Tractable Optimal Multiagent Collaborative Design

Yang Xiang, University of Guelph, Canada

Abstract

Optimal design is intractable in general. We identify a tractable class of design problems and propose the first framework for efficient, decision-theoretically optimal, collaborative design.

1 Introduction

Collaborative design networks (CDNs) [6] provide decision-theoretic graphical models for design in supply chains. Based on CDNs, a multiagent coordination protocol [7] enables an exponential complexity reduction in optimal design. Since local design at individual agents is assumed exhaustive, the overall complexity of the protocol is still intractable. We extend the single-agent division tree [5], and propose a multiagent design-time representation and an algorithm suite that allow optimal and efficient collaborative design.

2 Background

An agent for the design of a component encodes design knowledge and preference into a design network. A *design network* is a triple $S = (V, G, P)$. The *domain* is a set of discrete variables $V = D \cup T \cup M \cup U$, where D, T, M, U are disjoint. D is an non-empty set of *design parameters*. T is a set of *environmental factors* representing uncertain lifetime conditions of the product under design. M is an non-empty set of objective *performance measures* of the product. U is an non-empty set of subjective *utility functions* expressing the preference of the agent's principal. $G = (V, E)$ is a DAG and encodes dependence relations in V. P is a set of potentials, one per node. A potential either quantifies the uncertain dependency or the preference. From P, the expected utility of each design is well defined. However, computing the optimal design is intractable in general.

Division tree is a data structure for tractable optimal design [5]. A *division tree* has a top level junction tree (JT), called *association tree*, where each separator is made of design parameters *only* and each cluster is called a *division*. Variables in each division are organized into a nested JT, termed *division subJT*.

Knowledge representation of a multiagent design system can be specified as a collaborative design network (CDN).

Definition 1 *From a set of design subnet $\{S_i = (V_i, G_i, P_i)\}$, a CDN S is defined as a tuple (V, G, P, W). $V = \bigcup_i V_i$ is the* `domain` *where each V_i is a* `subdomain`*. $G = \bigcup_i G_i$ is a DAG* `structure` *and a hypertree over G exists. Each interface $V_k \cap V_m$ on the hypertree is a subset $D_k \cap D_m$. Let x be a variable and $\pi(x)$ its parents in G. P is the set of* `potentials` *one for each variable in V in the form $P(x|\pi(x))$. If x is contained in V_i only, $P(x|\pi(x))$ is identical to its occurrence in P_i. Otherwise, $P(x|\pi(x))$ is identical to its occurrence in a P_j such that G_j contains $\{x\} \cup \pi(x)$. W is a set $\{w_i\}$ of* `weights` *one per subnet and $\sum_i w_i = 1$.*

Weights for subnets express how preference from multiple stakeholders should be compromised. From P, the expected utility of each design is well defined and so is the optimal design. Figure 1 shows a simple CDN.

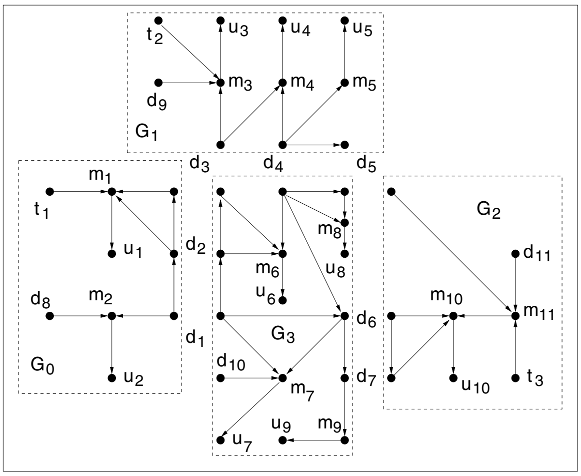

Figure 1. A CDN whose hypertree is a star with center G_3. Letter in each node label indicates its type.

3 Design Subnet Compilation

A CDN is syntactically an MSBN [4]. Compilation of a CDN to design time representation shares initial steps with compilation of an MSBN to its run-time representation (for multiagent probabilistic inference). For completeness, we review these common steps in the next paragraph. Details and formal analysis can be found from reference.

First, agents cooperate to triangulate dependence structures of design subnets into chordal graphs. Each chordal graph is then converted into a local JT. For each adjacent agent on the hypertree, an agent derives, from its local JT, another JT that contains only variables in the agent interface. The derived JT is called a *linkage tree* (LT). Given local JT T and interface I with another agent, the LT is derived by repeating following procedure in T until no action is possible: (1) Remove $x \notin I$ if x is contained in a unique cluster C. (2) After removal, if C becomes a subset of an adjacent cluster D, merge C into D. Each cluster in the LT is called a *linkage*. A cluster in T that contains a linkage (breaking ties arbitrarily) is its *linkage host*. Figure 2 shows

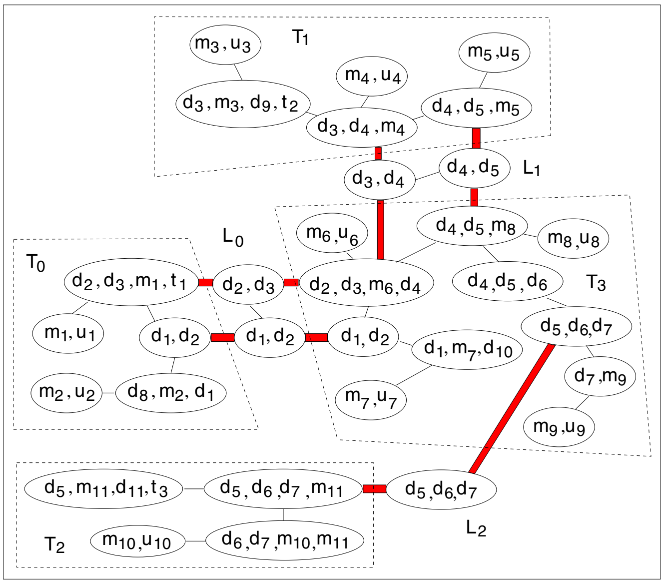

Figure 2. Local JTs (each bounded by a dashed box) compiled from CDN in Figure 1 and LTs derived (outside dashed boxes). A linkage host is connected to the linkage by a thick link.

local JTs and LTs compiled from CDN in Figure 1. The remainder of this section is novel to CDN compilation.

As with non-DT-based MA, collaborative design involves rooted inter-agent message passing along hypertree, during which communication between a pair of adjacent agents is always initiated by agent closer to the root, which we refer to as *caller* agent. Once root agent is determined, each agent has a unique caller agent, except the root. To fully explore the efficiency gain due to division-tree based local design, each agent compiles its local JT into multiple division trees one for each linkage with the caller agent as follows:

Let T be the local JT described above and L be the LT derived from T. Because of the above procedure used to compile L from T, separators of L are also separators in T. For instance, in Figure 2, L_0 is the LT derived from T_3 (also from T_0), L_0 has a separator $\{d_2\}$. It is also a separator in T_3. Deleting such separators from T splits T into subtrees. These subtrees map one-to-one into linkages in L and each subtree contains all variables of the corresponding linkage. For instance, deleting separator $\{d_2\}$ from T_3 (see Figure 2) splits it into two. One of them has three clusters, maps to linkage $\{d_1, d_2\}$ in L_0, and contains the two variables. Another subtree has eight clusters, maps to linkage $\{d_2, d_3\}$, and contains the two variables.

Since each subtree is itself a JT, by identifying its separators that are made of design parameters only, its divisions, association tree and division subJTs can be defined. The resultant is a division tree. For example, the above mentioned three-cluster subtree can be compiled into division tree DT_{31} shown in Figure 3. It consists of two divisions. One of them has a division subJT with two clusters $\{m_7, u_7\}$ and $\{d_1, m_7, d_{10}\}$. The other division has a degenerated division subJT with a single cluster $\{d_1, d_2\}$. Repeating the process for each subtree, multiple division trees are defined with one corresponding to each linkage. The collection of these division trees are called a *division forest* relative to L.

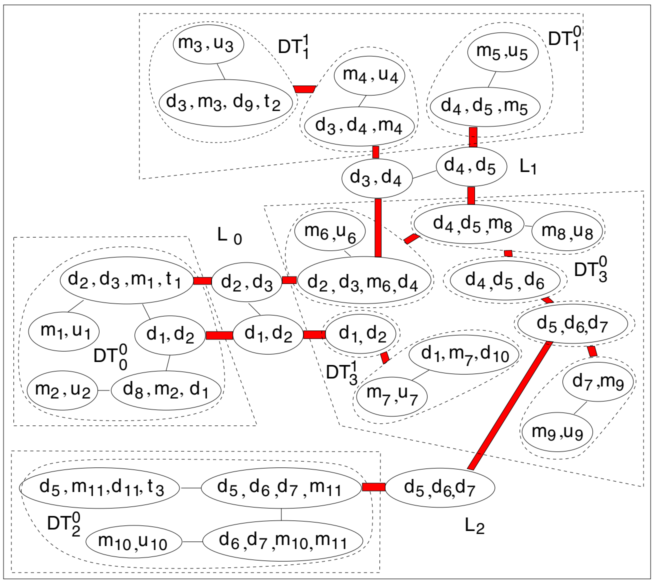

Figure 3. Division forests from CDN in Figure 1. Each division is indicated by a closed dashed spline. Each division separator is indicated by a thick link.

We refer to the above compilation of local JT T to a division forest as operation $MakeDivisionForest$ relative to L. Figure 3 illustrates the result, where the jth division tree in agent A_i is labeled DT_i^j. For A_1, division forest from $MakeDivisionForest$ relative to L_1 consists of two division trees DT_1^0 and DT_1^1 since L_1 has two linkages, where DT_1^0 has a single division but DT_1^1 has two divisions. For A_2, the division forest from $MakeDivisionForest$ relative to L_2 consists of a single division tree since L_2 has a single linkage. For A_3, the division forest from $MakeDivisionForest$ relative to L_0 consists of two division trees. DT_3^0 has five divisions and DT_3^1 has two divisions.

An agent may be adjacent to multiple other agents on the hypertree, e.g., A_3. It is sufficient for each agent to perform $MakeDivisionForest$ relative to its caller agent. Since root agent has no caller (e.g., A_0 in Figure 3), it performs $MakeDivisionForest$ without a given L. The resultant division forest has a single division tree DT_0^0.

Formally, we define $MakeDivisionForest$ as a recursive operation. Without losing generality, we denote agent executing the algorithm as A_0 with local JT T_0. Execution is activated by a caller, denoted as A_c, which is either an adjacent agent of A_0 or the system coordinator[1]. Note that coordinator activates root agent. The linkage tree between A_c (if an agent) and A_0 is denoted as L_c. If A_0 has additional adjacent agents, they are denoted as $A_1, A_2, ...$ and their interface with A_0 are denoted as $L_1, L_2, ...$, respectively.

Algorithm 1 (MakeDivisionForest) *When A_0 is called by A_c, it does the following:*

1. *If A_c is coordinator, A_0 compiles its local JT into a division forest with a single division tree. Otherwise (A_c is adjacent agent), A_0 compiles its local JT into a general division forest relative to L_c.*
2. *A_0 calls each adjacent agent, except A_c, to MakeDivisionForest.*

We refer to, collectively, the set of division forests (one per agent) and the set of linkages (one for each pair of adjacent agents) obtained through the above compilation as a *linked division forest*. Theorem 2 states that the linked division forest is well defined. We omit proof due to space for all formal results.

Theorem 2 *Let T be a JT compiled from a design subnet and L be a LT derived from T. Then a division forest isomorphic with L is well defined (an one-to-one mapping exists between division trees and linkages and between deleted separators in T and linkage separators).*

We note, without formal proof, that a linked division forest is an I-map under h-separation, if the CDN is an I-map under d-separation.

4 Collaborative Design

Next, we present the multiagent algorithm suite for optimal collaborative design using linked division forest. Collaborative design starts by local design. It is performed by each agent, at each division in each of its division trees. This is achieved by system coordinator calling DesignByDivision on any agent A.

Algorithm 2 (DesignByDivision) *When agent A_0 is called by A_c, it does the following:*

for each adjacent agent A_i except A_c,
call A_i to DesignByDivision;
for each division tree DT in A_0's division forest,
for each division Div in DT,
for each configuration $\mathbf{d}$ of the set D' of design parameters in Div,
enter $\mathbf{d}$ into the subJT in Div;
perform belief updating in the subJT;
compute $EU(\mathbf{d}) = \sum_i k_i \, (\sum_{\mathbf{m}} u_i(\mathbf{m}) \, P(\mathbf{m}|\mathbf{d}))$, where i indexes utility nodes in Div, $\mathbf{m}$ is a configuration of parents of u_i in G_0, and k_i is weight associated with u_i;

[1]The system coordinator may well be an elected agent.

As the result of DesignByDivision, for each agent, in each of its division trees, at each division, a utility distribution $EU(D')$ is obtained over partial designs of the division. Subsequently, agents collaborate to update these distributions through CollectUtility, which is called by system coordinator on agent A. The algorithm uses CollectDivisionUtility described in [5].

Algorithm 3 (CollectUtility) *When agent A_0 is called by A_c to CollectUtility, it does the following:*

if A_0 is not a leaf agent on the hypertree,
for each adjacent agent A_i except A_c,
call A_i to CollectUtility;
for each linkage Q_i with A_i,
receive from A_i utility distribution $MEU(Q_i)$;
find division tree DT in A_0's division forest that contains variables in Q_i;
find division Div in DT that contains variables in Q_i;
add adjacent division Div' to Div in DT;
set subJT of Div' with a single cluster Q_i;
associate $MEU(Q_i)$ with Div';
if A_c is an adjacent agent,
call CollectDivisionUtility at the host division Div of each linkage Q_c with A_c;
retrieve utility distribution $EU'(D')$ from Div, where D' is the set of design parameters in Div;
send $MEU(Q_c) = \max_{D' \setminus Q_c} EU'(D')$ to A_c;
else
call CollectDivisionUtility at any division Div_0 in A_0's unique division tree;

We illustrate using linked division forest in Figure 3 where CollectUtility is first called on A_0. A_0 executes first *if* section and calls CollectUtility on A_3. In turn, A_3 calls CollectUtility on A_1 and A_2.

In response, A_1 executes second *if* section, calls CollectDivisionUtility at the two divisions containing linkage hosts, computes the maximum expected utility distributions $MEU(d_3, d_4)$ and $MEU(d_4, d_5)$, and sends them to A_3. The similar is performed by A_2.

After receiving $MEU(d_3, d_4)$ from A_1, A_3 continues in first *if* section, identifies the division in division tree DT_3^0 that contains $\{d_3, d_4\}$, adds a new division $\{d_3, d_4\}$ adjacent to it, and associate $MEU(d_3, d_4)$ with the new division. The similar will be performed by A_3 relative to $MEU(d_4, d_5)$ received from A_1 and $MEU(d_5, d_6, d_7)$ from A_2. Subsequently, A_3 executes second *if* section and eventually sends $MEU(d_1, d_2)$ and $MEU(d_2, d_3)$ to A_0.

After receiving the message from A_3, A_0 continues in first *if* section and eventually performs *else* section.

After CollectUtility terminates, agents collaborate to determine the optimal design through DistributeOptimalDesign which is called by system coordinator on agent A. In DistributeOptimalDesign below, Div_0 refers to the same division in CollectDivisionUtility. The algorithm uses DistributeOptimalDivisionDesign as described in [5].

Algorithm 4 (DistributeOptimalDesign) *When agent A_0 is called by A_c, it does the following:*

if A_c is coordinator,
call DistributeOptimalDivisionDesign at division Div_0 in A_0's unique division tree;
else
for each linkage Q_c with A_c,
receive partial design $\mathbf{q}_c$ for Q_c;
call DistributeOptimalDivisionDesign at host division of Q_c in corresponding division tree relative to $\mathbf{q}_c$;
if A_0 is not a leaf agent on the hypertree,
for each adjacent agent A_i except A_c,
for each linkage Q_i with A_i,
find division tree DT in A_0's division forest that contains variables in Q_i;
find division Div in DT that contains variables in Q_i;
project the optimal partial design of Div to Q_i and denote by $\mathbf{q}_i$;
send $\mathbf{q}_i$ to A_i;
assemble design $\mathbf{d}_0^*$ over design parameters in A_0 from the optimal partial design at each division in each division tree;

For the example in Figure 3, DistributeOptimalDesign is called in A_0. A_0 executes first *if* section to determine its globally optimal local design. It is followed by second *if* section, which propagates the globally optimal partial designs over two linkages to A_3. Finally, A_0 assembles $\mathbf{d}_0^*$.

Next, A_3 executes *else* section. It computes its own globally optimal local design and uses the message from A_0 to constrain the computation. It then executes rest of the algorithm similar to A_0. When A_1 and A_2 receive messages from A_3, they only execute *else* section and assemble their $\mathbf{d}_0^*$.

The following algorithm combines the above algorithms and is executed by system coordinator. Its optimality is established in Theorem 3.

Algorithm 5 (CollaborativeDesign) *Select agent A arbitrarily. Call DesignByDivision in A. Call CollectUtility in A. Call DistributeOptimalDesign in A.*

Theorem 3 *After CollaborativeDesign terminates, the overall design defined by local design $\mathbf{d}^*$ at each agent is optimal.*

The following proposition shows that CollaborativeDesign is efficient when δ is upper-bounded.

Proposition 4 *The complexity of CollaborativeDesign is $O(g\ \theta\ \kappa^\delta)$, where g is the number of agents, θ the maximum number of divisions in an agent, δ the maximum number of design parameters per division, and κ the maximum number of possible values of a design parameter.*

5 Conclusion

The main contribution is the first general framework that allows efficient, decision-theoretically optimal, multiagent, collaborative design, where the domain is represented as a CDN. In doing so, we have identified a tractable class of design problems, namely, those expressible by sparse CDNs. Experimental result on designing customized PCs with CDNs is presented in [1]. The key enabling property of this class is CI rendered by design parameters through agent interfaces and division separators. Hence, the framework formally justifies a corresponding guideline for decomposing a product into components to facilitate collaborative design.

The following work are most closely related. In MAIDs, e.g., [2], an agent encodes knowledge on other agents with an influence diagram. Distributed algorithms based on backtracking or iterative improvement have been proposed to solve DCSPs, e.g., [8]. DCSP is generalized in DCOP and has been solved by algorithms such as ADOPT [3]. In our proposed framework, agents cooperate more closely than in MAIDs. Unlike DCSP, it addresses uncertainty in product life-cycle while satisfying design constraints. Unlike ADOPT, it computes the optimal design decision-theoretically.

Acknowledgements

Financial support from NSERC, Canada is acknowledged.

References

[1] J. Chen. Collaborative design in supply chains: Representation and optimization. Master's thesis, Univ. of Guelph, 2004.

[2] S. Maes, K. Tuyls, and B. Manderick. Modeling a multi-agent environment combining influence diagrams. In *Proc. Inter. Conf. on Intelligent Agents, Web Technology and Internet Commerce*, pages 379–384, 2001.

[3] P. Modi, W. Shen, M. Tambe, and M. Yokoo. An asynchronous complete method for distributed constraint optimization. In *Proc. 2nd Inter. Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS 03)*, pages 161–168, Melbourne, Australia, 2003. ACM Press.

[4] Y. Xiang. *Probabilistic Reasoning in Multi-Agent Systems: A Graphical Models Approach*. Cambridge University Press, Cambridge, UK, 2002.

[5] Y. Xiang. Optimal design with design networks. In *Procs. 3rd European Workshop on Probabilistic Graphical Models*, pages 309–316, Prague, Czech, 2006.

[6] Y. Xiang, J. Chen, and A. Deshmukh. A decision-theoretic graphical model for collaborative design on supply chains. In A. Tawfik and S. Goodwin, editors, *Advances in Artificial Intelligence, LNAI 3060*, pages 355–369. Springer, 2004.

[7] Y. Xiang, J. Chen, and W. Havens. Optimal design in collaborative design network. In *Proc. 4th Inter. Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS'05)*, pages 241–248, 2005.

[8] M. Yokoo. *Distributed Constraint Satisfaction*. Springer, 2001.