

Integrating a Variable Ordering Heuristic with BDDs and CSP Decomposition Techniques for Interactive Configurators

Sathiamoorthy Subbarayan

Henrik Reif Andersen

Department of Innovation, IT University of Copenhagen, Denmark
sathi,hra@itu.dk

Abstract

The Binary Decision Diagram (BDD) based compilation schemes are quite suitable for representing configuration knowledge bases. It is well-known that the size of BDDs are very sensitive to the variable order. In this paper, we study the variable ordering problem of BDDs when they represent a configuration knowledge. Such a study is crucial to better understand the BDD-based compilation schemes. We introduce a small variant of a classical variable ordering heuristic, that proves empirically useful. It is surprising since such a heuristic is known to fail when BDDs are used in formal-verification. We show that a compilation scheme using CSP tree-decomposition is less dependent on the variable order. We also show that, for a good variable order the benefit of tree-decomposition, which reduces space, can drastically diminish.

Introduction

The complexity of made-to-order products keeps increasing. Examples of such products include personal computers, cars, and power-backup systems. Such products will be represented in the form of a product model. A product model will list the number of parameters (variables) defining the product, their possible values (domains) and the rules (constraints) 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. The parameters, values and rules of a configuration problem, correspond to the variables, domains and constraints in a CSP.

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 a customer. Such systems, called configurators, read a product model as input and guide the user in choosing one among the valid configurations as close as possible to his requirements. An interactive configurator 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.

The Binary Decision Diagram (BDD) (Bryant 1986) based symbolic CSP compilation technique (Hadzic *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 tree-of-BDDs approach (Subbarayan 2005), a combination of the BDD-based compilation technique and a CSP tree-decomposition technique, exploited the fact that configuration problems are specified in tree-like hierarchies. Experiments in (Subbarayan 2005) showed that decomposition techniques can drastically reduce the space required to represent configuration knowledges.

It is well-known that the size of BDDs are very sensitive to the ordering of the variables. The variable ordering problem (Meinel & Theobald 1998) of BDDs is well-studied in the area of formal verification. But, the ordering problem while representing a configuration problem has not been studied. So far the default variable order, the order in which variables are declared in the input is typically being used. A study on the variable ordering problem is crucial to better understand the compilation schemes. The usage of tree decomposition techniques, which were not used in formal verification, increases the need for a detailed study. The contributions of this paper are:

1. *Cluster Sifting*, a small variant of *sifting* (Rudell 1993), a classical variable ordering heuristic. In the experiments, cluster sifting resulted in up to 97% decrease in the size of a monolithic-BDD. The classical sifting heuristic did not help. This is surprising since cluster sifting uses *cluster ordering*. When BDDs are used in formal-verification cluster ordering is considered bad and *interleaved ordering* is preferred (Kam *et al.* 1998).
2. The observation that the size of a tree-of-BDDs is less

dependent on the variable ordering when compared with that of a monolithic-BDD.

3. Experimental results on several real-life instances, which show that efficient variable ordering can drastically reduce the size of a monolithic-BDD, while it can decrease the size of a tree-of-BDDs only by a small factor. In one instance, the usage of cluster sifting even results in a monolithic-BDD with size *smaller* than the size of the corresponding tree-of-BDDs (see Table 2). This gives an interesting trade-off between the two schemes.

The drastic reduction in space requirement is of great importance in online configuration applications, where one needs to send the configuration details through the internet, and also in embedded configuration, where one needs to embed the configuration details on a product itself, so that it could be reconfigured as and when required.

The rest of the paper is organized as follows. The two BDD-based compilation schemes are explained in the next section. Subsequently the variable ordering problem is discussed and the cluster sifting heuristic is presented. Experimental results, followed by some suggestions for future work and concluding remarks, finish this paper.

BDD-based Interactive Configurators

Due to space limitation we just explain how a BDD or a tree-of-BDDs can represent a configuration problem. The methods required for providing interactive configuration using these data structures are explained in (Hadzic *et al.* 2004; Subbarayan 2005).

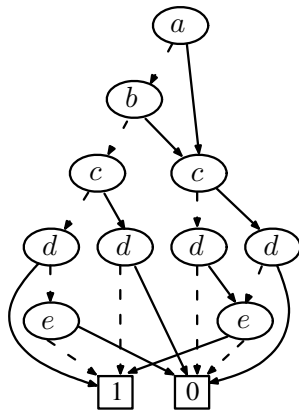


Figure 1: A Binary Decision Diagram.

A BDD (Bryant 1986) is a rooted directed acyclic graph with two terminal nodes marked 1 and 0, respectively. All the non-terminal nodes are associated with a Boolean variable. Each non-terminal node has two outgoing edges: *low* (dashed) and *high* (solid). The nodes in a BDD are ordered based on a linear variable order. BDDs can be used to represent Boolean functions.

A BDD for a given function can be obtained by standard composition functions on BDDs representing atomic elements of the function. Given an assignment to the variables in the function, there exists a unique path from the root

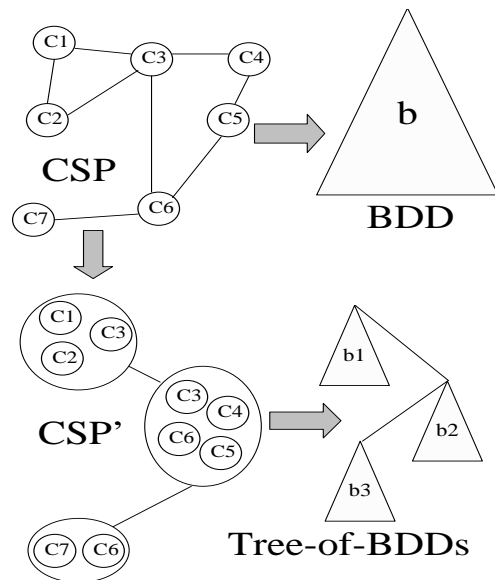


Figure 2: The two BDD-based compilation schemes.

node to one of the terminal nodes, defined by recursively following the high edge, when the associated variable is assigned true, and the low edge, when the associated variable is assigned false. If the path leads to the 1-terminal, then the assignment is a solution, otherwise not. Although the size of a BDD can be exponential in the worst case, the BDDs are small for many practical functions. Due to this BDDs have been successfully used in several research areas, including: verification, CSP, and planning. The size of the BDDs are very sensitive to the used variable ordering.

When solution space of a non-Boolean function needs to be represented by a BDD, each variable x_i with domain D_i will be represented by l_i Boolean variables, where $l_i = \lceil \lg |D_i| \rceil$. Each value of x_i will be represented by a unique combination of Boolean values for the corresponding Boolean variables. A BDD corresponding to any constraint can be obtained by the composition function on BDDs representing the atomic elements of the constraint. In case of CSPs, the conjunction of the constraints represents the solution space (SOL), and hence the conjunction function when applied to the BDDs obtained for all the constraints in a CSP will give a monolithic-BDD, representing the SOL. As l_i Boolean variables could represent 2^{l_i} values, in cases where $|D_i| < 2^{l_i}$, additional rules need to be added to maintain domain integrity.

In the monolithic-BDD based interactive configurator scheme (Hadzic *et al.* 2004), a monolithic-BDD will be compiled to represent the configuration knowledge. The tree-of-BDDs scheme (Subbarayan 2005), an improvement to this method, is based on the observation that configuration problems have tree-like hierarchies. Both the schemes are shown in Figure 2. In the figure, a CSP is represented by its constraint graph. The constraint graph of a CSP contains a node for each of its constraints and edges between

two nodes if the corresponding constraints share at least one variable. A CSP without cycles in its constraint graph can be efficiently solved (Dechter 2003).

The tree-of-BDDs scheme (Subbarayan 2005) uses tree-decomposition techniques to decompose a configuration problem (see Figure 2). The output of decomposition is CSP', in which a cluster of constraints in the original CSP form a constraint. Due to the properties of tree-structured CSP instances (Dechter 2003), BDDs obtained for each constraint in CSP' need not have to be conjoined into a monolithic-BDD. The tree-of-BDDs structure is enough to provide interactive configuration functionalities (Subbarayan 2005). The sum of the sizes of the BDDs in a tree-of-BDDs is potentially smaller than the corresponding monolithic-BDD. The experiments in (Subbarayan 2005) showed a drastic decrease in space requirements due to decomposition.

Cluster Sifting for BDD Variable Order

Variable ordering of BDDs are usually optimized using heuristics as finding the best variable order is a co-NP-Complete problem. Static variable ordering heuristics (Meinel & Theobald 1998) try to find a good variable order based on the topological relations in a given input instance, for which a BDD is going to be constructed. Dynamic variable reordering heuristics try to minimize a BDD, during or after the BDD construction. Mostly they are hill climbing procedures which will try to reorder a given variable order such that the BDD size decreases.

The sifting heuristic (Rudell 1993) is one of the best dynamic reordering heuristics. Given a BDD and its variable order, the sifting heuristic will select a variable and move it to a position in the current variable order, where the size of the given BDD is minimum. Other than the selected variable, the variable positions will not change. Similarly, the heuristic will try to move all the variables one-by-one, to their best possible position.

We initially implemented sifting heuristic, but it did not help in any of the configuration instances we tested. Then we implemented a small variant of sifting, which we call *Cluster Sifting*. In cluster sifting, all the Boolean variables representing a CSP variable will be clustered together (*cluster ordering*), and the cluster sifting will move the clusters in the ordering, instead of the individual Boolean variables.

In case of *interleaved ordering*, when Boolean variables encode a non-Boolean variable, the Boolean variables corresponding to the least significant values of the non-Boolean variables will be kept together in the variable order. Similarly, the Boolean variables corresponding to other significant positions will be grouped together in the order. The interleaved ordering works well in case of arithmetic constraints like $a+b=c$. If the three variables $\{a,b,c\}$ are each encoded by k Boolean variables, then all the Boolean variables of $\{a,b,c\}$, corresponding to each one among the k significant positions will be grouped together.

Cluster ordering performed well in the configuration instances we tested. This is surprising as it is unconventional. As in (Kam *et al.* 1998), when using BDDs for formal verification it is conventional to consider cluster ordering as bad,

and interleaved ordering is preferred. In case of configuration problems, most of the constraints are non-arithmetic constraints. This might be a reason for better performance of cluster ordering in configuration problems.

```

CLUSTERSIFT()
  vo:=GetVariableOrder()
  for i:=1 to n
    for j:=1 to n
      if vo[j] = i
        min_pos:=j; old_pos:=j
        min_size:=SizeOf(SOL); break
    for j:=old_pos to n - 1
      vo[j]:=vo[j + 1]; vo[j + 1]:=i
      SetVariableOrder(vo); size:=SizeOf(SOL)
      if size < min_size
        min_pos:=j + 1; min_size:=size
    for j:=n to old_pos + 1
      vo[j]:=vo[j - 1]
  vo[old_pos]:=i
  for j:=old_pos to 2
    vo[j]:=vo[j - 1]; vo[j - 1]:=i
    SetVariableOrder(vo); size:=SizeOf(SOL)
    if size < min_size
      min_pos:=j - 1; min_size:=size
  for j:=1 to min_pos - 1
    vo[j]:=vo[j + 1]; vo[j + 1]:=i
  SetVariableOrder(vo)

```

Figure 3: The Cluster Sifting Heuristic.

We assume that given a BDD and a new variable order, the BDD-package will be able to give a new BDD with the given variable order. Most of the BDD-packages have such a facility. The algorithm for cluster sifting is shown in Figure 3. In the algorithm, vo is an array of CSP variables. The position of each CSP variable in the array reflects their ordering in the BDD. The number of variables is represented by n . The function *GetVariableOrder()* will return vo . The function *SizeOf(b)* will return the number of nodes in the BDD b . In case b is a tree-of-BDDs, it will return the sum of the sizes of BDDs in b . The *SetVariableOrder()* function will take a vo as input and change the variable order of the monolithic-BDD (tree-of-BDDs) representing the solution space to vo . SOL is the BDD (tree-of-BDDs) representing the solution space.

The outermost loop in the algorithm is to move (sift) all the n CSP variables to their best possible position. There are five inner loops. The first one locates the current position of a variable i in the order. The second one moves i to the last position in the order, while remembering the cheapest visited position. The next two loops are to move i to the first place in the order. The last loop moves i to the best found position. Finally, the variable i is set to that position.

After a call to the cluster sift method, if the size of SOL has decreased more than 1% of the initial size, the cluster sift method is invoked again. This is repeated until there is no decrease or less than 1% decrease in the size.

The *Block Sifting* heuristic in (Meinel & Slobodová 1997) is quite similar to cluster sifting. However, block sifting is quite complex as it looks for blocks of functionally dependent Boolean variables. In case of cluster sifting, we do not look for any functional dependency between Boolean variables. We just group all the Boolean variables corresponding to a single CSP variable as a cluster.

Less Dependency of Tree-of-BDDs

From the structure of the tree-of-BDDs we can observe that the tree-of-BDDs are less dependent on the variable order than the monolithic-BDDs. This is illustrated in Figure 4. Suppose, we have a CSP with variables $\{a, b, c, d, e, f, g\}$ and the initial variable order is (a, b, c, d, e, f, g) . Let tree-decomposition results in three BDDs in the corresponding tree-of-BDDs, with the variables appearing in the nodes as shown. Let the variable order be changed, by moving variable g to the front, to (g, a, b, c, d, e, f) . This affects the size of only one BDD in the tree-of-BDDs, as g occurs only in it. In this sense the variable order change only results in a *local* change in the size of the tree-of-BDDs. But, in case of the corresponding monolithic-BDD, as all the variables appear in a single BDD the change is *global*.

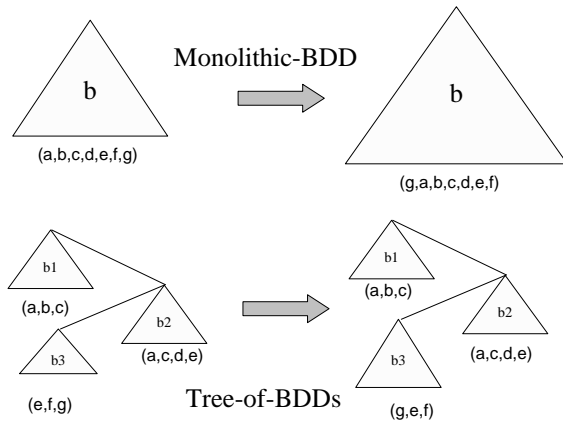


Figure 4: Effect of variable order.

Also, due to the tree-decomposition properties, the three variables $\{e, f, g\}$ appearing in BDD $b3$ are highly related in the CSP. It is a general rule of thumb to keep related variables close in the order, to reduce the size of BDDs. Although the variable g is moved away from e and f in the *global variable order*, in the *partial order* corresponding to BDD $b3$ they still remain close and the change in the size of $b3$ will be small, when compared to that of the monolithic-BDD. This implies that, when the monolithic-BDD size could increase (decrease) a lot with a change in the variable order, the corresponding increase or decrease in the size of the tree-of-BDDs will be relatively small. This shows the less dependency of tree-of-BDDs on the variable order. Hence, for a poor variable order the tree-of-BDDs will be a lot smaller than the corresponding monolithic-BDD, while

for a good variable order the difference should diminish. We observe this empirically in the next section.

Experimental Results

Experiments are done by implementing the Cluster Sifting heuristic, on top of CLab (Jensen 2004) and iCoDE (iCoDE 2004). CLab is an interactive configurator based on the monolithic-BDD scheme and the tree-of-BDDs scheme is available in iCoDE. In the experiments, first a monolithic-BDD (tree-of-BDDs) will be obtained using the initial variable order (IVO) and then cluster sifting will be done. IVO is the order in which variables are declared in the input.

A Pentium Xeon 3.2 GHz machine with 4GB RAM and 1MB L2 Cache is used in the experiments. Five configuration instances are used in the experiments: PC, Bike, Renault, psr-1-32-1 and psr-1-32-2. PC, Bike and Renault refer to computer, bike and car configuration problems, respectively (CLib 2004). Renault was the only instance used in the experiments of (Amilhastre, Fargier, & Marquis 2002) and it is quite large, the input file size is around 23 Megabytes, and it has around 200,000 tuples in 113 constraints. The other two instances are power supply restoration problems modelled as configuration instances (Sonne & Jensen 2005). The characteristics of the benchmarks are listed in Table 1. The column $\sum |D_i|$ refers to the sum of the domain sizes of the variables in the instance. Arity of a constraint is the size of its scope. The column $Max |a|$ refers to maximum arity of the constraints. The column $\sum |a_i|$ refers to the sum of the constraint arities. The column $|SOL|$ refers to the number of solutions.

The effect of the cluster sifting is shown in Table 2. In the table, *Compile* refers to the compilation time. The column *#nodes* refers to the number of nodes in the BDD for initial variable order (IVO). The column *ClusterSift* refers to the time taken by the cluster sifting heuristic. The column *#nodes'* refers to the number of nodes after cluster sifting. The size of a tree-of-BDDs is the sum of the sizes of the BDDs in each node of the tree. The column *#nodes''* refers to the number of nodes in the monolithic-BDD after using the variable order obtained by cluster sifting on the corresponding tree-of-BDDs. The values in the table clearly shows that, for IVO, the size of tree-of-BDDs are significantly smaller when compared to the corresponding sizes of the monolithic-BDDs. But after cluster sifting, the difference between them decreases. In case of the PC instance, after cluster sifting, the size of the tree-of-BDDs is even larger than the size of the monolithic-BDD. In case of the Renault instance, the cluster sifting decreases the size of the monolithic-BDD by 97%.

Although the time taken for cluster sifting is quite large, this is acceptable as a configuration problem instance will not change quite often. The compilation needs to be done only when the instance changes. Several interactions can be handled using the same monolithic-BDD (tree-of-BDDs). Even when it changes, the changes will be incremental and the basic structure of the instance will not change, and hence the good variable order will typically remain as a good one.

Response timings for simulation of 10000 random interactions in both the compilation schemes, using IVO and

Table 1: Benchmark Properties.

Benchmark	#Variables	$\sum D_i $	#Constraints	Max $ a_i $	$\sum a_i $	$ SOL $
PC	45	383	36	16	107	1.19×10^6
Bike	34	210	76	6	141	8.07×10^8
Renault	99	402	113	10	588	2.84×10^{12}
psr-1-32-1	110	258	222	9	913	1.72×10^{11}
psr-1-32-2	110	296	222	9	913	1.76×10^{13}

Table 2: Effect of Cluster Sifting in both the BDD-based schemes. Time values are in CPU seconds.

Benchmark	Monolithic-BDD					Tree-of-BDDs			
	Compile	#nodes	ClusterSift	#nodes'	#nodes''	Compile	#nodes	ClusterSift	#nodes'
PC	0.11	16494	1635	1312	1568	0.19	4458	2740	1477
Bike	0.1	10986	523	1966	10455	0.2	3391	267	1494
Renault	119	455796	95160	13248	32792	77	17602	22092	6802
psr-1-32-1	0.46	56923	15801	15894	16752	4	8917	18247	3918
psr-1-32-2	2	246775	34287	46512	46645	9	22101	34445	7711

Table 3: Response timings for 10000 random interactions. Time values are in CPU milli-seconds.

Benchmark	Monolithic-BDD				Tree-of-BDDs			
	ART	ART'	WRT	WRT'	ART	ART'	WRT	WRT'
PC	16	14	40	20	8	8	10	10
Bike	14	13	20	10	7	7	10	10
Renault	119	16	530	30	9	8	30	20
psr-1-32-1	17	16	60	30	11	8	40	20
psr-1-32-2	38	20	620	70	17	9	110	30

cluster sifting is shown in Table 3. ART refers to the average response time. WRT: worst response time. ART' and WRT' refer to the corresponding values obtained using cluster sifting. The results show that after cluster sifting the difference between the timings for both the compilation schemes decreases.

Figure 5 shows the convergence of cluster sifting heuristic in both the schemes. For the monolithic case, the cluster sifting heuristic converges quite quickly in the beginning, and then slowly until it reaches the stop criterion. For the tree-of-BDDs case, when compared with the difference in the monolithic case, there is not much difference between the initial size and the final size.

To find out the dependency of the two compilation schemes on the variable order, 1000 random variable orders are generated for each instance. Experiments on both the schemes are done, using each one of them as initial variable order for the corresponding instance. Results are reported in Table 4. The results for the last two instances are not reported as all of the variable orders resulted in huge memory requirement and had to be aborted. The table clearly shows that, the tree-of-BDDs method is less dependent on the variable order. It varies a little with different variable orderings, while the monolithic-method varies a lot. Especially in case of the Renault instance, the monolithic-method has to abort in 540 cases, while the sizes of the corresponding tree-of-BDDs varied a little. The statistics for Renault was obtained from the results of the successful 460 experiments.

As mentioned before, the classical sifting heuristic did not help in any of the configuration instances. We used

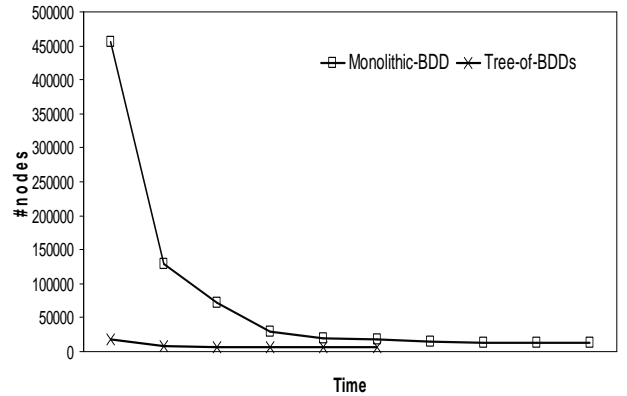


Figure 5: Convergence on Renault.

Configit-Developer (Configit Software 2005) tool to generate two heuristic variable orders for the Renault instance. The heuristics are MST (Minimum spanning tree) and DFS (Depth first search). Both the heuristics use the structure of the input configuration instance and give a variable order. As shown in Table 5 both the MST and DFS did not result in a variable order as good as cluster sifting.

Table 4: Statistics on experiments with 1000 random variable orders.

Statistic on #nodes	PC		Bike		Renault	
	Monolithic	Tree	Monolithic	Tree	Monolithic	Tree
Average	598337	29080	87484	10705	951968	23669
Max	4517525	71996	525532	27517	2857552	38344
Std.Dev	618876	12906	79448	4125	381705	4801
#Aborts	0	0	0	0	540	0

Future Work

It can be observed from Figure 4 that in the case of tree-of-BDDs, cluster sifting only has to consider *local variable order*. In the example shown in the figure, it does not matter whether variable g is moved to the front or between a and b . In both cases the local variable order for each node in the tree-of-BDDs remains the same. This observation can be used to improve the cluster sifting heuristic for the tree-of-BDDs.

Experiments on *domain reduction* methods for cluster sifting. For a given configuration instance, some of the values in the domain of some of its variables can be deleted, such that the monolithic-BDD size becomes small. This might reduce the time taken for cluster sifting. The cluster sifting can then be used to obtain a good variable order, which is most likely good for the original instance without domain reduction. This intuition is based on the observation that deleting some of the values still preserve the overall structure of the configuration instance.

Cluster sifting works on built BDDs. Instead of this the heuristic could be incorporated into the BDD building process itself. This is how most of the BDD-packages implement variable ordering heuristics.

Conclusion

Cluster sifting, a variable ordering heuristic, was introduced and its effect on two BDD-based compilation schemes were studied. It was shown that the tree-of-BDDs are less dependent on the variable order than the corresponding monolithic-BDDs.

The advantage of monolithic-BDDs is that the entire configuration knowledge is represented by a single BDD. Hence, some advanced features like *minimum explanations* (Subbarayan 2005) can be efficiently implemented in monolithic-BDDs. The disadvantage of them is their huge dependency on the used variable order. Similarly the advantage of tree-of-BDDs is their less dependency on the variable order, while at the overhead of having the knowledge spread across several nodes in a tree-of-BDDs. The cluster sifting heuristic gives a good tradeoff between them. Given enough time, the cluster sifting could be used to obtain a good variable order and hence a small monolithic-BDD. Otherwise, the tree-of-BDDs can be built to provide basic interactive configuration facilities.

In the cases where the initial variable order is so bad that a monolithic-BDD cannot be built, the tree-of-BDDs could be used to obtain a good variable order. As shown in the

Table 5: Heuristics on Renault.

Heuristic	#nodes
ClusterSift	13248
MST	707463
DFS	407382

column *#nodes* of Table 2, it will be a good variable order for the corresponding monolithic-BDD as well.

References

- Amilhastre, J.; Fargier, H.; and Marquis, P. 2002. Consistency restoration and explanations in dynamic CSPs-application to configuration. *Artificial Intelligence* 1-2:199–234.
- Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* 8:677–691.
- CLib. 2004. www.itu.dk/doi/VeCoS/clib/.
- Configit Software. 2005. configit-software.com.
- Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann.
- Hadzic, T.; Subbarayan, S.; Jensen, R. M.; Andersen, H. R.; Møller, J.; and Hulgaard, H. 2004. Fast backtrack-free product configuration using a precompiled solution space representation. In *International Conference on Economic, Technical and Organizational aspects of Product Configuration Systems*, 131–138.
- iCoDE. 2004. www.itu.dk/people/sathi/icode/.
- Jensen, R. M. 2004. CLab: A C++ library for fast backtrack-free interactive product configuration. In *Proceedings of CP-2004*, 816. <http://www.itu.dk/people/rmj/clab/>.
- Kam, T.; Villa, T.; Brayton, R. K.; and Sangiovanni-Vincentelli, A. L. 1998. Multi-valued decision diagrams: Theory and applications. *Multiple-Valued Logic: An International Journal* 4(1-2):9–62.
- Meinel, C., and Slobodová, A. 1997. Speeding up variable reordering of OBDDs. In *Proceedings of ICCD*, 338–343.
- Meinel, C., and Theobald, T. 1998. *Algorithms and Data Structures in VLSI Design*. Springer.
- Rudell, R. 1993. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, 139–144.
- Sonne, L. E., and Jensen, R. L. 2005. Power Supply Restoration. Master’s thesis, IT University of Copenhagen.
- Subbarayan, S. 2005. Integrating CSP decomposition techniques and BDDs for compiling configuration problems. Technical report, IT University of Copenhagen. www.itu.dk/people/sathi/icode.pdf.