Optimal Design with Design Networks

Y. Xiang
University of Guelph, Canada

Abstract

As part of decision-theoretic collaborative design, this work addresses representation and computation issues for a set-based design agent. We refine the definition of design networks to be more expressive for design knowledge and more effective for guiding model construction and verification. We propose a compiled structure of design networks and a suite of algorithms that computes optimal designs efficiently. The result provides a mechanism that combines probabilistic, constraint-based, and decision-theoretic reasoning for single-agent optimal design, and fills in a gap in optimal collaborative design.

1 Introduction

This work concerns optimal decision in centralized and collaborative design. At the centralized front, it deals with set-based design (Ward, 1989; Sobek et al., 1999) which, unlike point-based design often confined to local optimals, evaluates all alternative designs to seek the global optimal. Primary challenges for product lifecycle management include how to efficiently propagate constraints and search in a design space (Paredis et al., 2006). This work contributes to meeting these challenges with an algorithm suite that designs optimally decision-theoretically and efficiently.

At the collaborative front, it solves a subproblem in decision-theoretic design: Most research on collaborative design, e.g. (Konduri and Chandrakasan, 1999), focuses on designer information sharing but not on making design choices. Exceptions, such as *Collaborative optimization* (Braun et al., 1996) are essentially point-based and only produce locally optimal designs. A decision-theoretic graphical model, termed *collaborative design network* (CDN), is proposed in (Xiang et al., 2004). Component-centered design is considered in the context of product lifecycle management (PLM) with the objective of overall optimal performance counting diverse uncertainty from materials, manufacturing and operating, as well as preference of vendors and users. A scheme for multiagent collaboration is developed (Xiang et al., 2005) which reduces complexity exponentially from that of a centralized design by exhaustive evaluation. This contribution goes further to enable efficient local design at each agent. Additional constraints for CDN are also proposed to facilitate model construction and verification.

2 Design Networks

As introduced above, this work is an essential part of the research towards multiagent collaborative design. Since the focus here is the decision process at a single agent, multiagent issues can be set aside without affecting the integrity of the result presented. In this section, we define a *design network* (DN) as the key graphical model associated with such an agent. The definition here extends that in (Xiang et al., 2004) so that it is not only more expressive in representing design knowledge, but also more restrictive to better guide model construction and verification. Aspects in (Xiang et al., 2004) are outlined here briefly for completeness. Readers are referred to reference for more details.

A design network is a triple $S = (V, E, P)$ whose structure is a connected DAG $G = (V, E)$. The set of nodes, each corresponds to a variable, is $V = D \cup T \cup M \cup U$. D is an non-empty set of *design parameter*. T is a set of *environmental factors* representing the uncertain manufacturing and operating environment for 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* of the designer.

E is an non-empty set of *legal* arcs. We refer to distinct endpoints from D as d and d', from T as t and t', from M as m and m', from U as u and u', respectively. There are six types of legal arcs:

1. Arc (d, d') signifies that the two parameters are involved in a design constraint.
2. Arc (d, m) represents that performance m depends on design parameter d.
3. Arc (t, t') represents dependency between environmental factors. This type of arcs is added to legal arcs in (Xiang et al., 2004) to encode complex dependence relations among environmental factors.
4. Arc (t, m) signifies that performance m depends on environment factor t.
5. Arc (m, m') defines m' as a composite performance measure.
6. Arc (m, u) signifies that utility u depends on performance m.

P is a set of potentials one associated with each node x in the *form* of a conditional probability distribution $P(x|\pi(x))$, where $\pi(x)$ is the set of parent nodes of x. According to legal arcs, a design parameter d can have only design parameters as parents. If d is a root, $P(d)$ is a constant distribution. Otherwise, $P(d|\pi(d))$ must contain the value 0 (i.e., not strictly positive). The non-strictly-positive requirement can be understood as follows: When $\pi(d)$ is non-empty, $P(d|\pi(d))$ represents design constraints. It specifies under what configurations of $\pi(d)$, certain values of d are illegal, and the value 0 signifies violation of design constraints.

Potential $P(t|\pi(t))$ is a typical probability distribution. According to legal arcs, $\pi(t)$ consists of environmental factors only. $P(m|\pi(m))$ is also a typical probability distribution, representing the uncertain dependence of the performance on design, environment, as well as other performance measures.

All utility variables are binary with domain $\{y, n\}$. The potential $P(u = y|\pi(u))$ is assigned a utility function $u(\pi(u))$ whose values range in $[0, 1]$ and include the two bounds. According to legal arcs, $\pi(u)$ consists of performance measures only. The potential $P(u = n|\pi(u))$ is assigned $1 - P(u = y|\pi(u))$. Each node u is also assigned a weight $k \in [0, 1]$ such that the weights of all utility nodes sum to one.

With P thus defined, S is *syntactically* a Bayesian network. Assuming additive independence (Keeney and Raiffa, 1976) among utility variables, the expected utility of a design $\mathbf{d}$

$$EU(\mathbf{d}) = \sum_i k_i \, (\sum_{\mathbf{m}} u_i(\mathbf{m}) \; P(\mathbf{m}|\mathbf{d})) \qquad (1)$$

can be computed by standard probabilistic inference in S (Xiang et al., 2004), where $\mathbf{d}$ (bold) is a configuration of D, i indexes utility nodes in U, $\mathbf{m}$ (bold) is a configuration of the parents of u_i, and k_i is the weight associated with u_i. Figure 1 shows a trivial example DN.

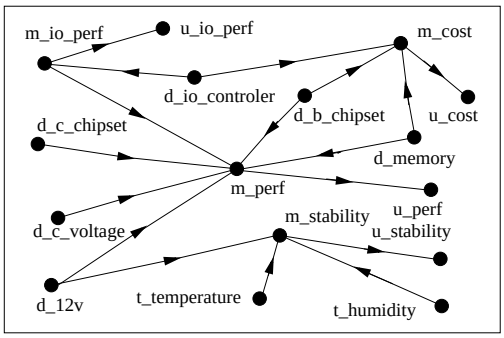

Figure 1: A trivial DN for PC motherboard.

Restriction of legal arcs does not constrain S sufficiently so that every component is relevant to the design task, as will become clear below. We impose the following *essentiality* requirement to provide further guidance to model construction and to facilitate model verification.

For any design parameter d, if there exists a directed path in S from d to a utility node u, then d is *essential*. Otherwise, d is *non-essential*. If d is non-essential, then the expected utility of any given design is independent of the value that d takes. In other words, the optimal design over the remaining design parameters is independent of d, and the optimal

value for d is undefined. Hence, d can be removed from S without affecting the optimal design over remaining design parameters and the maximum expected utility.

For any performance measure m, if there exists a directed path in S from a design parameter d to m, as well as a directed path from m to a utility node u, then m is *essential*. Without the path from d to m, m will not depend on any design. Without the path from m to u, m will not influence the optimal design. In either case, m is deemed *non-essential* and can be removed without affecting the optimal design.

The case for environmental factors differs from the above. Consider a path $m \leftarrow t \rightarrow t'$, where m is essential and t' is a leaf. Suppose that the value of t' is known at the time of design (an extension to the context for Eqn. (1)). Depending on the known value of t', the impact of t on m may differ. If a directed path from each environmental factor to a utility node is required, the node t' above will be disallowed. Therefore, an environmental factor t is deemed *essential* if there exists an undirected path in S from t to a performance measure m. Otherwise, t is *non-essential*.

We require that all design parameters, performance measures, and environmental factors in a design network to be essential.

3 Detecting Illegal Designs

An agent equipped with a DN can compute expected utility $EU(\mathbf{d})$ (Eqn. (1)) for each alternative design using standard probabilistic reasoning. However, finding the optimal design by exhaustively evaluating each design has the complexity $O(\kappa^{|D|})$, where κ is the maximum number of values that a design parameter can take. We present a much more efficient method below.

An illegal design violates at least one design constraint. The sooner it can be detected, the sooner the evaluation computation can be directed to alternative designs and the more efficient the overall design computation. Suppose that a constraint involves a subset $X \subset D$ of binary design parameters $d_0, ..., d_j$, where $d_1, ..., d_j$ are the parents of d_0 in the DN, such that $d_0, ..., d_j$ cannot take value 0 all at the same time. This is specified in the DN by $P(d_0 = 0 | d_1 = 0, ..., d_j = 0) = 0$. Any design that are consistent with the configuration $(d_0 = 0, ..., d_j = 0)$ is an illegal design. Note that the number of such illegal designs are in the order of $O(k^{|D \setminus X|})$.

A DN is syntactically a Bayesian network and hence can be compiled into a junction tree (JT) T. Due to the moralization step in compilation, there exists a cluster $C \supseteq X$ in T. After compilation, each cluster Q in T is assigned a potential $B(Q)$ over the set Q of member variables. The assignment to C satisfies $B(d_0 = 0, ..., d_j = 0) = 0$.

First, we assume that $C = X$. Consider a different configuration of C whose potential value is non-zero, for instance, the configuration $(d_0 = 0, d_1 = 1, d_2 = 0, ..., d_j = 0)$ such that $B(d_0 = 0, d_1 = 1, d_2 = 0, ..., d_j = 0) > 0$. To evaluate any design consistent with $(d_0 = 0, ..., d_j = 0)$, values $d_0 = 0$, ..., $d_j = 0$ are entered into C. When $d_1 = 0$ is entered into C, the potential value $B(d_0 = 0, d_1 = 1, d_2 = 0, ..., d_j = 0)$ is multiplied by 0 because the configuration is inconsistent with $d_1 = 0$. As the result, the updated potential value is $B(d_0 = 0, d_1 = 1, d_2 = 0, ..., d_j = 0) = 0$. To summarize, before entering values $d_1 = 0$, the potential value $B(d_0 = 0, ..., d_j = 0)$ is 0, and after entering $d_1 = 0$, the potential value for every other configuration is 0. Hence, after entering $d_0 = 0$, ..., $d_j = 0$, every potential value in $B(C)$ becomes 0.

Next, assume that $C = X \cup Y$, where $Y \neq \emptyset$ and $Y \cap X = \emptyset$. Since any potential $B(Y, d_0, ..., d_j)$ can be factorized into $B(Y, d_0, ..., d_j)B(d_0, ..., d_j)$, we conclude from the above that after entering $d_0 = 0$, ..., $d_j = 0$, every potential value in $B(C)$ becomes 0.

This analysis suggests a method to detect illegal designs. When a particular design (possibly illegal) is evaluated, for each cluster C that has been assigned $P(d|\pi(d))$ relative to a design parameter d, enter the value of d and the value of each design parameter in $\pi(d)$ into C. After entering, check if the sum of potential values of

$B(C)$ is 0. As soon as a positive test occurs, conclude that all designs that are consistent with this configuration of $(d, \pi(d))$ are illegal. There will be no need to evaluate any one of them.

4 Domain Division

In addition to early detection of illegal designs, we seek to evaluate legal designs more efficiently as follows. We assume that some separators in T consist of design parameters only. The structure of an example DN is shown in Figure 2. It is converted into the JT in Figure 3. In the JT, three separators $\{d\_a, d\_c\}$, $\{d\_b, d\_c\}$ and $\{d\_a, d\_d\}$ consist of design parameters only.

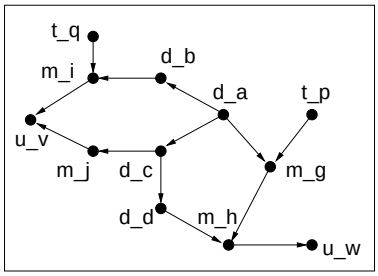

Figure 2: The structure of a design network.

Using these separators as boundaries, we compile the JT T into a representation, called *division tree* that is more effective for design computation.

Definition 1. Let S be a design network over an nonempty set of essential variables $V = D \cup T \cup M \cup U$. A `division tree` for S is a tuple $\Gamma = (V, \Delta, \Theta, \Upsilon)$. V is the `generating set` of Γ. Δ is a subset of the powerset $POW(V)$ of V such that $\cup_{Q \in \Delta} = V$. Each element of Δ is called a `division`. Θ is composed of the following:

$$\Theta = \{< Q, Q' > | Q, Q' \in \Delta, Q \neq Q',$$

$$Q \cap Q' \neq \emptyset, Q \cap Q' \subset D\}.$$

Each unordered pair $< Q, Q' >$ is called a `separator` between the two divisions and is labeled by the intersection $Q \cap Q'$. Δ and Θ are so composed that (V, Δ, Θ) forms a junction tree. Υ is a set of `division junction trees`. Its elements map one-to-one to elements of Δ such that each division junction tree has the corresponding division as its generating set.

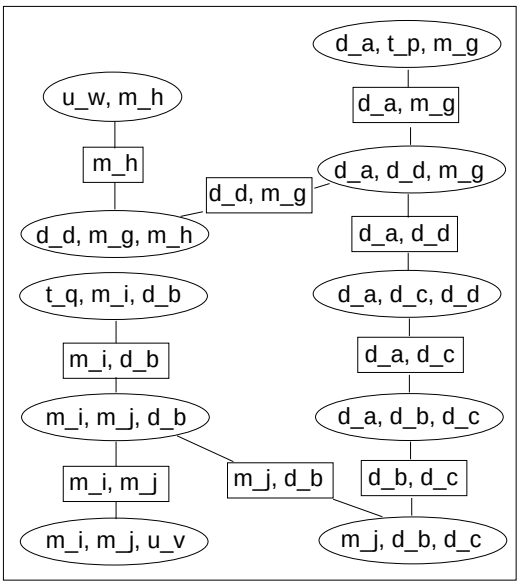

Figure 3: A junction tree representation of design network in Figure 2.

As an example, consider a division tree for the DN S in Figure 2. The set V of variables in S is the generating set. The set of divisions is $\Delta = \{V_0, V_1, V_2, V_3\}$ as shown in Figure 4. The set Θ of division separators is also shown in the figure. It can be easily verified that the graph depicted is a JT. The set of division JTs is $\Upsilon = \{ST_0, ST_1, ST_2, ST_3\}$ as shown in Figure 5, where the generating set of division JT ST_i is division V_i.

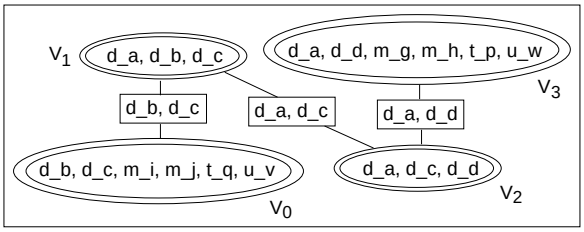

Figure 4: The division tree for design network in Figure 2. Each double-oval represents a division. Each box represents a separator.

Once DN S has been compiled into JT T, construction of a division tree Γ is straightforward: The set of division JTs are obtained from T by removing its separators that are composed of design parameters only. For instance, after removing separators $\{d\_a, d\_c\}$, $\{d\_b, d\_c\}$ and

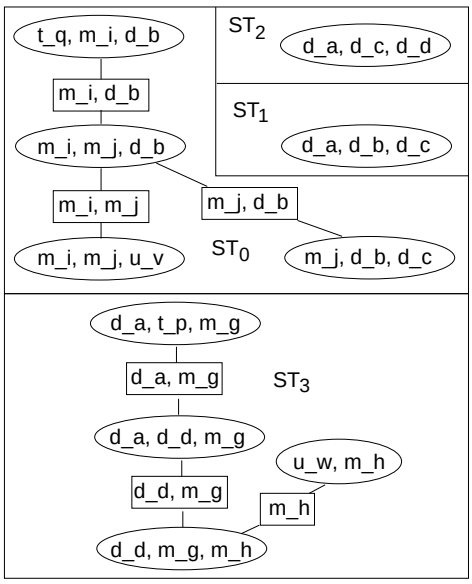

Figure 5: Division JTs for division tree in Figure 4. ST_i corresponds to division V_i.

$\{d\_a, d\_d\}$, T is split into division JTs in Figure 5. The generating sets of these division JTs become divisions in Δ. The removed separators become division separators in Θ. It is a simple matter to show that (V, Δ, Θ) thus constructed forms a junction tree.

5 Design Evaluation within Division

Each division V_i contains an nonempty subset D_i of design parameters, where i indexes the division. This is true since the division contains at least the design parameters that form the separator between itself and another division.

First, for each configuration of D_i, whether it is a legal partial design needs to be determined, to the extent allowed by information available in the division. This can be achieved as outlined in Section 3. A division may or may not contain utility variables. If it does not contain utility variables, then the evaluation within the division is complete. For the division tree in Figure 4, V_1 and V_2 are such divisions.

If the division contains utility variables, then for each legal configuration of D_i, further evaluation needs to be performed. This can be done by belief propagation within the division JT. The expected utility for each utility function can then be retrieved from the corresponding utility variable. The expected utility contribution of the partial design can be obtained by combining the result from individual utility variables.

More precisely, in division V_i, the following is obtained for each configuration $\mathbf{d_i}$ of D_i,

$$EU(\mathbf{d_i}) = \sum_j k_j \ (\sum_{\mathbf{m}} u_j(\mathbf{m}) \ P(\mathbf{m}|\mathbf{d_i}))$$

where j indexes utility nodes in V_i, $\mathbf{m}$ is a configuration of the parents of u_j, and k_j is the weight associated with u_j.

We extend the definition of $EU(\mathbf{d_i})$ to divisions without utility variables and to illegal partial design $\mathbf{d_i}$ as follows: We set $EU(\mathbf{d_i}) = 0$ if division V_i contains no utility variables. We set $EU(\mathbf{d_i}) = null$ if $\mathbf{d_i}$ is illegal.

After the above evaluation on each configuration of D_i, the design evaluation within the division is complete.

6 Message Passing in Division Tree

To combine design evaluations at individual divisions and determine the optimal design, messages are passed between divisions, along the separators of the division tree (e.g., Figure 4). The message passing is organized into three rounds along the division tree. The syntax and semantics of messages in each round differ.

The algorithms that we present below are mostly executed by individual divisions, except one by the agent A. Without losing generality, we denote the division object executing the algorithm by V_0. The execution is activated by a caller, denoted by V_c, which is either an adjacent division of V_0 in Γ or agent A. The separator between caller division V_c and V_0 is denoted as R_c. If V_0 has additional adjacent divisions, they are denoted as $V_1, V_2, ..., V_z$ and their separators with V_0 are denoted as $R_1, R_2, ..., R_z$, respectively.

7 Collecting Utility Contribution

In the first round of message passing, division V_0 receives a vector message from each adjacent division V_i. Elements of the vector are indexed by

partial designs over R_i. The kth element of the vector, indexed by partial design $\mathbf{e_i^k}$, is denoted as MEV_i^k. The corresponding components for the vector message that V_0 sends to division V_c are $\mathbf{e_c^k}$ (over R_c) and MEV_c^k, respectively.

If V_i is a leaf on the division tree (whose only adjacent division is V_0), MEV_i^k corresponds to the maximum expected utility

$$MEV_i^k = max_{\mathbf{d_j}} EU(\mathbf{d_i})$$

over all partial design $\mathbf{d_i}$ that are consistent with $\mathbf{e_i^k}$. Otherwise, its interpretation becomes clear below.

The following algorithm, when executed by each division, propagates utility contributions of partial designs at individual divisions inwards along division tree. The vector message sent from V_i to V_0 is denoted as MEV_i and that sent from V_0 to V_c is denoted as MEV_c.

Algorithm 1 *(CollectDivisionUtility).* When division V_0 is called by V_c to CollectDivisionUtility, it does the following:

1. For each adjacent division V_i, V_0 calls CollectDivisionUtility in V_i and receives MEV_i from V_i.

2. For each partial design $\mathbf{d_0}$, update

$$EU(\mathbf{d_0}) = EU(\mathbf{d_0}) + \sum_i MEV_i^k,$$

   where MEV_i^k is indexed by partial design $\mathbf{e_i^k}$ and $\mathbf{e_i^k}$ is consistent with $\mathbf{d_0}$.

3. If V_c is an adjacent division, for each partial design $\mathbf{e_c^k}$, V_0 computes

$$MEV_c^k = max_{\mathbf{d_0}} EU(\mathbf{d_0})$$

   over all partial design $\mathbf{d_0}$ that are consistent with $\mathbf{e_c^k}$, labels one such partial design that reaches the value MEV_c^k by $\mathbf{d_c^{k*}}$ (breaking ties arbitrarily), and sends MEV_c to V_c.

Note that $EU(\mathbf{d_0})$ may be equal to *null*. When *null* participates in an addition, we require that the sum is *null*. When *null* participates in a *max* operation, we require that the result is *null* if and only if all operants are *null*.

8 Distributing Optimal Design

In the second round of message passing, division V_0 receives from division V_c a partial design $\mathbf{e_c^k}$ that is consistent with the optimal design. Combining it with the design evaluation performed earlier within the division, V_0 identifies the optimal partial design within the division. It then sends the optimal partial design relative to the separator of each adjacent division to the corresponding V_i. As message passing progresses, each division identifies the optimal partial design within the division. This is achieved by the following algorithm.

Algorithm 2 *(DistributeOptimalDivisionDesign).* When division V_0 is called by V_c to DistributeOptimalDivisionDesign, it does the following:

1. If V_c is an adjacent division, V_0 receives a partial design $\mathbf{e_c^k}$ over R_c from V_c. Then, among partial designs over D_0 that are consistent with $\mathbf{e_c^k}$, it identifies the one with the highest EU value (breaking ties arbitrarily) and label it as $\mathbf{d_0^*}$.

2. Otherwise, among all partial designs over D_0, it identifies the one with the highest EU value (breaking ties arbitrarily) and label it as $\mathbf{d_0^*}$.

3. For each adjacent division V_i, call DistributeOptimalDivisionDesign in V_i and send the partial design $\mathbf{e_i^k}$ that is consistent with $\mathbf{d_0^*}$ to V_i.

9 Collecting Optimal Design

In the last round of message passing, division V_0 receives the optimal partial design over D_i from each adjacent division V_i. It combines them with its own optimal partial design over D_0 and sends the result to division V_c. At the end of this round, the optimal design over D is obtained.

Algorithm 3 *(CollectOptimalDesign).* When division V_0 is called by V_c to CollectOptimalDesign, it does the following:

1. For each adjacent division V_i, V_0 calls CollectOptimalDesign in V_i and receives $\mathbf{d_i^*}$ from V_i.

2. Combine $\mathbf{d_0^*}$ with all $\mathbf{d_i^*}$ received and send result to V_c.

The following algorithm activates the three rounds of message passing in turn and is executed by the agent A.

Algorithm 4 *(OptimalDesignByDivisionTree).*

1. Select a division V_x arbitrarily.
2. Call CollectDivisionUtility in V_x.
3. Call DistributeOptimalDivisionDesign in V_x.
4. Call CollectOptimalDesign in V_x.
5. When V_x finishes, receive from it design $\mathbf{d}$ over D.

Soundness of OptimalDesignByDivisionTree is established below. Proposition 2 asserts that $\mathbf{d}$ produced in the algorithm is a legal design. Only a proof sketch is given due to space limit.

Proposition 2. The design $\mathbf{d}$ produced by OptimalDesignByDivisionTree is legal.

Proof sketch: View the division tree as rooted at V_x and use induction on its depth *dep*. The base case is $dep = 0$. There is a single division and the proposition can be easily shown. Assume the proposition for $dep \leq m$ and consider $dep = m+1$. Let the root division be V_0 and its adjacent divisions be V_i $(i = 1, ..., z)$. The subtree rooted at each V_i has a depth $\leq m$. By assumption, if CollectDivisionUtility is called on each V_i by the agent, followed by a call of DistributeOptimalDivisionDesign on V_i, followed by a call of CollectOptimalDesign on V_i, then the (partial) design returned by V_i is legal relative to the subtree.

The actual execution differs from the above as follows: CollectDivisionUtility is called on each V_i by V_0 which receives MEV_i from V_i. For each element MEV_i^k in MEV_i, it is *null* if there exists no legal (partial) design consistent with $\mathbf{e_i^k}$ in the subtree rooted at V_i. By null addition, any partial design $\mathbf{d_0}$ in V_0 consistent with $\mathbf{e_i^k}$ will be evaluated to $EU(\mathbf{d_0}) = null$. Hence, $\mathbf{d_0}$ cannot be selected as $\mathbf{d_0^*}$ by V_0 during DistributeOptimalDivisionDesign and cannot be part of the design returned through CollectOptimalDesign. [end of sketch]

Theorem 3 shows that $\mathbf{d}$ is an optimal design.

Theorem 3. The design $\mathbf{d}$ produced by OptimalDesignByDivisionTree is optimal.

Proof sketch: It suffices to show that $EU(\mathbf{d_0^*})$ obtained by the root division V_0 in step 2 of DistributeOptimalDivisionDesign is the maximum expected utility over all legal designs. Once this is established, it follows that a design, that attains this maximum expected utility and is restricted to each division, is $\mathbf{d_0^*}$ labeled by the corresponding division during DistributeOptimalDivisionDesign. CollectOptimalDesign simply assembles them together. The body of the proof uses induction with a structure similar to the proof above. [end of sketch]

10 Complexity

Denote the total number of design parameters by $|D|$ and the maximum number of possible values of a design parameter by κ. A centralized optimal design that evaluates all designs exhaustively has the complexity $O(\kappa^{|D|})$.

For OptimalDesignByDivisionTree, let the number of divisions be $|\Delta|$, the maximum number of design parameters per division be δ, and the maximum cardinality of division separators be q. During CollectDivisionUtility, each division evaluates $O(\kappa^{\delta})$ partial designs and sends a message of size $O(\kappa^q)$ to the caller. Hence, the complexity of OptimalDesignByDivisionTree is $O(|\Delta|\ \kappa^{\delta} + (|\Delta| - 1)\ \kappa^q)$. Normally, q is much smaller than δ and the complexity becomes $O(|\Delta|\ \kappa^{\delta})$. When δ is upper-bounded, OptimalDesignByDivisionTree is efficient.

11 Other Related Work

In addition to previous work reviewed in Section 1, we discuss relations of this contribution to other related work below:

In the literature on graphical models, a related work is strong junction tree (Jensen et al., 1994) which addresses the issue of sequential decision making. As indicated by Paredis

et al. (Paredis et al., 2006), point-based design corresponds to sequential decision making. Instead, the optimal design addressed in this contribution takes the approach of set-based design, which involves simultaneous evaluation of a much large number of design parameters than what is considered in a typical sequential decision problem. The issue of design constraint violation is also dealt with in this contribution.

Another related work is nested junction trees (Kjaerulff, 1997), where a JT is nested in a cluster of a higher level JT to reduce space complexity and to allow more efficient belief propagation. The division tree, proposed in this contribution, is also a nested JT structure. The computation conducted within a division, however, goes beyond probabilistic reasoning. It reasons about design decisions, constraints and utilities, and is decision-theoretic in nature.

In the literature on constraint satisfaction problem (CSP), constraint graphs are converted to JTs to solve CSPs (Dechter and Pearl, 1989). The current contribution essentially extends that approach to constraint optimization and with a decision-theoretic objective function.

12 Conclusion

We refined the definition for design networks regarding legal arcs and essentiality. The new definition improves expressiveness and guidance to model construction and verification. We compiled a design network into a division tree and presented algorithms that combine probabilistic, constraint-based and decision-theoretic reasoning for optimal design in the compiled structure. This result not only provides a computational mechanism for single-agent optimal design, but also fills in a gap in optimal collaborative design.

The presented algorithm suite solves the problem of decision-theoretic optimal design in the context of catalog design where discrete design options are configured. The algorithm suite derives its efficiency by decomposing the design domain according to design separators in division trees. When multiple optimal designs exist, the algorithm suite returns one of them arbitrarily, but can be extended to return all.

Acknowledgments

Financial support to this research is provided by NSERC of Canada.

References

R.D. Braun, I.M. Kroo, and A.A. Moore. 1996. Use of the collaborative optimization architecture for launch vehicle design. In *Proc. 6th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, pages 306–318.

R. Dechter and J. Pearl. 1989. Tree clustering for constraint networks. *Artificial Intelligence*, 38(3):353–366.

F. Jensen, F.V. Jensen, and S.L. Dittmer. 1994. From influence diagrams to junction trees. In *Proc. 10th Conf. on Uncertainty in Artificial Intelligence*, pages 367–373.

R.L. Keeney and H. Raiffa. 1976. *Decisions with Multiple Objectives*. Cambridge.

U. Kjaerulff. 1997. Nested junction trees. In *Proc. 13th Conf. on Uncertainty in Artificial Intelligence*, pages 294–301, Providence, Rhode Island.

G. Konduri and A. Chandrakasan. 1999. A framework for collaborative and distributed web-based design. In *Proc. 36th Design Automation Conference*, pages 898–903.

C. Paredis, J. Aughenbaugh, R. Malak, and S. Rekuc. 2006. Set-based design: a decision-theoretic perspective. In *Proc. Frontiers in Design & Simulation Research 2006 Workshop*, pages 1–25.

D.K. Sobek, A.C. Ward, and J.K. Liker. 1999. Toyota's principles of set-based concurrent engineering. *Sloan Management Review*, 40(2):67–84.

A.C. Ward. 1989. *A Theory of Quantitative Inference Applied to A Mechanical Design Compiler*. Ph.D. thesis, MIT, Department of Mechanical Engineering.

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

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