

# Linear Functions for Interactive Configuration Using Join Matching and CSP Tree Decomposition

Sathiamoorthy Subbarayan, Henrik Reif Andersen  
Department of Innovation, IT University of Copenhagen  
Denmark

## Abstract

Quick responses are required for interactive configuration. For this an ideal interactive configurator needs to provide the functionalities required for interactive configuration with at most linear time complexity. In this paper, we present such a data structure called *Join Matched CSP* (JMCSP). When a JMCSP is used to represent a configuration problem, the functionalities required for interactive configuration can be obtained with linear time complexity.

Unlike the tree-of-BDDs [Subbarayan, 2005], the JMCSPs while taking advantage of tree-decomposition also provide linear configuration functions. Although obtaining a JMCSP is exponential in the tree-width of the input configuration problem, due to tree-like hierarchical nature of configuration problems, this is typically feasible. We present the JMCSP compilation process along with the linear interactive configuration functions on it. The linear functionalities provided by the JMCSPs include computation of *all minimum explanations*.

## 1 Introduction

The complexity of made-to-order products keeps increasing. Examples of such products include personal computers, bikes, and power-backup systems. Such products will be represented in the form of a product model. A product model will list the parameters (variables) defining the product, their possible values and the rules by which those parameter values can be chosen. A product model implicitly represents all valid configurations of a product, and it can be viewed as a Constraint Satisfaction Problem (CSP), where the solutions to the CSP are equivalent to valid configurations of the corresponding product model.

The increase in the complexity of made-to-order products rises the need for efficient decision support systems to configure a product based on the requirements posed by the customer. Such decision support systems are called configurators. Configurators read a product model which represents all valid configurations of the product and guides the user in choosing one among the valid configurations as close as possible to his requirements. An interactive configurator takes

a product model as input and interactively helps the user to choose his preferred values for the parameters in the product model, one-by-one.

The interactive configurator needs to be complete and backtrack-free. Complete means that all valid configurations need to be configurable using the configurator. Backtrack-free means that the configurator should not allow the user to choose a value for a parameter in the product model which would eventually lead him to no valid configurations. To ensure backtrack-freeness, the interactive configurator needs to prune away values from the possible values of parameters in the product model as and when those values will not lead to any valid configuration. In addition, the interactive configurator needs to give responses in a short period of time. Examples for commercial interactive configurators include Configit Developer [Configit-Software, 2005] and Array Configurator [Array-Technology, 2005].

The Binary Decision Diagram (BDD) [Bryant, 1986] based symbolic CSP compilation technique [Hadzic *et al.*, 2004; Subbarayan *et al.*, 2004] can be used to compile all solutions of a configuration problem into a single (monolithic) BDD. Once a BDD is obtained, the functions required for interactive configuration can be efficiently implemented. The problem with such approaches is that they do not exploit the fact that configuration problems are specified in hierarchies. Due to this the BDD obtained after compilation could be unnecessarily large. Such hierarchies are closer to trees in shape. The tree-of-BDDs approach: a combination of the BDD-based compilation technique and a CSP decomposition technique for efficient compilation of all solutions was presented in [Subbarayan, 2005]. The tree-of-BDDs scheme exploited the tree-like nature of configuration problems. The advantage of monolithic-BDD is that it can provide linear functions for interactive configuration. The advantage of tree-of-BDDs is that they need very small space to store, when compared with that of the monolithic-BDD. But, at the cost of having costly functions for interactive configuration. The results in [Subbarayan, 2005] have shown that, for functionalities like *minimum explanation generation*, the tree-of-BDDs can take significantly long time to respond. Hence, we would like to have a compilation scheme that takes advantage of the tree-decomposition techniques and at the same time provide linear interactive configuration functions.

Towards this we introduce the notion of *Join Matching* in

tree-structured CSP instances. Given a configuration problem, we can use tree-decomposition techniques to obtain an equivalent tree-structured CSP. Then, we can easily obtain a *join tree* for the tree-structured CSP. By performing Join Matching on the join tree, we obtain a data structure called *Join Matched CSP* (JMCSPP). The size of JMCSPP is exponential in the tree-width of the input configuration problem. As the configuration problems typically have very low tree-width, this should be practically feasible. We also present the linear functions required for interactive configuration using JMCSPPs.

This is a first step towards join matching in other configuration-problem compilation schemes, using the compression data structures like BDDs, DNNF [Darwiche, 2001], automata [Amilhastre *et al.*, 2002; Fargier and Vilarem, 2004], and cartesian product representation [Madsen, 2003]. That might lead to linear functions for interactive configuration with additional reduction in space than just using JMCSPPs. The other potential applications of join matching include: *the general constraint propagation techniques, model-based diagnosis, and database systems.*

In Section 2, the basic definitions are given. In Section 3 the CSP tree-decomposition techniques are discussed. The Section 4 describes the join matching process for compiling JMCSPPs. The following section presents the interactive configurator algorithm. The Section 6 describes the linear functions for interactive configuration using the JMCSPPs. Discussion on future work and related work, followed by concluding remarks, finish this paper.

## 2 Background

In this section we give the necessary background.

Let  $X$  be a set of variables  $\{x_1, x_2, \dots, x_n\}$  and  $D$  be the set  $\{D_1, D_2, \dots, D_n\}$ , where  $D_i$  is the domain of values for variable  $x_i$ .

**Definition** A relation  $R$  over the variables in  $M$ ,  $M \subseteq X$ , is a set of allowed combinations of values (tuples) for the variables in  $M$ . Let  $M = \{x_{m_1}, x_{m_2}, \dots, x_{m_k}\}$ , then  $R \subseteq (D_{m_1} \times D_{m_2} \times \dots \times D_{m_k})$ .  $R$  restricts the ways in which the variables in  $M$  could be assigned values.

**Definition** A constraint satisfaction problem instance CSP is a triplet  $(X, D, C)$ , where  $C = \{c_1, c_2, \dots, c_m\}$  is a set of constraints. Each constraint,  $c_i$ , is a pair  $(S_i, R_i)$ , where  $S_i \subseteq X$  is the scope of the constraint and  $R_i$  is a relation over the variables in  $S_i$ .

We assume that the variables whenever grouped in a set are ordered in a fixed sequence. The same ordering is assumed on any set of the values of variables and the pairs with variables in them.

**Definition** An assignment for a variable  $x_i$  is a pair  $(x_i, v)$ , where  $x_i \in X$  and  $v \in D_i$ . The assignment  $(x_i, v)$  binds the value of  $x_i$  to  $v$ . A partial assignment PA is a set of assignments for all the variables in  $Y$ , where  $Y \subseteq X$ . A partial assignment is complete when  $Y = X$ .

The notation  $PA|_{xs}$ , where  $xs$  is a set of variables, means the restriction of the elements in PA to the variables in  $xs$ .

Similarly,  $R|_{xs}$ , where  $R$  is a relation, means the restriction of the tuples in  $R$  to the variables in  $xs$ .

**Definition** Let the set of variables assigned values by a PA be  $var(PA) = \{x_i | \exists x_i. (x_i, v) \in PA\}$ . Let the tuple of values assigned by a PA be  $val(PA) = (v_{l_1}, \dots, v_{l_j})$  when  $var(PA) = \{x_{l_1}, \dots, x_{l_j}\}$  and  $(x_{l_n}, v_{l_n}) \in PA$ . A partial assignment PA satisfies a constraint  $c_i$ , when  $val(PA|_{var(PA) \cap S_i}) \in R_i|_{var(PA) \cap S_i}$ .

**Definition** A complete assignment CA is a solution  $S$  for the CSP when CA satisfies all the constraints in  $C$ .

Given a CSP instance, the set of all complete assignments is  $CAS \equiv D_1 \times D_2 \times \dots \times D_n$ . Let SOL denote the set of all solutions of the CSP. Then,  $SOL \subseteq CAS$ .

A *configurator* is a tool which helps the user in selecting his preferred product, which needs to be valid according to a given product specification. An *interactive configurator* is a configurator which interacts with the user as and when a choice is made by the user. After each and every choice selection by the user the interactive configurator shows a list of unselected options and the valid choices for each of them. The interactive configurator only shows the list of valid choices to the user. This prevents the user from selecting a choice, which along with the previous choices made by the user, if any, will result in no valid product according to the specification. The configurator, hence, automatically hides the *invalid choices* from the user. The invalid choices will still be visible to the user, but with a tag that they are inconsistent with the current partial assignment. When the current partial assignment is extended by the user, some of the choices might be implied for consistency. Such choices are automatically selected by the configurator and they are called *implied choices*.

A configuration problem can be modelled as a CSP instance, in which each available option will be represented by a variable and the choices available for each option will form the domain of the corresponding variable. Each rule in the product specification can be represented by a corresponding constraint in the CSP instance. The CAS and SOL of the CSP instance will denote the all possible-configurations and all possible-valid-configurations of the corresponding configuration problem. Hereafter, the terms CSP, CAS, and SOL may be used directly in place of the corresponding configuration terms.

Let us assume that the SOL of a configuration problem can be obtained. Given SOL, the three functionalities required for interactive configuration are: *Display*, *Propagate*, and *Explain*.

**Definition** Display is the function, which given a CSP, a subset of its solutions space  $SOL' \subseteq SOL$ , lists  $X$ , the options in CSP and  $CD_i$ , the available valid choices, for each option  $x_i \in X$ , where  $CD_i = \{v | (x_i, v) \in S, S \in SOL'\}$ .

$CD_i$  is the current valid domain for the variable  $x_i$ . The display function is required to list the available choices to the user.

**Definition** Propagate is the function, which given a CSP, a subset of its solutions space  $SOL' \subseteq SOL$ , and  $(x_i, v)$ , where  $v \in CD_i$ , restricts  $SOL'$  to  $\{S | (x_i, v) \in S, S \in SOL'\}$ .

By the definition of interactive configurator, the propagation function is necessary. Propagate could also be written as restricting SOL' to  $SOL'_{|(x_i, v)}$ . Sometimes the restriction might involve a set of assignments, which is equivalent to making each assignment in the set one by one.

**Definition** A choice  $(x_i, v_i)$  is an *implied choice*, when  $(x_i, v_i) \notin PA$  and  $(SOL_{|PA})_{|x_i} = \{v_i\}$ . A choice  $(x_i, v_i)$  is an *invalid choice*, when  $\{v_i\} \notin (SOL_{|PA})_{|x_i}$ . Let  $(x_i, v_i)$  be an implied or invalid choice. Then  $Explain(x_i, v_i)$  is the process of generating, E, a set of one or more selections made by the user, which implies or invalidates  $(x_i, v_i)$ . The E is called as an explanation for  $(x_i, v_i)$ .

An explanation facility is required when the user wants to know why a choice is implied or invalid. Let PA be the current partial assignment that has been made by the user. By the definition of explanation,  $Display(CSP, SOL_{|PA \setminus E})$  will list  $v_i$  as a choice for the unselected option  $x_i$ .

Each selection,  $(x_i, v)$ , made by the user could be attached a non-negative value as its priority,  $P(x_i, v)$ , and the explain function can be required to find a minimum explanation.

**Definition** The *cost* of an explanation is  $Cost(E) = \sum_{(x_i, v) \in E} P(x_i, v)$ . An explanation E is *minimum*, if there does not exist an explanation E' for  $(x_i, v_i)$ , such that  $Cost(E') < Cost(E)$ .

Minimum explanations are useful when different options in a product model have different priorities. For example, in a car configuration problem the main options like engine could be given high priority. Once the user decides on an option of high priority, minimum explanations will try to protect the high priority decision as much as possible.

The complexity of the three functions—display, propagate and explain—are exponential in the size of the input configuration problem.

### 3 Tree Decomposition of CSPs

The following definitions [Dechter, 2003] will be useful in discussing the tree decomposition techniques for CSPs.

**Definition** The *constraint graph*  $(V, E)$  of a given CSP will contain a node for each constraint in the CSP and an edge between two nodes if their corresponding constraints share at least one variable in their scopes. Each edge will be labelled by the variables that are shared by the scope of the corresponding constraints.

**Definition** A subset of the edges,  $(E' \subseteq E)$ , in a constraint graph is said to satisfy the *connectedness* property, if for any variable  $v$  shared by any two constraints, there exists a path between the corresponding nodes in the constraint graph, using only the edges in  $E'$  with  $v$  in their labels.

**Definition** A *join graph*  $(V, E_j)$  of a constraint graph contains all the nodes in the constraint graph, but the set of edges in the join graph,  $E_j$ , is a subset of the edges in the constraint graph,  $E_j \subseteq E$ , such that the connectedness property is satisfied. If the join graph does not have any cycles, then it is called a *join tree*.

A constraint graph of a CSP has a join tree if its *maximum spanning tree*, when the edges are weighted by the number of shared variables, satisfies the connectedness property [Dechter, 2003]. Several tree decomposition techniques [Dechter and Pearl, 1989; Gyssens *et al.*, 1994; Gottlob *et al.*, 2000] have been proposed to convert a CSP  $C$  into  $C'$  that has a join tree.

**Definition** The *tree width* of a join tree is the maximum number of variables in any of its nodes minus one. The tree width of a CSP is the smallest one among the tree width of all possible join trees.

Finding a tree decomposition of a CSP such that the join tree of the resulting CSP has the minimum tree width is a NP-hard task [Bodlaender, 2005]. Hence, most of the tree decomposition techniques use some form of heuristics. The complexity of creating a join tree of a CSP is exponential in the tree width of the join tree.

All the tree decomposition techniques create clusters of constraints in a CSP, such that all the constraints in the CSP will be in at least one of the clusters. The CSP' obtained after the conjunction of the original constraints in each of the clusters will have a join tree. A solution for CSP or CSP' will also be a solution for the other.

**Definition** A constraint of a CSP is said to be *minimal* when all the solutions of the constraint can be extended to a solution for the CSP.

A CSP with a join tree is called *acyclic (tree-structured) CSP*. Acyclic CSPs can be efficiently solved. The nice property of an acyclic CSP is that, local consistency between constraints adjacent in its join tree implies minimality of constraints. Minimality of constraints is enough to obtain the efficient functions required for interactive configuration. Since each constraint is minimal, a valid choice for a variable in a constraint will also be a valid choice for the variable in the entire CSP, and the display function will use this to efficiently display valid choices. Given an assignment, the propagate function just needs to reduce all the constraints in which the variable is present, and propagate the effect to other constraints through *semi-joins* [Goodman and Shmueli, 1982].

Although the resulting CSP after decomposition can be exponential in the worst case, for many practical configuration instances the decomposition results in a manageable sized acyclic CSP [Subbarayan, 2005].

### 4 Join Matched CSPs

**Definition** Let  $C$  be an acyclic CSP and its join tree be  $(V, E)$ , where each element in  $V$  corresponds to a constraint in  $C$  and the edges in  $E$  are undirected. Then, a *Directed Join Tree (DJT)* is a pair  $(V, E')$ , where all the edges in  $E'$  are directed such that, each edge  $e \in E$  will be converted into a directed edge  $e' \in E'$ , and only one node in  $V$  will not have any outgoing edge.

Let the conjunction of the constraints  $c_1$  and  $c_2$  be  $(c_1 \bowtie c_2)$ . Let  $ca$  be the common variables in the scope of those constraints.

**Definition** The *common join values (CJV)* of an edge  $e'$  between the constraints  $c_1$  and  $c_2$  is defined as  $CJV = (c_1 \bowtie c_2)|_{ca}$ .

**Definition** A *common join block (CJB)* is a structure, associated with a directed edge, containing the fields:  $cja$ ,  $left\_ptrs$ ,  $right\_ptrs$ ,  $min\_cost$ ,  $min\_ptr$ .

The  $cja$  in a CJB belongs to the corresponding CJV, i.e.  $cja \in CJV$ . The  $left\_ptrs$  is a list of pointers to the tuples on the source constraint of the corresponding directed edge. The list points to the tuples which when restricted to common variables of the corresponding edge evaluates to  $cja$ . Similarly, the  $right\_ptrs$  will point to the tuples containing  $cja$  on the destination constraint. The fields  $min\_cost$  and  $min\_ptr$  will be used by the Explain function and they are described later.

Given an acyclic CSP  $C$  and its directed join tree, for each edge in the join tree we can obtain the corresponding CJV. Then for each element of CJV, we can obtain the corresponding CJB.

**Definition** Let  $CJBS$  of an edge be the set of CJBs, having a CJB for each value in the corresponding common join values (CJV).

Whenever a CJBS is created for a directed edge  $e$  from  $c_1$  to  $c_2$ , for each tuple in  $c_1$  and  $c_2$ , we add a pointer to the corresponding CJB in the CJBS. The lists  $left\_ptrs$  and  $right\_ptrs$  in the corresponding CJBs will also be updated.

Given an edge  $e$ , the process of creating CJBS and adding appropriate pointers to the tuples in  $c_1$  and  $c_2$  is called *Join Matching* for the edge  $e$ .

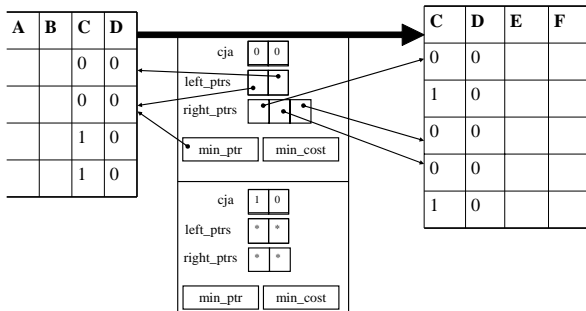


Figure 1: Join Matching for an edge.

In the example of Figure 1, the common attributes are  $\{C, D\}$ . There are two elements in the CJV  $\{(0,0), (1,0)\}$ . For the CJB corresponding to  $cja$  (0,0), the pointers stored in the CJB are marked by appropriate arrows. Note that each tuple will also have a pointer to one of those CJBs, but the figure does not show them explicitly.

**Definition** Given an acyclic CSP and its directed join tree, the process of join matching for each edge in the join tree results in a data structure called *Join Matched CSP (JMCS)*.

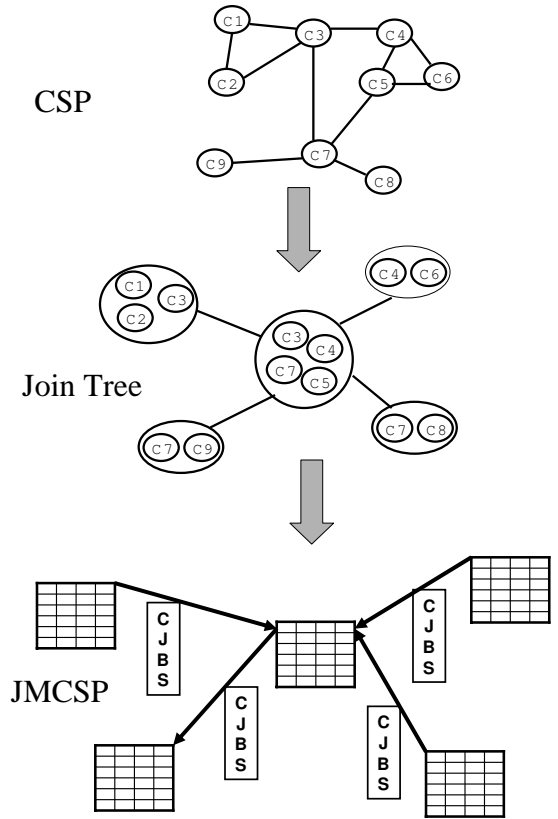


Figure 2: CSP to JMCS conversion.

Figure 2 shows an example for JMCS creation for a configuration problem represented by its constraint graph.

After creating a JMCS, minimality of the constraints in it can be obtained by removing tuples in the constraints, which do not have a counterpart in an adjacent constraint. Iteration of this process will reach a *fix point*, where all the constraints in the JMCS will be minimal.

### The Complexity of Join Tree to JMCS Compilation

It can be observed that the join matching process increases the space of each tuple by a constant. If there are totally  $t$  tuples in a join tree, then the size of JMCS is  $O(t)$ . Thus the size of JMCS is linear in the size of the join tree.

Let a join tree have  $n$  nodes (constraints) and there are  $l$  tuples in each one of those nodes. Join matching of an edge can be done by first sorting the two end constraints of the edge on the common attributes (variables). The sorting of the constraints takes  $O(l \log l)$  time. Then, an  $O(l)$  algorithm is enough to obtain the corresponding CJBS as both the end constraints are lexicographically ordered on the join attributes. Hence, the join matching of an edge can be done in  $O(l \log l)$  time. As there are  $n-1$  edges in the join tree, the join tree to JMCS conversion process takes  $O(n l \log l)$  time.

## 5 An Interactive Configurator Using JMCSPs

The algorithm for interactive configuration is given in Figure 3. The COMPILER function takes a configuration problem as input, converts it into an acyclic CSP, and then returns the corresponding JMCSP. The JMCSP represents the solution space (SOL). Since a configuration problem will not change quite often, the COMPILER function need not have to do the compilation every time the interactive configurator is used. SOL could be stored and reused.

```

INTERACTIVECONFIGURATOR(CP)
1 SOL:=COMPILE(CP)
2 SOL':=SOL, PA:={ }
3 while |SOL'| > 1
4   Display(CP, SOL')
5   (xi,v) := 'User input choice'
6   if (xi,v) ∈ CDi
7     SOL':=Propagate(CP,SOL',(xi,v))
8     PA:=PA ∪ {(xi,v)}
9   else
10    E:=Explain(CP, SOL', (xi,v))
11    if 'User prefers (xi,v)'
12      PA:=(PA \ E) ∪ {(xi,v)}
13    SOL':=SOL|PA
14 return PA

```

Figure 3: An Interactive Configurator Using JMCSPs.

## 6 Configuration Functions on JMCSPs

### 6.1 The Propagate Function

Given a JMCSP and an assignment  $(x,v)$ , we just have to restrict (mark appropriately) the tuples of one among the constraints having the variable  $x$  and propagate the changes to the adjacent constraints in the join tree. Using the CJBs lists, we can obtain the list of non-compliant tuples in the adjacent constraints and mark them as removed. During the process, if all the tuples corresponding to any CJB are removed, then the corresponding CJB is marked as removed and the effect is propagated. The Propagate function should not consider the direction of the edges in the join tree. Due to the properties of acyclic CSPs, such a propagation is enough to maintain minimality after unary assignments. The time complexity of Propagate is linear in the size of the JMCSP.

### 6.2 The Display Function

Given a JMCSP, with tuples in it marked as removed or not, the Display function just has to access all the tuples once and obtain the allowed values for the unassigned variables. This function also has a linear time complexity.

### 6.3 The Explain Function

Given a JMCSP, a partial assignment PA, a cost-function for each assignment in PA, and an assignment  $(x_i, v_i)$  for which explanation is required, *all* minimum explanations can be obtained as follows.

**Definition** A *topological ordering* of nodes (constraints) in a JMCSP, is an ordering such that, if  $c_1$  and  $c_2$  are two nodes in the JMCSP, then  $c_1$  precedes  $c_2$  in the ordering, if  $c_2$  can be reached from  $c_1$ .

A topological ordering can be obtained in linear time during the JMCSP creation process.

**Definition** The *valuation* is a function which given a PA and a corresponding JMCSP, maps a tuple in the JMCSP to a positive integer value or  $\infty$ . The valuation function is defined as follows:

1. Each element in the tuple corresponding to an unassigned variable will contribute nothing to the valuation.
2. Each element in the tuple that violates an assignment in PA will contribute the value given by the cost-function, for that assignment.
3. The cost for violation of  $(x_i, v_i)$  is  $\infty$ .
4. If the tuple contains a pointers to a CJB of an incoming directed edge, then the *min\_cost* field in the CJB will contribute towards the valuation.

After a valuation of a tuple is obtained, it is compared with the *min\_cost* value in the corresponding CJBs of its outgoing edges, if any. If the existing *min\_cost* is larger than the valuation of the tuple then, the *min\_cost* field is assigned the valuation of the tuple and the corresponding *min\_ptr* is updated to point to the tuple.

When the tuples of the constraint (node) without any outgoing edge are valuated, a tuple with minimum valuation in the constraint will be obtained. Recall that there can be only one such node.

Given these steps, a minimum explanation can be obtained as follows:

1. Assign  $\infty$  to all *min\_cost* fields in the JMCSP.
2. Following the topological ordering, select constraints and obtain valuation for their tuples.
3. All the *min\_ptr* pointed tuples and the cheapest tuple in the last node now gives a minimum explanation. The assignments in PA that are violated by those tuples is the minimum explanation.

Note that all minimum explanations can be obtained by remembering all the *min\_cost* valuated tuples in each node. Even with that facility, the Explain function will just have a linear time complexity.

## 7 Future Work

The tree-of-BDDs [Subbarayan, 2005] scheme has two types of space reduction. Potentially exponential space reduction due to tree-decomposition and another potential exponential space reduction due to BDD-based representation of each constraint. The JMCSPs just takes advantage of tree-decomposition and provides linear functions. The next step will be to adapt the join matching process for tree-of-BDDs. This seems plausible although not the generation of all minimum explanations. But, generation of one minimum explanation seems easy. Further research is required in this direction.

The join matching process has potential applications in general constraint propagation, model-based diagnosis, and database systems. The join matching might reduce the complexity of some of the functions in these applications.

## 8 Related Work

In [Fattah and Dechter, 1995; Stumptner and Wotawa, 2001], the authors have presented techniques for generating *minimal diagnosis* in model-based diagnosis systems. The problem of minimal diagnosis is very similar to our problem of generating minimum explanations. But they use sorting operations on constraints in the diagnosis process, and this increases the complexity of the operations by a logarithmic factor. Such a logarithmic factor might influence a lot, since the constraints in the real-world instances could have several thousand tuples in a single constraint, even before decomposition. After tree decomposition the number of tuples in the constraints of the resulting acyclic CSP is normally more than the number of tuples in the constraints before decomposition. Hence, it will take a significant amount of time to sort those constraints while generating explanations. For example, the *Renault car configuration* instance used in [Amilhastre *et al.*, 2002] has around 40,000 tuples in a constraint, even before decomposition. Also, in their complexity analysis a linear factor is *suppressed* and hence their minimal diagnosis algorithms will demand more time.

The JMCSPs, whose data structures remains relatively *static* during the explanation generation process, can hence be used in generating minimal diagnosis without any complex steps like sorting used in [Fattah and Dechter, 1995; Stumptner and Wotawa, 2001].

In [Madsen, 2003], the author uses some preprocessing techniques along with tree-decomposition and *cartesian product representation* for interactive configuration. The author was able to provide linear propagation functions but not explanations.

## 9 Conclusion

The JMCSPs, a data structure with linear configuration functions was presented. Experiments on real-life instances need to be done to empirically test the usefulness of JMCSPs. The join matching technique in JMCSPs can be combined with the compression capability of BDDs or automata [Amilhastre *et al.*, 2002] such that we get an additional decrease in space, while having linear configuration functions.

## Acknowledgements

Special thanks to Tarik Hadzic for his comments on a previous version of this paper. Erik van der Meer has independently developed a dynamic version of Tree-of-BDDs in his PhD thesis and discussions with him were helpful.

## References

[Amilhastre *et al.*, 2002] J. Amilhastre, H. Fargier, and P. Marquis. Consistency restoration and explanations in dynamic CSPs-application to configuration. *Artificial Intelligence*, 1-2:199–234, 2002.

[Array-Technology, 2005] Array-Technology. <http://www.array.dk/>, 2005.

[Bodlaender, 2005] Hans L. Bodlaender. Discovering treewidth. In *Proceedings of SOFSEM*, pages 1–16, 2005. Springer LNCS.

[Bryant, 1986] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 8:677–691, 1986.

[Configit-Software, 2005] Configit-Software. <http://www.configit-software.com>, 2005.

[Darwiche, 2001] Adnan Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):608–647, 2001.

[Dechter and Pearl, 1989] R. Dechter and J. Pearl. Tree-clustering schemes for constraint-processing. *Artificial Intelligence*, 38(3):353–366, 1989.

[Dechter, 2003] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[Fargier and Vilarem, 2004] H. Fargier and M-C. Vilarem. Compiling CSPs into tree-driven automata for interactive solving. *Constraints*, 9(4):263–287, 2004.

[Fattah and Dechter, 1995] Yousri El Fattah and Rina Dechter. Diagnosing tree-decomposable circuits. In *Proceedings of IJCAI*, pages 1742–1749, 1995.

[Goodman and Shmueli, 1982] Nathan Goodman and Oded Shmueli. The tree property is fundamental for query processing. In *Proceedings of PODS*, pages 40–48, 1982.

[Gottlob *et al.*, 2000] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124(2):243–282, 2000.

[Gyssens *et al.*, 1994] Marc Gyssens, Peter G. Jeavons, and David A. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66(1):57–89, 1994.

[Hadzic *et al.*, 2004] T. Hadzic, S. Subbarayan, R. M. Jensen, H. R. Andersen, J. Møller, and H. Hulgaard. Fast backtrack-free product configuration using a precompiled solution space representation. In *Proceedings of PETO conference*, pages 131–138, 2004.

[Madsen, 2003] J. N. Madsen. Methods for interactive constraint satisfaction. Master’s thesis, Department of Computer Science, University of Copenhagen, 2003.

[Stumptner and Wotawa, 2001] Markus Stumptner and Franz Wotawa. Diagnosing tree-structured systems. *Artificial Intelligence*, 127(1):1–29, 2001.

[Subbarayan *et al.*, 2004] S. Subbarayan, R. M. Jensen, T. Hadzic, H. R. Andersen, H. Hulgaard, and J. Møller. Comparing two implementations of a complete and backtrack-free interactive configurator. In *CP’04 CSPIA Workshop*, pages 97–111, 2004.

[Subbarayan, 2005] Sathiamoorthy Subbarayan. Integrating CSP decomposition techniques and BDDs for compiling configuration problems. In *Proceedings of the CP-AI-OR*. Springer LNCS, 2005.