2 0
0 0 2	1 1 2	2 2 1
0 1 2	1 2 0	

R_8	0 0 2
a b r	1 1 0
c f n	1 1 2
d e r	2 2 0
0 0 1	2 2 1

true. T_3 is full arc-consistent with cluster constraints: R_1 (Table 1) for $\{d, e, g, h\}$ and R_5 (Table 2) for $\{g, h, i\}$. Before completing CollectInterfaceConstraint, A_3 sends A_1 a message containing constraint R_6 (Table 2) over linkage $\{d, e\}$.

At the same time, A_2 also performs UnifyConstraint by calling CollectSeparatorConstraint in cluster, say, $\{a, b, j, k\}$, followed by calling DistributeSeparatorConstraint in $\{a, b, j, k\}$. During CollectSeparatorConstraint, the message from $\{b, c, m\}$ to $\{a, b, j, k\}$ is a universal relation over $\{b\}$, which causes no change in $\{a, b, j, k\}$. During DistributeSeparatorConstraint, the message from $\{a, b, j, k\}$ to $\{b, c, m\}$ is the same universal relation that causes no change in $\{b, c, m\}$. UnifyConstraint at A_2 returns with true and T_2 is full arc-consistent. Before completing CollectInterfaceConstraint, A_2 sends A_1 a message containing two constraints with one over each linkage. The constraint over $\{a, b\}$ is R_6 (Table 2) and that over $\{b, c\}$ is universal.

After A_1 receives the message from A_3, it calls AbsorbInterfaceConstraint, which causes the constraint at linkage host $\{d, e, r\}$ to be modified into the relation R_8 (Table 2). Similarly, after receiving the message from A_2, A_1 calls AbsorbInterfaceConstraint. It modifies the constraint at linkage host $\{a, b, r\}$ into the relation R_8 (Table 2) but constraint at linkage host $\{b, c, t\}$ remains as R_3 (Table 1).

Subsequently, A_1 performs UnifyConstraint by calling CollectSeparatorConstraint in cluster, say, $\{a, b, r\}$, followed by calling DistributeSeparatorConstraint. During CollectSeparatorConstraint, the message sent from $\{e, r, v\}$ to $\{d, e, r\}$ is a universal relation over $\{e, r\}$ and hence causes no change to the constraint at $\{d, e, r\}$. The message sent from $\{d, e, r\}$ to $\{a, b, r\}$ is a universal relation over $\{r\}$ and hence causes no change to the constraint at $\{a, b, r\}$. The message from $\{b, r, s\}$ to $\{a, b, r\}$ is a universal relation over $\{b, r\}$ and causes no change to the constraint at $\{a, b, r\}$. The message from $\{c, f, t, u\}$ to $\{b, c, t\}$ is R_4 (Table 2) over $\{c, t\}$ and changes the constraint at $\{b, c, t\}$ to R_7 (Table 2). The message from $\{b, c, t\}$ to $\{a, b, r\}$ is universal over $\{b\}$ and causes no change to constraint at $\{a, b, r\}$.

During DistributeSeparatorConstraint, the message from $\{a, b, r\}$ to $\{d, e, r\}$ is a universal relation over $\{r\}$ and causes no change to the constraint at $\{d, e, r\}$. The message from $\{d, e, r\}$ to $\{e, r, v\}$ is R_4 (Table 2) over $\{e, r\}$ and it modifies the constraint at $\{e, r, v\}$ to R_5 (Table 2). The message from $\{a, b, r\}$ to $\{b, r, s\}$ is R_4 (Table 2) over $\{b, r\}$ and modifies the constraint at $\{b, r, s\}$ to R_5 (Table 2). The message from $\{a, b, r\}$ to $\{b, c, t\}$ is a universal relation over $\{b\}$ and causes no change to the constraint at $\{b, c, t\}$. The message from $\{b, c, t\}$ to $\{c, f, t, u\}$ is R_4 (Table 2) over $\{c, t\}$ and causes no change to the constraint at $\{c, f, t, u\}$. UnifyConstraint at A_1 returns with true. T_1 is full arc-consistent with the following cluster constraints: R_1 (Table 1) for $\{c, f, t, u\}$, R_7 (Table 2) for $\{b, c, t\}$, R_8 (Table 2) for $\{d, e, r\}$ and $\{a, b, r\}$, R_5 (Table 2) for $\{e, r, v\}$ and $\{b, r, s\}$. Before completing CollectInterfaceConstraint, A_1 sends A_0 a message containing constraint R_6 (Table 2) over linkage $\{c, f\}$.

After A_0 receives the message, it calls AbsorbInterfaceConstraint which replaces the constraint at linkage host $\{c, f, n\}$ by R_8 (Table 2). Afterwards, A_0 performs UnifyConstraint by calling CollectSeparatorConstraint in cluster, say, $\{f, n, p\}$, followed by calling DistributeSeparatorConstraint. During CollectSeparatorConstraint, the message from $\{c, f, n\}$ to $\{f, n, p\}$ is R_4 (Table 2) over $\{f, n\}$. It modifies the constraint at $\{f, n, p\}$ into R_5 (Table 2). During DistributeSeparatorConstraint, the message from $\{f, n, p\}$ to $\{c, f, n\}$ is R_4 (Table 2) over $\{f, n\}$ and has no effect at $\{c, f, n\}$. UnifyConstraint at A_0 returns with true. T_0 is full arc-consistent with the following cluster constraints: R_8 (Table 2) for $\{c, f, n\}$ and R_5 (Table 2) for $\{f, n, p\}$. As the result, A_0 terminates CollectInterfaceConstraint and returns ∇.

Subsequently, A_0 is called to DistributeSolution. It runs GetLocalSolution by first calling DistributeSeparatorSolution at, say, $\{f, n, p\}$. This produces the partial solution R_{11} for $\{f, n, p\}$ first and then R_{10} (Table 3) for $\{c, f, n\}$ at T_0.

Table 3. Relations generated during DistributeSolution.

R_9
c f t u
2 2 0 1
2 2 1 0

R_{10}	
a b r	c f n
b c m	d e r
b c t	2 2 1

R_{11}	
b r s	2 1 0
e r v	
f n p	

R_{12}	
a b	d e
b c	2 2
c f	

R_{13}	
a b j k	2 2 1 0
c f t u	
d e g h	

R_{14}
g h i
1 0 2

Next, A_0 calls A_1 to DistributeSolution with the message containing the relation R_{12} (Table 3) over $\{c, f\}$. In response, A_1 modifies its constraint in linkage host $\{c, f, t, u\}$ to R_9. It then calls DistributeSeparatorSolution in the host $\{c, f, t, u\}$. The resultant partial solution at each cluster of T_1 are as follows: R_{13} over $\{c, f, t, u\}$, R_{10} over $\{b, c, t\}$, $\{a, b, r\}$, R_{11} over $\{b, r, s\}$, R_{10} over $\{d, e, r\}$, and R_{11} over $\{e, r, v\}$.

After that, A_1 calls A_2 to DistributeSolution with the message containing relations R_{12} over $\{a, b\}$ and $\{b, c\}$. In response, A_2 generates partial solutions R_{13} (Table 3) over $\{a, b, j, k\}$ and R_{10} over $\{b, c, m\}$ at T_2.

Similarly, A_1 calls A_3 to DistributeSolution with the message containing relation R_{12} over $\{d, e\}$. In response, A_3 generates partial solutions R_{13} over

$\{d, e, g, h\}$ and R_{14} over $\{g, h, i\}$ at T_3. SolveDCSP now terminates successfully and the natural join of the above partial solutions in all agents is the solution.

8 Conclusion

In this contribution, we proposed a representation of DCSPs as MSCNs, extended techniques for MSBNs [10] to compilation of MSCNs into runtime LJFs, and presented the first algorithm suite that solves efficiently DCSPs of bounded primal graph density. The algorithm suite is shown to be sound and complete. Therefore, we have shown that MSCNs form a tractable class of DCSPs. Experimental study on distributed scheduling is underway.

Acknowledgement

Financial support to the first author through Disvoery Grant from NSERC, Canada is acknowledged.

References

1. R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.
2. R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.
3. R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38(3):353–366, 1989.
4. C.L. Mason and R.R. Johnson. DATMS: a framework for distributed assumption based reasoning. In L. Gasser and M.N. Huhns, editors, *Distributed Artificial Intelligence II*, pages 293–317. Pitman, 1989.
5. P. Meseguer and M. Jimenez. Distributed forward checking. In *Proc. CP'2000 Distributed Constraint Satisfaction Workshop*, 2000.
6. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.
7. M.C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with aggregations. In *Proc. AAAI'2000*, pages 917–922, 2000.
8. M.C. Silaghi, D. Sam-Haroud, and B. Faltings. ABT with asynchronous reordering. In *Proc. Inter. Conf. on Intelligent Agent Technology*, pages 54–63, 2001.
9. M.C. Silaghi, D. Sam-Haroud, and B. Faltings. Consistency maintenance for ABT. In *Principles and Practice of Constraint Programming - Proc. CP2001, LNCS 2239*, pages 271–285. Springer-Verlag, Berlin, 2001.
10. Y. Xiang. *Probabilistic Reasoning in Multi-Agent Systems: A Graphical Models Approach*. Cambridge University Press, Cambridge, UK, 2002.
11. M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Principles and Practice of Constraint Programming - CP95, Lecture Notes in Computer Science, Vol.976*, pages 88–102. Springer-Verlag, 1995.
12. M. Yokoo. *Distributed Constraint Satisfaction*. Springer, 2001.
13. M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *Proc. 12th IEEE Inter. Conf. on Distributed Computing Systems*, pages 614–621, 1992.
14. M. Yokoo and K. Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In *Proc. 2nd Inter. Conf. on Multi-Agent Systems*, pages 401–408, 1996.