Combining Binary Decision Diagrams and Backtracking Search for Scalable Backtrack-Free Interactive Product Configuration

Andreas Hau Nørgaard, Morten Riiskjær Boysen, Rune Møller Jensen
IT University of Copenhagen, Denmark
ahn@itu.dk, boysen@itu.dk, rmj@itu.dk

Peter Tiedemann
Configit A/S
pt@configit.com

Abstract

This paper demonstrates how to create approximations of a configuration problem using BDDs to improve performance over a pure search-based configurator, for problems that are intractable to represent by a monolithic BDD. The paper demonstrates several ways to build the approximations and it shows that using search results for building and improving the approximations leads to a significant performance gain.

1 Introduction

Configuration Problems (CPs) occur whenever a product that can be configured needs to be tailored to specific requirements. Examples of this ranges from buying t-shirts or computers online, to configuring large wind-turbines all the way up to large data centers. These problems are a prime target for AI techniques, either because their complexity is so high that even a trained user cannot oversee all requirements or because the user is untrained an must be guided e.g. during a purchase in an online store. Thus in the case of the online store, solving configuration problems is important because it allows companies to use less resources for support, thus reducing their cost and because it can give them a competitive advantage by making the purchase as simple and straightforward as possible. In the case where a trained person needs to configure large machineries, configuration becomes a question of increasing the productivity of the operator. Perhaps even more importantly, configuration technology aids in preventing costly invalid configurations, by catching errors during the configuration instead of after production has begun, where the cost of fixing the mistakes can be very high.

1.1 Interactive Configuration

A special form of the Configuration Problem is *Interactive Product Configuration* (IPC). For these problems a user is interfacing directly with the configurator and needs to see the consequences of the choices he makes. This is in contrast to an automated system, where e.g. a partial assignment is given and the configurator then has to complete the product. The requirements of interactive configuration are:

Complete Meaning that all valid configurations can be reached by the user. If the configurator is not complete, certain valid product configurations cannot be configured. This is very unfortunate, e.g. if the configurator is used to configure products in an online store, as it would mean that some valid product configurations cannot be sold.

Backtrack Free Whenever the user selects a value, all values that cannot extend the current partial assignment to a solution will be removed. This means a partial assignment is *always* extendable to a solution, and hence the user never needs to backtrack. This is not a strict requirement, but it is a very desirable property of an interactive configurator since backtracking can be very tedious to the user.

Fast Response Times It is important to display the consequences of an assignment to the user as fast as possible, so the user does not grow impatient with the configurator. How fast this must happen depends on the type of user and the environment the configurator is used in.

Arbitrary Order of Assignments The user must be allowed to make assignments to the variables in any order the user likes.

1.2 Search

Using backtracking search to solve various kinds of CSPs (not just configuration) is a commonly used technique. When performing the search, the *solver* chooses a variable and branches on it, thereby obtaining a reduced problem, after which the solver chooses a variable again to branch on and so on until either the CSP is proven unsatisfiable or a solution has been found. Much work has been put into improving the basic search by using consistency techniques and heuristics to reduce the search tree. These techniques are invaluable in a modern solver. Unfortunately, since the search tree is potentially exponential in size, these techniques gives very little guarantees on the performance regarding time use.

1.3 Compiled Representations

Another way to solve a CP is through *compilation*: The entire set of solutions to the problem is stored in some compact form. This is the preferred technique for interactive configuration, since the representation of solutions only has to be built once and can be shared by users afterwards. There are many ways to store the solution space, including *Binary Decision Diagrams* (BDDs) [Bryant, 1986], *Multi-valued Decision Diagrams* (MDDs) [Kam *et al.*, 1998] and *Cartesian Product Tables* (CPTs) [Møller, 1995].

There exists polynomial time algorithms to do valid domain calculations on these data structures, thus giving good guarantees on the performance. However, there is no such thing as free lunch: since CPs are NP-complete, the compilation phase might take exponential time, and possibly even worse, the output might take up exponential space. It can be shown that BDDs and MDDs require exponential space for the `alldifferent` constraint [van Hoeve, 2001], which is vital in modeling configuration problems involving placement.

The main contribution of this paper is to show a way to implement a complete, backtrack free interactive configurator capable of handling configuration problems better than a pure search-based configurator or a BDD-based configurator. In this paper, we show that it is possible by using a combination of both techniques.

The results were obtained by combining the Gecode solver [Schulte *et al.*, 2009] with CLab [Jensen, 2004], which is a BDD-based configuration library. We used BDDs for storing approximations of the configuration problem that makes it possible to *eliminate* some of the searches performed by the solver in the valid domain computation. Our computational results are on industrial data from Configit A/S [Andersen and Hulgaard, 2007] and show that a substantial lower average response time of the configurator can be achieved in this way compared with a pure BDD-based or a pure search-based approach.

The idea to use approximations to speed up interactive configurations was first presented in [Tiedemann, 2008], but the author did not provide any tests or implementation. A previous study in [Subbarayan *et al.*, 2004] compared a purely search-based configurator with a BDD-based configurator, showing that the latter performed better in most cases. However, the study did not involve global constraints and all problem instances could be represented in a BDD. Furthermore, some ideas was presented in the study for creating an efficient search-based configurator, which are extended in this paper. Previous research has been conducted in extracting no-goods from constraints represented as BDDs [Subbarayan, 2008], but the paper does not mention the use of search results for building the BDDs. It focuses entirely on extracting small no-goods from a static BDD. In [Subbarayan *et al.*, 2006] the authors used BDD to build a hybrid SAT solver. However, the work does not include configuration problems, nor does it use BDDs to store the results of time-consuming searches. Thus, the main contributions of this paper is the implementation and test of a hybrid configurator using BDDs for good- and no-good recording (good-recording is described in [Cheng and Yap, 2006]) and a solver for problems that is intractable to represent entirely as a BDD.

The remainder of this paper is organized as follows: In section 2, we present the concept of interactive configuration and show two different ways of implementing it. One uses a search-based solver, the other uses BDDs. In section 3 we show how to combine a solver with BDDs to obtain better performance than what is possible if using each technique alone. In section 6 we show the empirical results obtained, and finally, section 7 concludes on the results. These results are obtained on a data center configuration example provided by Configit [Andersen and Hulgaard, 2007] that models the configuration of a large-scale data center.

2 Backtrack Free Interactive Configuration

A configuration problem C is a triple (X, D, F), where

- X is a set of variables $x_1, x_2, \ldots, x_n$
- D is the Cartesian product of their finite domains $D = D_1 \times D_2 \times \ldots \times D_n$
- $F = \{f_1, f_2, \ldots, f_m\}$ is a set of propositional formulas over atomic propositions $x_i = v$, where $v \in D_i$, specifying the conditions that the variable assignments must satisfy. Each formula is inductively defined by $f \equiv x_i = v \mid f \wedge g \mid f \vee g \mid \neg f$

An interactive product configurator (IPC) enables the configuration process as described in section 1.1. The main task of an IPC is to compute the set of *valid assignments* VD for a configuration problem, where $VD = \{VD_1, VD_2, \ldots, VD_n\}$ and $VD_i \subseteq D_i$. These are the assignments that are guaranteed to be extendable to a solution. Once the valid domains have been computed, the user can make a valid assignment. These two steps are repeated until the product has been configured (all variables has been assigned), see Algorithm 1.

Algorithm 1 An informal definition of the IPC algorithm

```
1: procedure IPC
2:     read and process configuration problem
3:     while not all variables assigned do
4:         $VD \leftarrow$ COMPUTEVALIDDOMAINS
5:         user makes a valid assignment
```

2.1 Search-based Configuration

The simplest way, albeit very naive, to calculate the valid domains, using a search based solver, is shown in Algorithm 2. This algorithm enumerates all possible assignments $(x_i = v_{ij})$ where $\{(x_i, v_{ij}) \mid x_i \in X, v_{ij} \in D_i\}$. An assignment is added to the existing configuration problem where after this augmented problem is tested for satisfiability. If the augmented problem is satisfiable, it is known that $v_{ij} \in VD_i$. This step is repeated for all possible assignments. This method performs a search for all (x_i, v_{ij}) that *might* be valid. Hence it performs $\sum_{i=1}^{n} |D_i|$ searches. In section 3 we describe several ways to improve this initial algorithm.

2.2 BDD-based Configuration

A binary decision diagram (BDD) is a rooted directed acyclic graph. A BDD has one or two terminal nodes[1], labeled 1 or 0,

[1]A terminal node has out-degree zero

Algorithm 2 A naive way to determine the valid domains

```
1: procedure CVD-NAIVE($C$)
2:     $VD' \leftarrow$ PROPAGATE($C$)
3:     for all $x_i \in X$ do
4:         $VD_i \leftarrow \emptyset$
5:         for all $v_{ij} \in VD'_i$ do
6:             if $C|x_i = v_{ij}$ is satisfiable then
7:                 $VD_i = VD_i \cup v_{ij}$
```

and a set of variable nodes. The terminal node labeled 0 is denoted by T_0 and the terminal node labeled 1 is denoted by T_1. Each variable node is an internal node in the BDD and has exactly two outgoing edges marked *low* and *high*. A BDD represents a boolean function f on a set of n boolean variables $f : \mathbb{B}^n \rightarrow \mathbb{B}$. The value of the boolean function, given an assignment of the variables, can be found by recursively traversing the BDD. The traversal begins at the root, and continues to a terminal node. Whenever a variable is assigned to true, the high branch of the corresponding node along the path is taken. If a variable is assigned to false, the low branch of the corresponding node is taken. If the path ends at a terminal labeled 1, the assignments means the value of the function is true. If the path ends at a terminal labeled 0, the value of the function is false.

A reduced ordered binary decision diagram (ROBDD) [Bryant, 1986] is a BDD with the two additional properties of being ordered and reduced. A BDD is said to be ordered when all paths from the root node to a terminal node respect a given variable ordering, meaning that the variables associated with the nodes will be met in the order defined. A BDD is said to be reduced when all nodes where the low and high branches leading to the same node are removed and when all nodes are unique. A node is unique if no other node exist that has the same associated variable and branches to the same destinations on the high and low branches respectively. If such a duplicate node exist, it can be removed by collapsing the two nodes into a single node. In the rest of this paper we only use ROBDDs, and since it is a De facto standard to use the abbreviation BDD to mean a reduced ordered binary decision diagram, we will follow the convention and consequently write BDD from now on when we refer to a reduced ordered binary decision diagram.

BDDs have been widely used in verification, but it was later discovered that they are also well suited for configuration problems [Hadzic *et al.*, 2004]. However, a configuration problem can have variables with finite integer domains whereas a BDD only has boolean variables. Fortunately, an integer variable x_i can be encoded efficiently in a BDD using $k_i = \lceil log_2|D_i|\rceil$ boolean variables $x_i^0, \ldots, x_i^{k_i-1}$. Furthermore, these variables are places in *layers*, so all boolean variables encoding the same finite domain variable are placed in the same layer and all finite domain variables define a unique layer. The BDD nodes comprising the layer i are denoted by V_i.

Example: A simple example of a CP is shown in Figure 1. The constraints corresponds to the relations $x_1 < x_2$, $x_1 < x_3$ and $x_2 \neq x_3$.

$$
\begin{aligned}
&X = \{x_1, x_2, x_3\} \\
&D = \{\{1,2,3\},\{1,2,3\},\{1,2,3\}\} \\
&C = \{((x_1,x_2),\{(1,2),(1,3),(2,3)\}), \\
&\qquad ((x_1,x_3),\{(1,2),(1,3),(2,3)\}), \\
&\qquad ((x_2,x_3),\{(1,2),(1,3),(2,1),(2,3),(3,1),(3,2)\})\}
\end{aligned}
$$

Figure 1: A simple CP

This simple example can be represented by a BDD as seen in Figure 2.

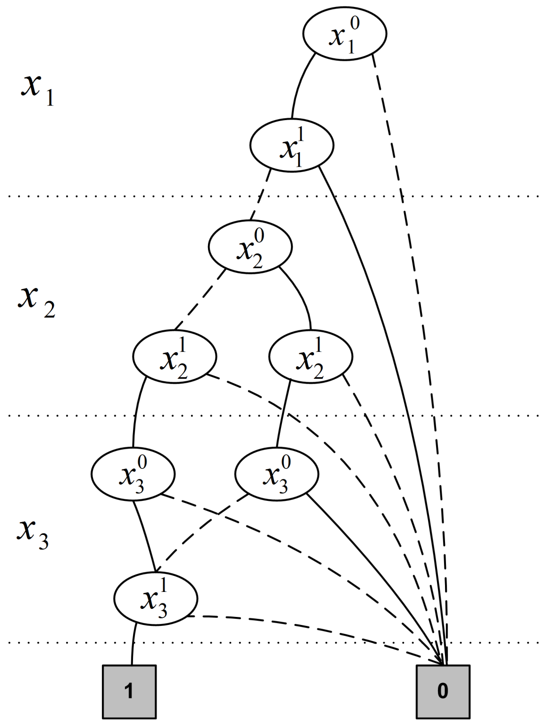

Figure 2: The CP from Figure 1 encoded as a BDD with high edges shown as solid lines and low edges shown as dashed lines. The variable ordering is $x_1^0 < x_1^1 < x_2^0 < x_2^1 < x_3^0 < x_3^1$. The layers are shown as the horizontal dashed lines.

2.3 BDD-based Valid Domain Computation

In order to use BDDs for IPC we need to be able to perform assignments and compute the valid domains. Assignments can be made by using the standard APPLY BDD operation by conjoining the BDD representing $x_i = v$ onto the BDD G_1 representing the current solution space restricted by the assignments made so far in the configuration process. The complexity of the restriction operation for variable x_i in BDD G_1 is thus $O(|G_1| \cdot \lceil log_2 |D_i|\rceil)$.

The *Compute Valid Domains* (CVD) operation determines from a BDD representing a configuration problem what values that are guaranteed to be in the solution space and extensible to a full assignment. Let VD_i denote the valid domain for variable x_i, then $VD_i \subseteq D_i$. Thus, any assignment

$\{(x_i = v) \,|\, v \in \mathit{VD}_i\}$ will never lead to the user backtracking.

The valid domain computation works by *probing* the layers. Each value $v \in D_i$ is tested by traversing the i'th layer from all nodes in V_i with incoming edges from the preceding layers until support has been found or all nodes in a layer has been probed. If all traversals for v ends in T_0, there is no support for v, so $v \notin \mathit{VD}_i$. To avoid probing the same nodes while checking for support for v, all nodes are checked if they have already been probed with v. If a node has already been checked, the traversal is stopped, since the traversal will end in T_0 (Since it has been probed earlier, that probe failed. Otherwise support would have been found and the probing for v would stop). The checking ensures that a node in v_i is only checked once for each $v \in D_i$. Thus, the worst case complexity of the compute valid domains operation over a BDD is $O(\sum_{i=1}^{n} |V_i| \cdot |D_i|)$.

Example: Assume a valid domain computation is performed on the BDD from Figure 2 and that the algorithm is about to test the valid domain of x_3. First, the value 1 is tested. There are two nodes with incoming edges from preceding layers. The probing starts from the left-most node with the binary encoding of 1. In this probing, the traversal ends up in T_0 after having gone though the node labeled x_3^1. The right-most node in the layer is then probed, but with the binary encoding of 1, this leads directly to T_0. Thus, $1 \notin \mathit{VD}_3$. When probing for support for 2, the left most node is again chosen as the start, but this leads directly to T_0. The right-most node is then used to start a traversal, and after passing through the node labeled x_3^1, the traversal ends in T_1, so $2 \in \mathit{VD}_3$. When checking for the value 3, the traversal beginning from the left-most node ends in T_1, so there is also support for 3. The result of the probing is that $\mathit{VD}_3 = \{2, 3\}$.

3 BDD and Search-based Hybrid Configurator

To be able to make a fair comparison between the performance of a search-based configurator and our hybrid configurator, the algorithm behind the search-based configurator needs to be improved. In the following we will present a series of improvements to the naive algorithm shown in Algorithm 2.

Only a very small part of the information provided by the solver in Algorithm 2 is actually used: namely whether a single value is part of the valid domain of a variable or not. The search result has a lot more information than that: All the assignments in the search result are part of the valid domains of there respective variables. The naive algorithm can therefore be improved in two ways: First the result is traversed and all assignments are stored as part of the valid domains. Secondly, before a search is started, it is checked whether the value v_{ij} has already been verified to be part of VD_i. If it is, the search is simply skipped.

Additionally we can use former valid domain results to speed up the valid domain computation, since when an assignment is made, the solution space can never grow meaning that $S_{|x_i=v_{ij}} \subset S$. This implies that a search is redundant if an assignment $(x_i = v_{ij})$ has been discovered as invalid by a previous search but has not been pruned by propagation yet, because of the non-increasing property of the solution space. This information can be fed back to the solver and propagation mechanism by posting the unary inequality constraint $x_i \neq v_{ij}$ whenever a search fails. The value v_{ij} is thus removed from the current domain of x_i and hence augments the solver with information that propagation alone could not detect. This can improve propagation and increase the search performance for other variables.

Finally it is an invariant in the configurator that once the domain has size 1, it cannot shrink any more. If it could, the configurator would not be backtrack free. It is therefore possible to skip the search, if all but the last value v_i^{last} in VD_i' has been found to be invalid. Therefore $x_i = v_i^{\text{last}}$ must be a valid assignment.

3.1 Hybrid Configurator

The search preventing hybrid configurator utilizes a combination of BDDs and search-based techniques. The basic idea is to avoid as many searches as possible by using BDD-based approximations.

Over-Approximations and Under-Approximations

An over-approximation of a CP with solution set S, is a CP with solution set $S_o \supseteq S$. An under-approximation of a CP with solution set S, is a CP with solution set $S_u \subseteq S$. Given an over-approximation CP_o of a CP and a partial assignment (PA), CP_o can be used to determine if PA is not extendable to a solution in CP. However, it cannot be used to determine whether it is extendable to a solution. Conversely given an under-approximation CP_u of a CP, CP_u can be used to determine if a partial assignment PA is extendable to a solution in CP, but CP_u cannot be used to determine if PA is not extendable to a solution. Thus, if the two approximations are used together, a search is only needed when neither approximation is able to determine whether PA is definitely extendable to a solution or definitely not.

This relation is shown in Figure 3. The picture shows the Cartesian product of the domains of the variables in a CP. The grey area to the left of the curved line represent the solution set and the white area to right of the curved line represent the non-solutions (the set of full assignments that violate one or more constraints). The box with the bold dashed line represents the under-approximation and the box with the bold solid line represents the over-approximation. As the drawing shows we need to perform a search for elements in the set $S_o \backslash S_u$.

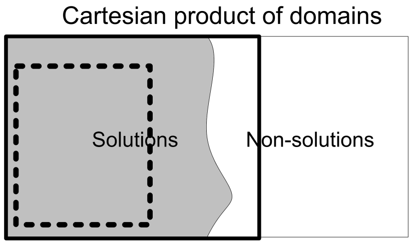

Figure 3: The relation between the solution space and the over- and under-approximation for a CP.

4 Using the Approximations

As described in section 2.1 a search-based configurator uses a two-step approach, by first propagating and then searching whenever an assignment has been made and we want to find the new valid domains. Using an over-approximation changes the two-step approach into what we could call a two-and-a-half-step approach because we need to utilize the over-approximation after the propagation step in order to avoid the search step as often as possible. As mentioned, we can avoid a search for all values not in the valid domains of the over-approximation restricted to the current partial assignment, since these will clearly not be in the domain of the CP. Furthermore, all values in the valid domain of the under-approximation restricted to the current partial assignment can be added to the valid domains of the CP before the search phase because $S_u \subseteq S$.

The valid domain computation including all the optimization from section 3 and the approximations can be seen in Algorithm 3, where BDD^o is the BDD representing the over-approximation and BDD^u is the BDD representing the under-approximations.

4.1 Constructing an Over-Approximation

Given a CP with solution set S, an over-approximation of this is also a CP (which we call CP$'$). Since we represent our over-approximation by a BDD, a simple way to construct CP$'$ is by removing all constraints from the original CP that are intractable to represent in a BDD. This would make CP$'$ less restricted than the original CP, and therefore $S \subseteq S_o$ which was the requirement.

4.2 Constructing an Under-Approximation

We construct the under-approximation CP$''$ by putting additional constraints on the CP we are approximating. As mentioned, `alldifferent` constraints puts an exponential lower bound on the number of nodes in a BDD. Since the under-approximation needs to be at least as strict as the original CP we cannot remove the `alldifferent` constraint from the under-approximation. We have therefore investigated what additional restrictions to add to a CP that contains an `alldifferent` constraint in order to limit the amount of nodes generated in the BDD. The way we have attained this is by limiting the combinations of values the variables involved in an `alldifferent` constraint can have. This is done by limiting the domain of each variable in such a way

Algorithm 3 Solver-based valid domain computations algorithm using an over- and under-approximation

```
1: procedure CVD-SP($C$)
2:     $VD' \leftarrow$ PROPAGATE($C$)
3:     $VD^o \leftarrow$ COMPUTEVALIDDOMAINS($BDD^o$)
4:     $VD^u \leftarrow$ COMPUTEVALIDDOMAINS($BDD^u$)
5:     $VD \leftarrow VD^u$
6:     for all $x_i \in X$ do
7:         if $|VD'_i| = 1$ then
8:             $VD_i = VD'_i$
9:             continue
10:        for all $v_{ij} \in VD'_i$ do
11:            if $v_{ij} \in VD_i$ then
12:                continue
13:            else if $VD_i = \emptyset \wedge v_{ij} = v_i^{\text{last}}$ then
14:                $VD_i \leftarrow \{v_{ij}\}$
15:            else if $v_{ij} \notin VD^o_i$ then
16:                continue
17:            else if $C_{|x_i=v_{ij}}$ is satisfiable then
18:                $S \leftarrow$ solution to search
19:                for all $(x_k, v_k) \in S$ do
20:                    $VD_k = VD_k \cup v_k$
21:            else
22:                $C \leftarrow C_{|x_i \neq v_{ij}}$
23:        $VD' \leftarrow$ PROPAGATE($C$)
```

that the union of the limited domain of all the variables is still the complete domain.

Example: If for example we have 10 variables with $D_i = \{1, 2, \ldots, 10\}$, we can *slice off* one value from each variable so the domains become $D_1 = \{1, 2, \ldots, 9\}$, $D_2 = \{1, 2, \ldots, 8, 10\}$, $D_3 = \{1, 2, \ldots, 7, 9, 10\}$, etc. If we continue with the example and we wanted to do a *domain slice* of half the values, the domains would become $D_1 = \{1, 2, \ldots, 5\}$, $D_2 = \{2, 3, \ldots, 6\}$, $D_3 = \{3, 4, \ldots, 7\}$, etc. To avoid making the under-approximation too narrow, we always construct the complement set of values when we slice off values of the domains. In the last example given, the complement domain values would be $D'_1 = \{6, 7, \ldots, 10\}$, $D'_2 = \{1, 7, \ldots, 10\}$,
$D'_3 = \{1, 2, 8, 9, 10\}$, etc. After slicing the domains, the `alldifferent` constraint becomes

$$\begin{aligned}&Alldiff(x_1, x_2, \ldots, x_n) \wedge \\ &(x_1 \in D_1 \wedge x_2 \in D_2 \wedge \ldots \wedge x_n \in D_n \vee \\ &x_1 \in D'_1 \wedge x_2 \in D'_2 \wedge \ldots \wedge x_n \in D'_n)\end{aligned}$$

5 Approximations Build Over Time

An alternative way of constructing the approximations is by building it over time. We can achieve this by noting each time we perform a search to find a solution in the CP given a partial assignment PA that takes an excessive amount of time and does not find a solution. Each time this happens we can conjoin an additional constraint on to the over-approximation of the form $\neg$PA. By doing this we are using the over-approximation as a way of performing no-good recording [Hawkins and Stuckey, 200].

In the case of the under-approximations, we are interested in the case where we perform a search that takes an excessive amount of time and actually finds a solution. In this case we can extend the under-approximation by setting it equal to the disjunction of the solution found and the existing under-approximation.

6 Results

In this section we compare the search-preventing configurator(s) with the pure search configurator. When building the over- and under-approximations for the various problems using the search results as described above, we added all results that took more than $10\,\mathrm{ms}$.

The different configurators use these abbreviations:

CVD-S The pure search-based configurator.

CVD-R The search-preventing hybrid configurator described using the statically build over- and under approximations.

CVD-CB The search-preventing hybrid configurator configurator using over- and under-approximations build purely from search results.

CVD-WB The search-preventing hybrid configurator using the statically build over- and under approximations augmented with results gathered while performing search. This configurator is thus a combination of the two described above.

One of the problems we have tested the hybrid configurator on is the data center configuration problem.

We have performed tests with 5 different sized data center configuration problems. The sizes are 4, 6, 8, 10 and 11 servers. For each of the problem instances we have created the under-approximation by slicing half of the domains of the variables in the `alldifferent` constraint representing the constraint that each server can only be used once. It is worth noting that the maximum number of servers we can have in a monolithic BDD representing the data center configuration problem is 10. For this reason we have tried to see how little we could slice of the domains in the under-approximation representing the data center configuration problem with 11 servers, and still be able to contain it in the under-approximation BDD. The limit we found is 4 values sliced of each domain of 11 values. The problem instances in the experimental results are listed as `dcNN-SS` where `NN` denotes the number of servers in the problem instance, and `SS` denotes the number of values sliced from the domains. All tests were performed on an Intel Core 2 Duo 6600 2.4 GHz Dual Core Processor workstation with 2 GB RAM running Windows XP Professional SP3.

The result of these tests are shown in 3 tables, where Table 1 shows the maximum valid domain computation times. As can be seen, CVD-CB performs the best overall. We attribute this to the fact that CVD-CB cuts of all those searches that takes too long and has a relatively small size BDDs compared to the BDDs used by CVD-WB and CVD-R.

The average valid domain computation times are shown in Table 2. It is apparent that for the smallest problems the overhead of using BDDs is not made up by the searches skipped. We see however that when the problem size grows it more than makes up for it. CVD-CB and CVD-WB performs best. Furthermore we see that pure search (CVD-S) and CVD-Reg are about the same.

In Table 3 we see as expected that the maximum number of searches is performed in CVD-S and the least is performed in CVD-WB. If we assume that the searches skipped are evenly distributed among those that are fast and those that slow then this is an important fact since it decreases the likelihood that CVD-WB run into a search that takes an extremely long time.

	Max CVD time [ms]			
Problem	CVD-S	CVD-R	CVD-CB	CVD-WB
dc4-2	**16**	**16**	**16**	**16**
dc6-3	**31**	32	32	32
dc8-4	47	63	47	**47**
dc10-2	94	63	**47**	157
dc10-5	79	79	78	**63**
dc11-4	110	172	**78**	329
dc11-5	125	157	**79**	188

Table 1: Max time of the valid domain computations of the search preventing CVD algorithms for the Data center configuration problem.

	Average CVD time [ms]			
Problem	CVD-S	CVD-R	CVD-CB	CVD-WB
dc4-2	**2**	4	6	5
dc6-3	**7**	11	10	11
dc8-4	17	22	16	**15**
dc10-2	28	25	**21**	24
dc10-5	28	33	24	**22**
dc11-4	44	55	**31**	42
dc11-5	44	54	**30**	35

Table 2: Average time of the valid domain computations of the search preventing CVD algorithms for the Data center configuration problem.

	Searches performed			
Problem	CVD-S	CVD-R	CVD-CB	CVD-WB
dc4-2	1082	785	927	**578**
dc6-3	2116	2110	**1138**	1183
dc8-4	3796	3800	1406	**1281**
dc10-2	5222	3016	1601	**1084**
dc10-5	5358	5268	1656	**1477**
dc11-4	7560	7362	1690	**1622**
dc11-5	7560	7384	1543	**1444**

Table 3: Searches performed of each of the search preventing CVD algorithms for the Data center configuration problem.

7 Conclusion

This paper has introduced three new algorithms that combines BDDs and backtracking search for backtrack-free interactive configuration. Our results show that the performance of these algorithms dominate purely search- or BDD-based approaches.

7.1 Directions and Future Work

Another approach to constructing hybrid configurators is to augment a propagator-centric solver by BDD-based propagators. The idea is that several constraint can be represented by a single BDD, and thus improve propagation strength, since there is strong n-consistency between the constraints in the BDD.

We tested this idea be implementing a BDD-propagator in Gecode, but no improvement of runtime was achieved even though we did get stronger propagation.

Future work could go into exploring new ways of constructing the approximations, that enables them to be as close to the original problem as possible and at the same time limits the amount of space needed to represent them. To achieve this, new data structures could be tested, for representing the approximations. Interesting data structures could be MDDs [Kam *et al.*, 1998], Tree-of-BDDs [Subbarayan, 2005] and cartesian product tables [Møller, 1995]. It could also be investigated whether it would be beneficial to use different data structures for two approximations.

References

[Andersen and Hulgaard, 2007] Henrik Reif Andersen and Henrik Hulgaard. Configit software, 2007.

[Bryant, 1986] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, aug 1986.

[Cheng and Yap, 2006] Kenil C. K. Cheng and Roland H. C. Yap. Maintaining generalized arc consistency on ad-hoc n-ary boolean constraints. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *ECAI*, pages 78–82. IOS Press, 2006.

[Hadzic *et al.*, 2004] Tarik Hadzic, Sathiamoorthy Subbarayan, Rune Møller Jensen, Henrik Reif Andersen, Henrik Hulgaard, and Jesper Møller. Fast backtrack-free product configuration using a precompiled solution space representation. In *Proceedings of the International Conference on Economic, Technical and Organizational aspects of Product Configuration Systems*, pages 131–138. DTU-tryk, 2004.

[Hawkins and Stuckey, 200] Peter Hawkins and Peter J. Stuckey. A hybrid BDD and SAT finite domain constraint solver. In P. Van Hentenryck, editor, *Proceedings of the Practical Applications of Declarative Programming, 8th International Symposium*, volume 3819 of *LNCS*, pages 103–117. Springer, 200.

[Jensen, 2004] Rune M. Jensen. CLab: A C++ library for fast backtrack-free interactive product configuration. In Mark Wallace, editor, *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 of *Lecture Notes in Computer Science*, page 816. Springer, 2004.

[Kam *et al.*, 1998] T. Kam, T. Villa, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Multi-valued decision diagrams: Theory and applications. *International Journal on Multiple-Valued Logic*, 4:9–62, 1998.

[Møller, 1995] Gert Møller. *On the Technology of Array Based Logic*. PhD thesis, Technical University of Denmark, Lyngby, Denmark, 1995.

[Schulte *et al.*, 2009] Christian Schulte, Mikael Lagerkvist, and Guido Tack. Gecode. Software download and online material, 2009. `http://www.gecode.org`.

[Subbarayan *et al.*, 2004] Sathiamoorthy Subbarayan, Rune M. Jensen, Tarik Hadzic, Henrik R. Andersen, and Henrik Hulgaard. Comparing two implementations of a complete and backtrack-free interactive configurator. In *Proceedings of the CP-04 Workshop on CSP Techniques with Immediate Application*, pages 97 – 111, aug 2004.

[Subbarayan *et al.*, 2006] Sathiamoorthy Subbarayan, Lucas Bordeaux, and Youssef Hamadi. On hybrid SAT solving using tree decompositions and BDDs. Technical Report MSR-TR-2006-28, Microsoft Research (MSR), March 2006.

[Subbarayan, 2005] Sathiamoorthy Subbarayan. Integrating csp decomposition techniques and bdds for compiling configuration problems. In Roman Barták and Michela Milano, editors, *CPAIOR*, volume 3524 of *Lecture Notes in Computer Science*, pages 351–365. Springer, 2005.

[Subbarayan, 2008] Sathiamoorthy Subbarayan. Efficient reasoning for nogoods in constraint solvers with BDDs. In Paul Hudak and David Scott Warren, editors, *Practical Aspects of Declarative Languages, 10th International Symposium, PADL 2008, San Francisco, CA, USA, January 7-8, 2008*, volume 4902 of *Lecture Notes in Computer Science*, pages 53–67. Springer, 2008.

[Tiedemann, 2008] Peter Tiedemann. *Compiled Data Structures and Global Constraints in Constraint Processing*. PhD thesis, ITU, 2008.

[van Hoeve, 2001] Willem Jan van Hoeve. The alldifferent constraint: A survey. *CoRR*, cs.PL/0105015, 2001. informal publication.