Constrained Decision Diagrams

Kenil C.K. Cheng and **Roland H.C. Yap**
National University of Singapore
3 Science Drive 2, Singapore
{chengchi,ryap}@comp.nus.edu.sg

Abstract

A general n-ary constraint is usually represented explicitly as a set of its solution tuples, which may need exponential space. In this paper, we introduce a new representation for general n-ary constraints called *Constrained Decision Diagram* (CDD). CDD generalizes BDD-style representations and the main feature is that it combines constraint reasoning/consistency techniques with a compact data structure. We present an application of CDD for recording all solutions of a conjunction of constraints. Instead of an explicit representation, we can implicitly encode the solutions by means of constraint propagation. Our experiments confirm the scalability and demonstrate that CDDs can drastically reduce the space needed over explicit and ZBDD representations.

Introduction

Many real-life combinatorial problems such as scheduling can be modeled as a constraint satisfaction problem (CSP). Despite the vast improvements on search strategies and consistency algorithms for binary and specific constraints, relatively less efforts have been put into issues with general n-ary constraints. In particular, this paper looks at the issue of efficient representations of the solution space. This is important when we want to record all solutions to a CSP or as efficient representations of n-ary constraints which can be used with consistency algorithms.

An efficient representation of all solutions is useful for some applications. For example, a configuration system may be interactive where real-time response is necessary. So finding the solutions of the CSP in advance and filtering that against the user's requirements is more efficient than solving from scratch. In this paper, we are interested in efficient and compact representations which can be used to represent solution spaces during search. Moreover, we show how various local consistencies and the constraint solver can be exploited in the representation to get more compactness.

To represent the solutions of a CSP or a general n-ary constraint compactly, one approach is an implicit representation using a symbolic relation, e.g. a logical formula involving arithmetic. While the implicit form can be very compact, they can be difficult expensive to use and discover, especially when constraints are dynamically created. The opposite approach is to use an explicit representation where data structures are used to compress the solution set directly, i.e. suffix or prefix sharing. These might not be practical when the solution set is huge (since its explicit) or difficult to compress.

In this paper, we introduce a new representation for solution sets and general n-ary constraints, called Constrained Decision Diagram (CDD). CDD is a hybrid which combines the strengths of implicit and explicit approaches. On the one hand, it can be explicit and generalizes BDD-style representations. It is also implicit, combining constraint reasoning/consistency techniques with a compact data structure.

Binary decision diagram (BDD) (Bryant 1986) is the state of the art representation for propositional logic, and CAD. Unlike BDD which only allows propositional variables (i.e. the only available constraints are $x = 1$ and $x = 0$), a CDD can use arbitrary constraints. We show how CDDs can be used to represent all solutions of a CSP where space savings come because of the implicit representation and use of constraint propagation. Our experiments show that CDDs can give 1-2 orders of magnitude space savings over explicit and ZBDDs (Okuno, Minato, & Isozaki 1998) representations. We remark that for large solution spaces, memory usage can be the primary consideration.

Preliminaries

A *constraint satisfaction problem* (CSP) is a triple $\mathcal{P} = \langle X, \mathcal{D}, \mathcal{C} \rangle$, where $X = \{x_1, \ldots, x_n\}$ is a set of *variables*, $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a set of *domains*, and $\mathcal{C}$ is a set of *constraints*[1] Each variable x_i can only take values from its domain D_i, which is a set of integers. A *valuation* θ is a mapping of variables to integer values, written as $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$. Let $vars$ be the function that returns the set of (free) variables appearing in a constraint or valuation. A k-ary constraint $c \in \mathcal{C}$ on an ordered set of k distinct variables, sometimes written as $c(x_1, \ldots, x_k)$, is a subset of the Cartesian product $D_1 \times \cdots \times D_k$ that restricts the values the variables in c can take simultaneously. A valuation θ *satisfies* c, a.k.a. a *solution* of c, if and only if $\theta \in c$. Solving a CSP requires finding a value for each variable from its domain so that all constraints are satisfied.

[1] In this paper we use the terms "a set of constraints" and "a conjunction of constraints" interchangeably.

Let Γ be a set of constraints. Two constraints c_1 and c_2 are *equivalent w.r.t.* Γ, denoted by $c_1 \equiv_\Gamma c_2$, if and only if for any valuation θ, we have $\theta \in (\Gamma \wedge c_1) \iff \theta \in (\Gamma \wedge c_2)$. In particular, when Γ is $true$, c_1 and c_2 are *equivalent*, written as $c_1 \equiv c_2$, if and only if they define the same relation.

Constrained Decision Diagrams

A *constrained decision diagram* (CDD) $\mathcal{G} = \langle \Gamma, G \rangle$ consists of a set of constraints Γ (named *CDD constraints*) and a rooted, directed acyclic graph (DAG) $G = (V \cup T, E)$ (named *CDD graph*). We call a node $v \in V \cup T$ a *CDD node* (or simply *node*). The *0-terminal* ($\mathbf{0} \in T$) represents $false$ and the *1-terminal* ($\mathbf{1} \in T$) represents $true$. G has at least one terminal. Every non-terminal node $v \in V$ connects to a subset of nodes $U \subseteq V \cup T - \{v\}$. Each $u \in U$ is a *successor* of v, i.e., there exists a directed edge $vu \in E$ from v to u. A non-terminal node v denotes a non-empty set $\{(c_1, u_1), \ldots, (c_m, u_m)\}$. Each *branch* (c_j, u_j) consists of a constraint $c_j(x_1, \ldots, x_k)$ and a successor u_j of v. Figure 1 gives the graphical representation of v. Each outgoing

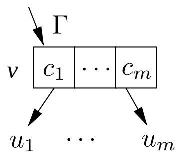

Figure 1: Graphical representation of a CDD node v

arrow from c_j points to the corresponding successor u_j of v. If $\mathcal{G} = \langle \Gamma, G \rangle$ is a CDD and v is the root node of G we will explicitly draw an incoming arrow to v and label it with Γ. Hereafter, when we say "a CDD rooted at a node v", we actually mean its CDD graph is rooted at v.

A CDD $\mathcal{G} = \langle \Gamma, G \rangle$ rooted at a CDD node v represents the constraint $\Gamma \wedge [\![v]\!]$ where

$$[\![v]\!] \equiv \begin{cases} true & : \quad v = \mathbf{1} \\ false & : \quad v = \mathbf{0} \\ \bigvee_{j=1}^{m} (c_j \wedge [\![u_j]\!]) & : \quad v = \bigcup_{j=1}^{m} \{(c_j, u_j)\} \end{cases}$$

Let Γ be a set of constraints. In an abuse of notation, we define two CDD nodes u and v are *equivalent w.r.t.* Γ, written as $u \equiv_\Gamma v$, if and only if $[\![u]\!] \equiv_\Gamma [\![v]\!]$.

Example 1 *Figure 2 shows a CDD $\mathcal{G} = \langle \Gamma, G \rangle$. Let $c_{11} \equiv$*

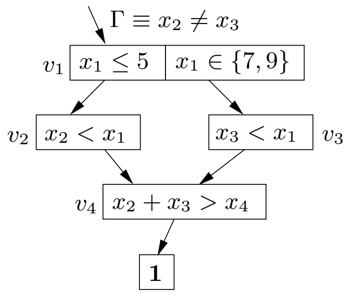

Figure 2: A CDD

$x_1 \le 5$, $c_{12} \equiv x_1 \in \{7, 9\}$, $c_2 \equiv x_2 < x_1$, $c_3 \equiv x_3 < x_1$ and $c_4 \equiv x_2 + x_3 > x_4$. The constraint represented by $\mathcal{G}$ is

$$\begin{aligned} & \Gamma \wedge [\![v_1]\!] \\ \equiv\ & \Gamma \wedge (c_{11} \wedge [\![v_2]\!]) \vee (c_{12} \wedge [\![v_3]\!]) \\ \equiv\ & \Gamma \wedge (c_{11} \wedge c_2 \wedge [\![v_4]\!]) \vee (c_{12} \wedge c_3 \wedge [\![v_4]\!]) \\ \equiv\ & \Gamma \wedge (c_{11} \wedge c_2 \wedge c_4 \wedge [\![\mathbf{1}]\!]) \vee (c_{12} \wedge c_3 \wedge c_4 \wedge [\![\mathbf{1}]\!]) \\ \equiv\ & \Gamma \wedge ((c_{11} \wedge c_2) \vee (c_{12} \wedge c_3)) \wedge c_4. \end{aligned}$$

A CDD $\mathcal{G} = \langle \Gamma, G \rangle$ rooted at the CDD node $v = \{(c_1, u_1), \ldots, (c_m, u_m)\}$ is *reduced* if and only if each CDD graph G'_j rooted at u_j is either a terminal or reduced, and

$$c_i \wedge c_j \quad \equiv_\Gamma \quad false \tag{1}$$

$$u_i \quad \not\equiv_\Gamma \quad u_j \tag{2}$$

for all $1 \le i < j \le m$. In this paper, we are only interested in reduced CDDs.

Example 2 *A box constraint collection (BCC) (Cheng, Lee, & Stuckey 2003)*

$$\bigvee_{i=1}^{n} (c_i \wedge \bigwedge_{j=1}^{m} a_{ij} \le x_i \le b_{ij})$$

where a_{ij} and b_{ij} are integers, is a (non-reduced) CDD $\mathcal{B} = \langle true, G \rangle$ whose root $v = \{(c_1, u_{11}), \ldots, (c_n, u_{n1})\}$ and all other nodes $u_{ij} = \{(a_{ij} \le x_i \le b_{ij}, u_{ij+1})\}$.

Example 3 *A multi-valued decision diagram (MDD) (Srinivasan* et al. *1990) over variables $x_1, \ldots, x_n$ is a CDD $\mathcal{M} = \langle true, G \rangle$ in which every CDD node v is of the form $\{(x_k = d, u) \ : \ d \in D_k\}$ where D_k is the domain of x_k and u is the corresponding successor. In particular, if all variables have a Boolean domain $\{0, 1\}$, the MDD reduces to a binary decision diagram (BDD) (Bryant 1986).*

Given a fixed variable ordering, a multi-valued function is uniquely represented by a MDD. This strong property makes equivalence checking of two MDDs efficient, which is critical for their traditional use. For CDD, however, as a representation for general n-ary constraints, we argue flexibility (e.g. being able to make use of global constraints at ease) is more important, while keeping a canonical form is optional.

Example 4 *Consider a CSP $\mathcal{P}$ with three variables x_1, x_2 and x_3. All variables have a domain $\{1, 2, 3\}$. The constraint is that the variables must take distinct values, i.e., $\mathcal{C} = \{x_1 \neq x_2, x_2 \neq x_3, x_1 \neq x_3\}$. Figure 3 depicts three possible CDDs for $\mathcal{C}$. The left one (a) is a MDD representation. The integer d in a node means the valuation $x_i \mapsto d$. We draw out-going edges to the 0-terminal as dashed arrows. The middle one (b) shows a CDD based on a set of disequality constraints $x_i \neq x_j$. The right one (c) gives a CDD with just one all-different constraint `all_distinct`(x_1, x_2, x_3). Notice the compactness a CDD could achieve by representing solutions implicitly.*

Representing Solutions with CDD

We can use a CDD to represent all solutions of a CSP. Instead of a direct compression on the solution set, a CDD is an implicit representation using constraint propagation. Adding the CDD as an implied constraint to the original CSP, a deep backtrack-free search for solutions is guaranteed. The next example gives the overall picture.

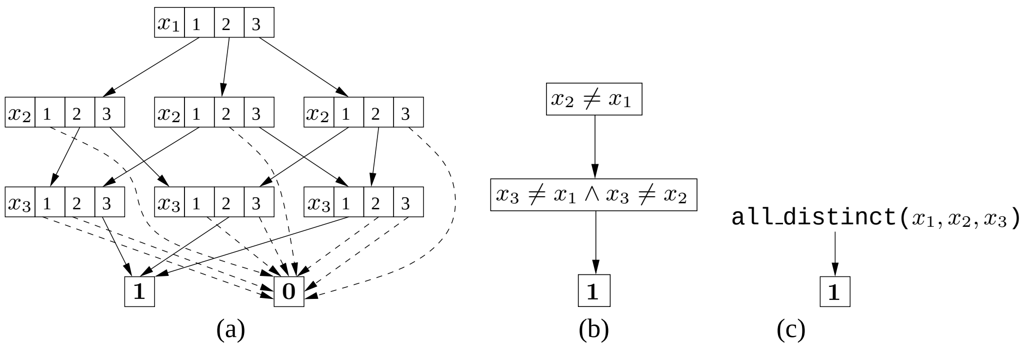
x_1 1 2 3
x_2 1 2 3
x_3 1 2 3
1
0
(a)
$x_2 \neq x_1$
$x_3 \neq x_1 \wedge x_3 \neq x_2$
1
(b)
all_distinct(x_1, x_2, x_3)
1
(c)

Figure 3: Three different CDDs for $x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_2 \neq x_3$. (a) A MDD-like representation, all solutions are maintained explicitly. (b) A CDD with disequality constraints inside nodes. (c) A CDD with a single `all_distinct(`x_1, x_2, x_3`)`.

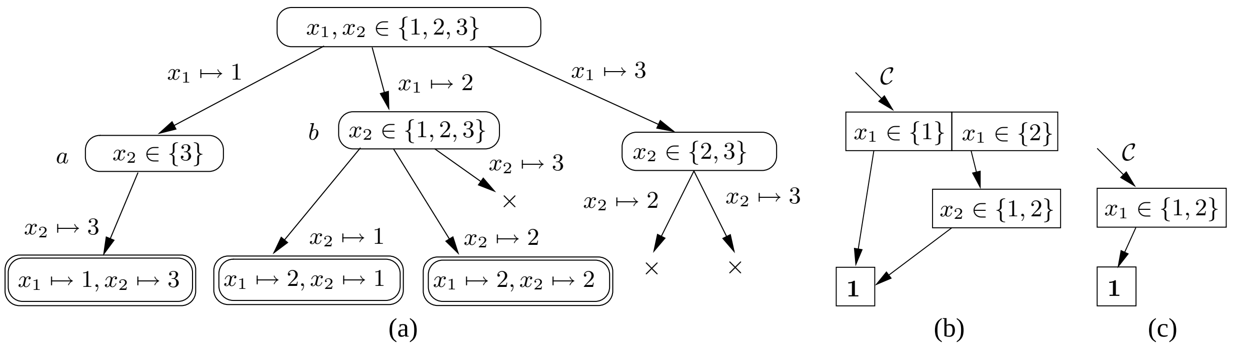

Figure 4: Let $\mathcal{P} = \langle\{x_1, x_2\}, \{\{1,2,3\},\{1,2,3\}\}, \mathcal{C}\rangle$ be a CSP with three solutions $\{x_1 \mapsto 1, x_2 \mapsto 3\}$, $\{x_1 \mapsto 2, x_2 \mapsto 1\}$ and $\{x_1 \mapsto 2, x_2 \mapsto 2\}$. (a) A depth-first backtracking search tree T for $\mathcal{P}$. A round rectangular node represents a search state. A double-edged round rectangular node represents a solution state (i.e. $true$). A cross ($\times$) indicates a dead-end (i.e. $false$). A transition from a search state to another is shown as a directed edge labeled with the corresponding valuation $x_i \mapsto d$. (b) A CDD $\mathcal{G}_1$ that represents T. (c) A CDD $\mathcal{G}_2$ that "approximates" T.

Example 5 *Let $\mathcal{P} = \langle X, \mathcal{D}, \mathcal{C}\rangle$ be a CSP where $X = \{x_1, x_2\}$ and $D_1 = D_2 = \{1,2,3\}$. We assume that there are some constraints $\mathcal{C}$ so that $\mathcal{P}$ has three solutions $\{x_1 \mapsto 1, x_2 \mapsto 3\}$, $\{x_1 \mapsto 2, x_2 \mapsto 1\}$ and $\{x_1 \mapsto 2, x_2 \mapsto 2\}$. Figure 4(a) depicts the depth-first backtracking search tree T for $\mathcal{P}$. We will explain how to "compress" T into a CDD. First, we look at node a. After we have assigned 1 to x_1, the domain of x_2 is reduced to the singleton set $\{3\}$. Thus, we can determine the first solution by posing the constraint $x_1 = 1$. Constraint propagation will do the rest. Next, consider the situation if we have assigned 2 to x_1. At node b, we can assign x_2 with either 1 or 2, each corresponds to a solution of $\mathcal{P}$. However, $x_2 \mapsto 3$ leads to a dead-end. We can represent the two solutions and exclude the dead-end with the constraint $x_1 = 2 \wedge x_2 \in \{1,2\}$ There is no solution for the branch $x_1 \mapsto 3$. To summarize, adding the implied constraint*

$$(x_1 = 1) \vee (x_1 = 2 \wedge x_2 \in \{1,2\})$$

to $\mathcal{C}$ we can find the solutions without backtracking. The corresponding CDD $\mathcal{G}_1$ is shown in Figure 4(b).

Very often, we can obtain a smaller CDD if we "assume" the constraints achieve a higher level of consistency than they actually do. For instance, consider node b again, if we enforce singleton consistency (Debruyne & Bessière 1997) on D_2, the domain of x_2, we can immediately remove 3 from D_2. Since then every value in D_2 contributes to a solution of $\mathcal{P}$, a simpler constraint $x_1 = 2$ is enough to represent these two solutions. Figure 4(c) gives the corresponding CDD $\mathcal{G}_2$. Note that it only has one node, while $\mathcal{G}_1$ has two. Of course, as the original constraints do not achieve singleton consistency, $\mathcal{G}_2$ contains some shallow[2] dead-ends, hence we say it only "approximates" T.

This example shows the use of constraint propagation and approximating the search tree could reduce the amount of explicit information needed for representing solutions. We are ready present our `cddFindall` algorithm. Taking a CSP $\mathcal{P} = \langle X, \mathcal{D}, \mathcal{C}\rangle$ as input, it compiles the solutions of $\mathcal{P}$ into a CDD $\mathcal{G} = \langle \mathcal{C}, G\rangle$. Each CDD node v is of the form $\{(x_k \in r_1, u_1), \ldots, (x_k \in r_m, u_m)\}$ where $r_1, \ldots, r_m \subset D_k$ are pairwise disjoint (so as to satisfy Property (1) of a reduced CDD). Consequently, $\{r_1, \ldots, r_m\}$ is a partition of

[2]We say a valuation $x_i \mapsto d$ leads to a shallow (deep) dead-end if this dead-end can(not) be detected by enforcing singleton consistency on the domain of x_i.

some non-empty subset of D_k.

```
cddFindall ($X, \mathcal{D}, \mathcal{C}$)
begin
1   if $vars(\mathcal{C}) = \emptyset$ then
        return 1 // 1-terminal
    else
2       choose $x_k \in vars(C)$
        $S$ := $\emptyset$
        $D'_k$ := $\emptyset$
3       foreach $d \in D_k$ do
            $\mathcal{C}'$ := $\mathcal{C}$ with all $x_k$ replaced by $d$
4           if $\mathcal{C}'$ is not locally inconsistent then
                $u$ := cddFindall($X, \mathcal{D}, \mathcal{C}'$)
                $S$ := $S \cup \{\langle d, u\rangle\}$
                $D'_k$ := $D'_k \cup \{d\}$
5       if $D'_k = \emptyset$ then
            return 0 // 0-terminal
        else
6           $v$ := mkNode($x_k, S, \mathcal{C} \cup \{x_k \in D'_k\}$)
            return $v$
end
```

Figure 5: `cddFindall`

Figure 5 gives the pseudo-code of `cddFindall` which uses singleton consistency (can be extended to other consistencies). Its skeleton is a depth-first backtracking search algorithm. The control flow is as follows. If all variables have been assigned, i.e., a solution has been found, the 1-terminal will be returned (line 1). Otherwise, we select an uninstantiated variable x_k (line 2). We also initialize two empty sets: S will collect branches of a CDD node, and D'_k will be a singleton consistent domain of x_k. Next, for each value d in the domain D_k (line 3) we make new constraints $\mathcal{C}'$ that are equivalent to $\mathcal{C} \wedge (x_k = d)$. If $\mathcal{C}'$ is not inconsistent, as determined by the constraint solver, (*this makes use of constraint propagation in the existing solver*) (line 4) we invoke `cddFindall` (recursively) to find a CDD node u for the sub-problem $\langle X, \mathcal{D}, \mathcal{C}'\rangle$, add the pair (d, u) to S[3], and insert d to D'_k. If $D'_k = \emptyset$ (line 5) when the iteration is over, the 0-terminal will be returned as the constraints $\mathcal{C}$ are unsatisfiable. Otherwise, `mkNode` is called (line 6) to find a CDD node v for the current (sub)-problem. (The reasons for posing $x_k \in D'_k$, namely to make the domain of x_k smaller or singleton consistent, will become clear when we discuss the reduction rules for CDD.) Finally we return v. The execution of `cddFindall` stops after traversing the entire search tree.

Figure 6 shows the pseudo-code of `mkNode`. The global variable V maintains a set of all existing CDD nodes. At line 7, an intermediate CDD node v is created in such a way that for every $d \in r$, $x_k \mapsto d$ leads to the same successor u. Next, in order to satisfy Property (2) of a reduced CDD, at line 8, we check if there already exists a node $v' \in V$ such that v and v' are equivalent w.r.t. $\mathcal{C}$. If so, we will reuse v'. Otherwise, we insert v to V (line 9) and return v.

[3]Theoretically it is unnecessary to keep (d, u) when u is the 0-terminal, i.e. a CDD needs not explicitly state $x_k \neq d$. This, however, simplifies the implementation and our presentation.

```
mkNode ($x_k, S, \mathcal{C}$)
// Global variable $V$ contains all CDD nodes
begin
7   $v$ := $\{(x_k \in r, u) : d, d' \in r \iff \langle d, u\rangle, \langle d', u\rangle \in S\}$
    // i.e. $x_k \mapsto d$ and $x_k \mapsto d'$ point to the same node $u$
8   if $\exists v' \in V$ s.t. $v' \equiv_{\mathcal{C}} v$ then
        return $v'$
    else
9       $V$ := $V \cup \{v\}$
        return $v$
end
```

Figure 6: `mkNode`

We are going to present two lemmas which allow efficient equivalence checking and minimization for CDD. They are generalizations of the reduction rules for MDDs. Because of the space constraint, proofs are omitted.

Lemma 1 *Suppose x_k is variable with domain D_k, and Γ a set of constraints. Let u and v be two CDD nodes. If $v = \{(x_k \in D_k, u)\}$ then $v \equiv_\Gamma u$.*

Lemma 1 says v can be replaced by u because, by the definition of a CSP, a variable can only take values from its domain. One reason to skip any singleton inconsistent values SI (line 4 of `cddFindall`) is that, the chance to apply this lemma will be higher if these easy-to-detect dead-ends are not explicitly expressed as an extra branch $(x_k \in SI, \mathbf{0})$.

We will formalize the example below into Lemma 2.

Example 6 *The left of Figure 7 depicts the scenario when a CDD node u can be "embedded" into another CDD node v. It is not difficult to verify that*

$$x \in \{2, 5\} \wedge [\![u]\!] \equiv x \in \{2, 5\} \wedge [\![v]\!].$$

Intuitively speaking we use the domain of x to mask out unwanted branches.

Merging can occur more often if the domain of a variable is tight, say, singleton consistent. In fact, when there is no CDD constraint, such as in BDD or MDD, the domains of (uninstantiated) variables are invariant along any different paths, and thus only identical sub-graphs can merge.

Lemma 2 *Suppose $\mathcal{G} = \langle\Gamma, G\rangle$ is a CDD rooted at v. Let v' be a CDD node and x_k a variable with domain D_k. If for every $(x_k \in r, u) \in v$ there exists $(x_k \in r', u') \in v'$ such that $u = u'$ and $r = D_k \cap r'$, we have $v \equiv_\Gamma v'$.*

Let G be the CDD graph returned by `cddFindall` for the CSP $\mathcal{P} = \langle X, \mathcal{D}, \mathcal{C}\rangle$. Clearly, the CDD $\mathcal{G} = \langle\mathcal{C}, G\rangle$ satisfies Property (1). It also satisfies Property (2) of a reduced CDD if `mkNode` finds equivalent CDD nodes in a static order, say, a new node is always appended to the end of a list V, and `mkNode` always returns from V the first equivalent node. Finally, we show the correctness of our `cddFindall` algorithm:

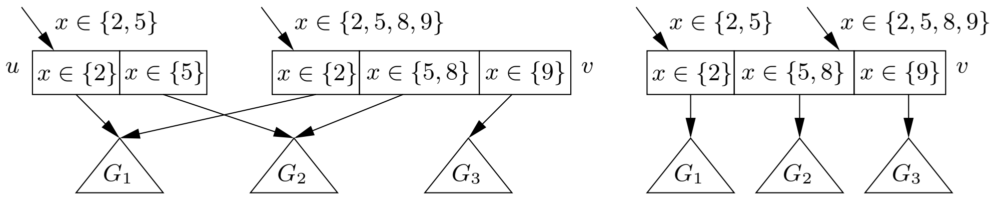

Figure 7: Merging two CDD nodes (left) into a single one (right)

Theorem 1 *Given a CSP $\mathcal{P} = \langle X, \mathcal{D}, \mathcal{C} \rangle$, the* `cddFindall` *algorithm returns a CDD graph G such that the CDD $\mathcal{G} = \langle \mathcal{C}, G \rangle$ is reduced and preserves all solutions of $\mathcal{P}$, i.e. $\mathcal{C} \wedge [\![v]\!] \equiv \mathcal{C}$, where v is the root node of G.*

Corollary 1 *Given a CSP $\mathcal{P} = \langle X, \mathcal{D}, \mathcal{C} \rangle$ as input,* `cddFindall` *produces a CDD graph G such that, a solution of $\mathcal{P}$ can be obtained from the reduced CDD $\mathcal{G} = \langle \mathcal{C}, G \rangle$ by doing a deep backtrack-free traversal from the root of G to the 1-terminal, in the presence of the constraints $\mathcal{C} \wedge [\![v]\!]$, where v is the root node of G.*

Experimental Results

We have implemented a prototype of `cddFindall` in SICStus Prolog 3.12.0. Our prototype represents CDD nodes as dynamic clauses for ease of implementation. This effects the efficiency as the number of CDD nodes increases. This is not important since we focus on the scalability of CDD in terms of its size and memory used. Experiments were run on a PC running Windows XP, with a P4 2.6GHz CPU and 1 GB physical memory. Due to the lack of space, we only present the results on the N-queens problem. Finding a compact representation for all the solutions of the N-queens problem is challenging for two reasons. First, its permutation nature implies compression methods by just suffix or prefix sharing on the solution set, such as to use a trie or a MDD, are doomed to failure.[4] Second, its number of solutions grows exponentially with N, which makes scalability critical.

The N-queens problem is to place N queens on a $N \times N$ chess board such that no two queens attack each other. Our model involves N variables $x_1, \dots, x_N$. Each variable has a domain $D_i = \{1, \dots, N\}$. A valuation $x_i \mapsto j$ means the queen is placed in the i-th row and the j-th column. We use `all_distinct(`$x_1, \dots, x_N$`)` that enforces generalized arc consistency, and a no-attack constraint

$$|x_i - x_j| \neq |i - j|$$

for each pair of variables x_i and x_j. There are two versions of the no-attack constraints, one achieves bounds consistency (BC) and the other achieves arc consistency (AC).

Table 1 summarizes the results. N is the number of queens. S is the number of solutions. The columns under "findall+BC/AC" show the execution time (t seconds) and the total memory (m MB) used by the built-in predicates `labeling/2` and `findall/3` to generate and store all solutions. Note that `findall` stores the solutions in a list. BC (AC) means the no-attack constraints achieve bounds (arc) consistency. The columns under "CDD+BC/AC" show the time (t seconds) and the total memory (m MB) used by `cddFindall` to find all solutions and save them into a CDD with nn nodes. To measure the compactness of a CDD independent of the implementation details, which removes the overhead of dynamic clauses, we use the compression ratio

$$cr = \frac{N \cdot S}{nn}$$

Namely, the ratio of the number of cells in a (virtual) table that stores all the solutions to the number nodes of the CDD. Time, memory use and cr are reported up to 1 decimal place. Both `labeling` and `cddFindall` use static lexicographical variable and value ordering.

Firstly, regardless the consistency the no-attack constraints achieve, `cddFindall` uses significantly less memory than `findall`, especially when N is large. For instance, when $N = 14$, 119 MB was used by `findall`, compared to less than 10 MB by `cddFindall`. CDD is also significantly superior to the combinatorial sets representation using ZBDD[5] (Okuno, Minato, & Isozaki 1998) where we found that ZBDDs cannot compress the solutions well. Also, the compression ratio increases with N, which indicates our CDD representation becomes more efficient (in terms of size) when the solution set grows.

Secondly, let us compare the memory consumed by `cddFindall` and `findall` when we strengthen the consistency (from BC to AC) that the no-attack constraints achieve. For `cddFindall`, its memory use decreases with the increase of the level of consistency. In fact, when AC instead of BC is achieved, the size (number of nodes) of the CDD (for the same N) is reduced by about 33% or more. This is because the search tree shrinks, so does the CDD, when the consistency becomes stronger. However, the memory used by `findall` is insensitive to the level of consistency. The reason is obvious as the number of solutions is independent to the consistency the constraints achieve.

Our results justify the scalability of CDD for solutions recording, and the usefulness of combining constraints and decision diagrams. Currently, `cddFindall` runs slower than `findall` for larger instances. However, we claim this is because of the excessive use of dynamic clauses. A better implementation should improve the runtime.

[4]Compressing the 14200 solutions of the 12-queens problem using WinZip can only reduce its size from 403KB to 81KB.

[5]We implemented their proposed operations using the (Z)BDD package CUDD 2.4.0 (http://vlsi.colorado.edu/ fabio/CUDD/). For $N = 12$, the ZBDD has 45833 nodes, where each node encodes a valuation $x_i \mapsto j$. Our PC ran out of memory for $N > 12$.

N	S	findall+BC		findall+AC		CDD+BC				CDD+AC			
		t	m	t	m	t	m	nn	cr	t	m	nn	cr
10	724	0.3	2.3	0.3	2.3	0.4	2.1	173	41.8	0.3	2.1	116	62.4
11	2680	1.7	3.6	1.4	3.5	1.8	2.3	564	52.3	1.4	2.1	389	75.8
12	14200	9.1	8.1	7.4	8.1	9.8	2.5	2187	77.9	7.1	2.3	1445	117.9
13	73712	44.6	21.7	37.5	21.7	59.9	3.6	9462	101.3	41.5	3.1	6291	152.3
14	365596	339.1	119.1	244.6	119.1	496.6	9.0	42877	119.4	307.7	6.7	28450	179.9

Table 1: Experimental results on the N-queens problem

Discussion and Related Work

In this paper we have proposed a novel representation for general n-ary constraints called constrained decision diagrams. We have demonstrated its use in maintaining all solutions of a CSP. Our experimental results confirm the scalability and compactness of our new representation.

There are two streams of related work: either to find a high-level description (say, an arithmetic relation) from a solution set, e.g. (Apt & Monfroy 1999; Cheng, Lee, & Stuckey 2003; Dao *et al.* 2002), or to directly compress the solution set, e.g. (Barták 2001; Hubbe & Freuder 1992; van der Meer & Andersen 2004; Okuno, Minato, & Isozaki 1998; Weigel & Faltings 1999). We call the former the inductive approach and the latter the compression approach.

The inductive approach usually outputs a (symbolic) representation which is compact and efficient to process (by the constraint solver). However, it requires fairly expensive preprocessing on the solution set and compilation of the representation into some program codes. Hence, scalability is a problem and this approach is not suitable to process constraints that are generated during CSP solving.

The compression approach, in contrast, retains the set of solutions in its explicit form, but reduces its size via compression techniques such as suffix or prefix sharing. The compression process is incremental and needs no preprocessing. However, since the number of solutions could be exponential and sharing might not be very common among solutions (e.g. for permutation problems), dealing with solutions explicitly is sometimes impractical.

One pitfall to both approaches is they treat the solutions of a CSP or a constraint solely as a set of data, just like there is no other constraint. In practice, however, a constraint rarely comes alone. As we have already presented, taking existing constraints into account is indeed the key to a compact representation for solution set.

We consider CDD a bridge between the inductive and the compression approaches. On the one hand, arbitrary constraints can be used inside a CDD (node). In fact, by definition, a CDD interacts with other constraints via constraint propagation. On the other hand, a CDD, just like MDD and other BDD variants, can represent individual solution and be constructed incrementally. Its dual nature makes CDD a very flexible and powerful representation for constraints.

We would like to point out that, there exists variants of BDD which use some restricted form constraints. For instance, in difference decision diagram (Møller *et al.* 1999) each node represents a constraint $x < y + c$ where c is some constant. However, they are tailor-made for specific applications and do not apply (general) constraint solving.

We plan to implement a more efficient consistency algorithm for general n-ary constraints (say, GAC) on top of a CDD. As a compact and efficient representation for a huge solution set, CDD might also benefit applications such as (no)-goods learning during search and CSP decomposition.

References

Apt, K., and Monfroy, E. 1999. Automatic generation of constraint propagation algorithms for small finite domains. In *Principles and Practice of Constraint Programming*, 58–72.

Barták, R. 2001. Filtering algorithms for tabular constraints. In *Colloqium on Implementation of Constraint and Logic Programming Systems (CICLOPS)*, 168–182.

Bryant, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.* 35(8):667–691.

Cheng, C. K.; Lee, J. H. M.; and Stuckey, P. J. 2003. Box constraint collections for adhoc constraints. In *Principles and Practice of Constraint Programming*, 214–228.

Dao, T.; Lallouet, A.; Legtchenko, A.; and Martin, L. 2002. Indexical-based solver learning. In *Principles and Practice of Constraint Programming*, 541–555.

Debruyne, R., and Bessière, C. 1997. Some practicable filtering techniques for the constraint satisfaction problem. In *IJCAI*, 412–417.

Hubbe, P. D., and Freuder, E. C. 1992. An efficient cross product representation of the constraint satisfaction problem search space. In *AAAI*, 421–427.

Møller, J.; Lichtenberg, J.; Andersen, H. R.; and Hulgaard, H. 1999. Difference decision diagrams. In *Comp. Sci. Logic*.

Okuno, H.; Minato, S.; and Isozaki, H. 1998. On the properties of combination set operations. *Information Processing Letters* 66:195–199.

Srinivasan, A.; Kam, T.; Malik, S.; and Brayton, R. 1990. Algorithms for discrete function manipulation. In *Intl. Conf. on CAD*, 92–95.

van der Meer, E., and Andersen, H. R. 2004. Bdd-based recursive and conditional modular interactive product configuration. In *CSPIA Workshop*, 112–126.

Weigel, R., and Faltings, B. 1999. Compiling constraint satisfaction problems. *Art. Intell.* 115(2):257–287.