Approximate Compilation of Constraints into Multivalued Decision Diagrams

Tarik Hadzic[1], John N. Hooker[2], Barry O'Sullivan[1], and Peter Tiedemann[3]

[1] Cork Constraint Computation Centre
{thadzic,b.osullivan}@4c.ucc.ie
[2] Carnegie Mellon University
john@hooker.tepper.cmu.edu
[3] IT University of Copenhagen
petert@itu.dk

Abstract. We present an incremental refinement algorithm for approximate compilation of constraint satisfaction models into multivalued decision diagrams (MDDs). The algorithm uses a vertex splitting operation that relies on the detection of equivalent paths in the MDD. Although the algorithm is quite general, it can be adapted to exploit constraint structure by specializing the equivalence tests for partial assignments to particular constraints. We show how to modify the algorithm in a principled way to obtain an approximate MDD when the exact MDD is too large for practical purposes. This is done by replacing the equivalence test with a constraint-specific measure of distance. We demonstrate the value of the approach for approximate and exact MDD compilation and evaluate its benefits in one of the main MDD application domains, interactive configuration.

1 Introduction

Compiling a constraint satisfaction model into a tractable representation is useful for a number of tasks related to model analysis and decision support. Various forms of tractable structures have been suggested as target compilation languages, including automata [1], binary decision diagrams [2], and/or decision diagrams [3], and deterministic decomposable negation normal form (d-DNNF) [4].

In this paper we focus on compiling CSP models into *multivalued decision diagrams* (MDDs), as they are well suited for a number of decision support tasks [2,5]. We identify the tests of *infeasibility*, *entailment* and *equivalence* as critical for reasoning about the properties of various compilation schemes. We recognize that the *semantics* of global constraints can be utilized to enhance the compilation, and suggest using *incremental refinement* as a way of dealing with the weaknesses of semantic tests when compiling multiple constraints. Our incremental scheme generalizes both the search based BDD compilation of [9] and standard bottom-up compilation of [10]. We represent two different algorithms for achieving this, one that constructs a new MDD and one that refines the input MDD. The later algorithm makes use of the concept of *vertex splitting* which was first introduced in [8].

Because the full MDD can grow too large for practical use, we are particularly concerned with generating *approximate* MDDs that are limited in size but useful in applications. Additionally we are also concerned with generating approximate MDDs under

P.J. Stuckey (Ed.): CP 2008, LNCS 5202, pp. 448–462, 2008.

tight time requirements, since some of the constraints might be known only during user interaction. We show how equivalence checking offers a principled way to create approximate MDDs (approximate in the sense that they represent a superset of the feasible solutions). Rather than check for equivalence, we measure the "distance" between two partial assignments and view them as equivalent for algorithmic purposes when the distance is below a threshold. The distance measure is specialized to each constraint type, thus again allowing us to exploit special structure in the problem. The refinement process is an iterative one in which the threshold is gradually reduced. This injects a learning element, because the algorithm refines equivalence detection as it refines the MDD, thus allowing the next MDD to be more accurate. An exact MDD can be obtained by reducing the threshold to zero, or an approximate MDD by reducing the threshold to a positive number or terminating when the MDD exceeds a size limit. Terminating before obtaining the exact MDD still provides bounds on the degree of violation of each individual constraint.

We are not aware of related work utilizing explicitly the semantics of highly structured constraints for the purpose of compilation. The related compilation techniques enhance compilation by exploiting independencies among variables [3,9]. The idea is to recognize two partial assignments p_1, p_2 as equivalent when they assign same values to *critical* variables. In [9] the critical variables are determined by a *cutset* and in [3] by a *context* with respect to a *pseudo-tree* extracted from a constraint graph. We note however, that neither technique can enhance equivalence detection when presented when individual global constraints span all variables.

Some work has already been done on generic techniques for approximate compilation [6,7], but these techniques have two major drawbacks in relation to constraint models. Firstly, they conjoin individual constraints precisely until a threshold is reached, and only then start approximating. Therefore, they do not take all constraints into consideration. Secondly, since they are not relying on semantic information captured by highly structured constraints, they provide no guarantees regarding the degree of violation of individual constraints.

2 Preliminaries

A *multivalued decision diagram* (MDD) can be viewed as a branching tree in which isomorphic subtrees have been merged. The tree is constructed to find feasible solutions of a constraint set containing finite-domain variables $x_1, \ldots, x_n$. The tree branches on the variables in a fixed order $x_1, \ldots, x_n$. The branches at each node correspond to possible values of some variable x_j, or more generally, to disjoint subsets of possible values. To form the MDD, subtrees containing no feasible solutions are first deleted, and subtrees having the same shape are then merged to remove redundancy from the tree. Additional edges connect each vertex in the bottom layer to a single terminal vertex 1.

Thus an MDD for a constraint set S is a directed acyclic graph whose vertices are arranged in layers corresponding to the variables $x_1, \ldots, x_n$ in S. If vertex u lies in layer j (corresponding to x_j), we say $var(u) = x_j$, and each edge (u, v) leaving u corresponds to a subset D_{uv} of the domain D_j of x_j. The top layer consists only of the root vertex r, with $var(r) = x_1$. Each path $p = (u_1, \ldots, u_{n+1})$ from r to 1 is identified with the cartesian product $\prod_{j=1}^{n} D_{j,j+1}$, where $u_1 = r$ and $u_{n+1} = 1$. Every path p

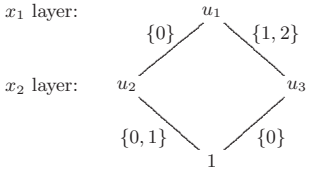

Fig. 1. MDD for $2x_1 + 3x_2 \leq 4$ with domains $x_i \in \{0, 1, 2\}$

from r to 1 in the MDD must *satisfy* S, meaning that every tuple $(x_1, \ldots, x_n)$ in p is a feasible solution of S. Conversely, every feasible solution of S belongs to some path from r to 1. For example, the MDD for the constraint $2x_1 + 3x_2 \leq 4$ (with domains $x_i \in \{0, 1, 2\}$) appears in Fig. 1.

We assume that the MDD is *reduced*, meaning that all isomorphic subtrees have been merged. To make this precise, let each vertex u in layer j of the MDD correspond to the function $f_u : D_j \times \cdots \times D_n \rightarrow \{0, 1\}$ defined by $f(x_j, \ldots, x_n) = 1$ when $(x_1, \ldots, x_n)$ belongs to a path from u to 1. Then $f_r(x_1, \ldots, x_n) = 1$ if and only if $(x_1, \ldots, x_n)$ satisfies S. Two vertices u, v in a given layer are *equivalent* if $f_u = f_v$, and the MDD is *reduced* if no two vertices in any layer are equivalent. When the variable ordering is fixed, there is a unique reduced MDD representing a given constraint set. It is common in the literature to remove vertex u in layer j (and join the two edges incident to u) when there is a single outgoing edge (u, v), and it has the property that $D_{uv} = D_j$. This results in "long edges" that skip one or more layers, but to simplify notation we do not remove any vertices in this fashion.

For convenience in describing the algorithms we further assume that the operation of choosing the edge corresponding to value α returns the special vertex $False$ if no such edge exists. When this vertex is included as a child in constructing a node, the semantics is to simply ignore this child. Furthermore we use $True(i)$ to indicate the MDD corresponding to the set of solutions $D_i \times \ldots D_n$. Given an MDD M and a partial assignment p to variables $x_1, \ldots, x_k$ we use M_p to denote the vertex reached in M when following the path corresponding to p. Given a constraint C, we also use C to denote the set of solutions to C. For a partial assignment p to variables $x_1, \ldots, x_k$ we use $C(p)$ to represent the solution space of $C(p)$ restricted to the assignments in p.

3 Top-Down Compilation of MDDs

An MDD is a compact representation of a branching tree for a given constraint set. The naive MDD construction based on first constructing the tree and then reducing it can be significantly improved by performing reductions *during search*. This has already been recognized in a more specific context, where a CNF formula is compiled into a binary decision diagram (BDD) using DPLL search with caching [9]. As a starting point in this paper, we suggest a generalized approach for compiling CSPs into MDDs. It is important to realize that the general compilation algorithm is based on three fundamental tests: recognizing when partial assignments encountered during search lead to *infeasible*, *(domain) entailed* or *equivalent* subbranches. Algorithm 1 emphasizes these tests during a depth first search (DFS) traversal of the branching tree.

Algorithm 1. `CompileDFS`(path p, int i, constraints S): A generic backtracking algorithm constructing an MDD for a set of constraints in a cached top-down manner. It is initiated with the call `Compile`(S) which just executes `CompileDFS`$(\emptyset, 0, S)$.

if $p \equiv_S 0$ **then**
  return $False$;
if $p \equiv_S 1$ **then**
  return $True(i)$;
key = $id_S(p)$;
result = `cache-lookup`(*key*);
if *result* $\neq$ *null* **then**
  return *result*;
Let $v_1, \ldots, v_k$ be the values in D_i;
result =
`get-vertex`$(i,$ `CompileDFS`$(p \times \{v_1\}, i+1), \ldots,$ `CompileDFS`$(p \times \{v_k\}, i+1)$;
`cache-insert`(*key, result*);
return *result* ;

We define a partial assignment p to be *infeasible* for a constraint set S ($p \equiv_S 0$) if it cannot be completed to an assignment in $p \times \prod_{i=k}^{n} D_i$ satisfying all constraints in S. In that case, the above algorithm returns $False$ indicating infeasibility. We say that p is *domain entailed* ($p \equiv_S 1$) if every completion of p satisfies S. In this case, we return $True(i)$ representing an MDD for the entire set of solutions $D_i \times \ldots \times D_n$.

Finally, p_1 and p_2 are said to be *equivalent* ($p_1 \equiv_S p_2$) if they have the same completions satisfying S. The equivalence test induces a set of equivalence classes among all partial assignments, and we use $id_S(p)$ to denote a unique identifier key for the equivalence class to which p belongs. An MDD is stored as a *cache* of keys that is maintained during search. A new node is created (using `cache-insert` and `get-vertex`) only if the current key cannot be found (using `cache-lookup`). While the equivalence class identifiers might be prohibitively large in general, in practice they are usually compact.

We say that tests for infeasibility, entailment, and equivalence are *sound* if every "yes" answer is correct, *complete* if every "yes" answer is recognized, and *efficient* if the test can be computed in polynomial time (with respect to the size of the MDD). The performance of Algorithm 1 critically depends on these three properties. Unsound tests lead to MDDs not representing the desired solution space. Incomplete tests make the algorithm traverse equivalent or infeasible parts of the search space. Inefficient tests increase the running time. Ideally, if we have sound and complete tests requiring constant time, and it is possible to represent equivalence classes efficiently then Algorithm 1 builds an MDD in output-optimal time and space. In the remainder of the paper we will use these tests as a basis for discussing the efficiency of various MDD compilation schemes.

4 Semantic Caching

Previous approaches that enhance equivalence tests are based on identifying variable independencies in the underlying model. Two partial assignments to variables $x_1, \ldots, x_{i-1}$

are equivalent if they assign the same values to variables on which x_i *critically* depends. The set of such variables might be much smaller than $\{x_1, \ldots, x_{i-1}\}$ and therefore, equivalence detection could be enhanced [3,9]. However, these approaches cannot be applied if *all variables* depend on each other. It suffices to introduce just a single global constraint, spanning over all variables, to get to this situation.

We argue that in addition to looking at the variable independencies, we should also consider the *semantics* of well-structured constraints. Namely, the CSP modeling vocabulary is full of constraints with rich structure, which is normally exploited during search through efficient filtering algorithms.

We illustrate how the same can be exploited for designing better compilation tests for *inequality*, *equality*, and *Alldiff* constraints. We will then discuss how to extend this to multiple constraints.

Inequality. An inequality constraint C has the form $\sum_i f_i(x_i) \leq b$, where each x_i is a finite domain integer variable and f_i is some cost function. For a given partial assignment $p = (v_1, \ldots, v_{k-1})$ to variables $(x_1, \ldots, x_{k-1})$, we denote the cost of p with respect to C as $a(p) = \sum_{i=1}^{k-1} f_i(v_i)$. A simple equivalence test for an inequality constraint C is

$$p_1 \equiv_C p_2 \Leftrightarrow a(p_1) = a(p_2).$$

The test is efficient but incomplete, because two equivalent partial assignments can be identified as nonequivalent. For example, $p_1 = (0)$ and $p_2 = (1)$ are equivalent for $x_1 + 2x_2 \leq 3$ (where $x_1, x_2 \in \{0, 1\}$), but they fail the above test for equivalence. We can formulate a complete equivalence test that requires pseudo-polynomial time. Assuming without loss of generality that $a(p_1) < a(p_2)$, the test is

$$p_1 \equiv_C p_2 \Leftrightarrow a(p_1) \leq b - a(p) < a(p_2) \text{ for no } p \in D_k \times \ldots \times D_n.$$

The following infeasibility test is both complete and efficient:

$$p \equiv_C 0 \Leftrightarrow a(p) + \mathit{SP}(p) > b. \tag{1}$$

where $\mathit{SP}(p) = \sum_{i=k}^{n} \min\{f_i(v) \mid v \in D_i\}$ is the *shortest path* in $D_k \times \ldots \times D_n$.

An analogous entailment test is also complete and efficient:

$$p \equiv_C 1 \Leftrightarrow a(p) + \mathit{LP}(p) \leq b. \tag{2}$$

where $\mathit{LP}(p) = \sum_{i=k}^{n} \max\{f_i(v) \mid v \in D_i\}$ is the *longest path* in $D_k \times \ldots \times D_n$.

Equality. The equivalence test

$$p_1 \equiv_C p_2 \Leftrightarrow a(p_1) = a(p_2). \tag{3}$$

for an equality constraint C (defined as $\sum_i f_i(x_i) = b$) is complete and efficient. The infeasibility test is essentially a subset sum problem:

$$p \equiv_C 0 \Leftrightarrow a(p) + a(p') \neq b \text{ for all } p' \in D_k \times \ldots \times D_n. \tag{4}$$

which is complete but inefficient (pseudo-polynomial). A complete and efficient domain entailment test checks whether all completions of the path have the same cost:

$$p \equiv_C 1 \Leftrightarrow \mathit{SP}(p) = \mathit{LP}(p). \tag{5}$$

Alldiff. Given a partial assignment $p = (v_1, \ldots, v_{k-1})$ we define $D(p) = \bigcup_{i=1}^{k-1} v_i$. We can now define a complete and efficient equivalence test for an Alldiff constraint C:

$$p_1 \equiv_C p_2 \Leftrightarrow D(p_1) = D(p_2).$$

Additionally we have the following complete and efficient infeasibility test.

$$p \equiv_C 0 \Leftrightarrow |\bigcup_{i=k}^{n} D_i| < n - k + 1.$$

Finally, a complete and efficient entailment test is given by

$$p \equiv_C 1 \Leftrightarrow D(p), D_k, \ldots, D_n \text{ are disjoint and nonempty.} \quad (6)$$

The above equivalence detection rules directly indicate how to compute $id_C(p)$ for a constraint C. In case of inequality or equality constraints, $id(p) = a(p)$, and for an Alldiff constraint it is $D(p)$.

Multiple Constraint Caching. The semantic tests described above can be directly generalized to a set of constraints $S = \{C_1, \ldots, C_m\}$:

$$p \equiv_S 0 \Leftrightarrow \bigvee_{i=1}^{m} (p \equiv_{C_i} 0),\ p \equiv_S 1 \Leftrightarrow \bigwedge_{i=1}^{m} (p \equiv_{C_i} 1),\ p_1 \equiv_S p_2 \Leftrightarrow \bigwedge_{i=1}^{m} (p_1 \equiv_{C_i} p_2).$$

The equivalence class identifier, $id_S(p)$, can be generically constructed as a tuple of individual keys, $id_S^{\times}(p) = (id_{C_1}(p), \ldots, id_{C_n}(p))$. In this case, Algorithm 1 detects the equivalence of two paths p_1, p_2 as soon as $id_{C_i}(p_1) = id_{C_i}(p_2)$ for each $C_i \in S$. This way of combining the individual tests ensures soundness but not completeness of the generic test even if individual tests are complete. Namely, the test allows for a number of "fake" equivalence classes that appear to be different even though they are the same. The potential number of fake classes explodes exponentially if individual tests are incomplete or as we add more constraints to S. For example, if id_S allows for K_f fake equivalence classes and K_e exact equivalence classes, and if we add to S a constraint C' with K' equivalence classes (all exact), then the resulting number of fake equivalence classes is at least $K_f \cdot K'$. Even among remaining $K_e \cdot K'$ pairs, there could be many fake classes.

We could partially remedy this if we could uncover interactions amongst the set of constraints rather than treating them independently. For example, infeasibility detection $p \equiv_S 0$ of a set of integer inequalities could be enhanced by checking for feasibility of their linear relaxation. In addition, if we can detect that some constraints become *entailed* by the remaining ones, we could ignore them when denoting the equivalence classes. In the following section we will show how this can be done efficiently in a very interesting special case.

5 Incremental Refinement

In the previous section we saw that the performance of Algorithm 1 critically depends on completeness of semantic tests, and that these tests become significantly weaker

when dealing with multiple constraints. In order to avoid the explosion of "fake" equivalence classes, we can make the compilation process *incremental*. We compile only a subset S' of S into an MDD M and insert this intermediate MDD into S instead of S'. Each vertex in the MDD represents an exact equivalence class for S' and we can take $id_{S'}(p) = id_M(p) = M_p$, allowing us to compute $id_S(p)$ as $id_M(p) \times id^{\times}_{S \setminus S'}(p)$ which can provide a reduction in the number of fake equivalence classes that is exponential in $|S'|$. This approach is illustrated in Algorithm 2. In each step it compiles a subset of constraints S' in the manner discussed above. We effectively have a number of different compilation approaches, ranging from compiling all constraints in one pass ($S' = S$), similar to [9], to pairwise conjunctions ($|S'| = 1$), which resembles the standard bottom-up compilation approach to building BDDs [10].

Algorithm 2. `IncrementalRefine`(M, S)

Data: Constraint set S
Result: MDD Representation of the Solution Space of S
$M \leftarrow True(1)$;
while $S \neq \emptyset$ **do**
  $S' \leftarrow$ some subset of S;
  $M \leftarrow$ `Compile`$(S' \cup \{M\})$;
  $S \leftarrow S \setminus S'$;
return M;

In the remainder of the paper we will focus on pair-wise operations, where one constraint C is combined with one MDD M in each step. This case is especially interesting as it allows us to create some very efficient tests for $S = \{M, C\}$, while in many cases retaining completeness. In particular, all individual constraint tests described previously relying on shortest or longest path computations of C can easily be generalized efficiently for $S = \{M, C\}$ in such a way to preserve completeness. This is achievable because it is easy to compute shortest and longest paths through an MDD as long as the cost function is separable [2]. For the Alldiff the domain entailment test remains complete, but the infeasibility test of Alldiff is no longer complete since it is an NP-complete problem to determine if an MDD contains a solution satisfying an external Alldiff constraint [8]. In addition, it is efficient to detect whether the MDD entails an inequality or equality since we can compute longest and shortest paths efficiently in the MDD, thereby providing a further reduction in the fake equivalence classes. The same is possible for the Alldiff. For each MDD vertex u in layer $l(u)$ we can efficiently compute the set $D(u)$ of values occuring on *all* paths to u. The MDD then entails the Alldiff constraint iff $|D(u)| = l(u) - 1$ for all nodes u, that is iff a distinct set of values leads to each node.

As previously mentioned the above approach of pair-wise compilation closely resembles standard bottom-up compilation. We do however have two significant advantages. Firstly, the standard approach requires each constraint to be represented as an MDD. To see why this is a problem in itself, consider an Alldiff constraint combined with a lexical ordering constraint. The conjunction of these only allows a single solution, but if we build the Alldiff separately, we will require exponential time and space. If

we on the other hand, build the MDD for the lexical ordering constraint first (yielding a polynomial size MDD), we can efficiently compute the conjunction as most of the equivalence classes from the Alldiff need never be considered since they are disallowed by the lexical ordering constraint. Secondly, the semantic information allows us to detect domain- and general-entailment of some interesting constraint types more efficiently. For example, detecting that an inequality is entailed by an MDD is more efficient if it is represented symbolically rather than as an MDD.

5.1 Vertex Splitting

The algorithm described above operates by always constructing an entirely new MDD, instead of updating the input MDD, even when differences between them are only minor. We can try to minimize redundant work by modifying the input MDD rather than creating a new one. We do this by identifying non-equivalent paths ending in the same vertex, and then *splitting* it.

Figure 2 illustrates a vertex-split and the separation of nonequivalent paths for an Alldiff constraint. The edge (u_4, u_5) is, for algorithmic purposes, regarded as two edges that correspond respectively to values 1, 2. If two or more paths coming into u_5 are nonequivalent, we will split u_5 into two vertices in order to refine the MDD. In this case, the paths (u_1, u_2, u_5) and (u_1, u_3, u_5) are equivalent, but other pairs of paths are nonequivalent. We therefore split u_5 into three vertices and distribute the incoming edges between these two vertices in such a way that no two edges coming into a vertex are nonequivalent. No fewer than three vertices will accomplish this.

This algorithm is shown in Algorithm 3 and replaces `Compile`. It traverses the MDD in a breadth-first manner (BFS) manner. Instead of considering the equivalence classes of partial assignments (correspond to paths), it considers equivalence classes of edges (considering an edge (u, v) where $|D_{uv}| > 1$ as $|D_{uv}|$ seperate edges). Since the algorithm always splits the previous layer completely before splitting nodes in the next layer, it is guaranteed that all partial assignments ending in a given edge in the next layer belong to the same equivalence class. Therefore the edge can be considered to be identical to any of these paths for the purpose of equivalence, entailment and

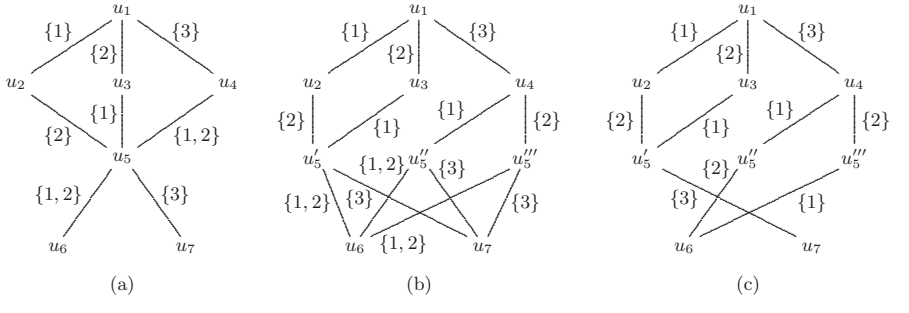

Fig. 2. (a) Part of an MDD just before splitting vertex u_5 with respect to an Alldiff constraint. (b) The edges coming into vertex u_5 have been partitioned into three equivalence classes, and u_5 split into three vertices to receive them. (c) After the split we can prune some infeasible values.

Algorithm 3. `SplitCompile`(M, C)

```
Data: MDD M, constraint C
Result: Refined MDD
if M ⇒ C then
    return M;
foreach vertex u ∈ M in layer-by-layer top-down order do
    foreach e ∈ In(u) do
        if e ≡_C 0 then
            delete e from M and from In(u);
    if In(u) = ∅ then
        delete u from M;
    else if In(u) ≢_C 1 then
        Partition In(u) into sets E_1, ..., E_m such that for each e, e' ∈ E_i, e ≡_C e';
        Split(M, u, E_1, ..., E_m);
return Reduce(M);
```

Algorithm 4. `Split`$(M, u, E_1, \ldots, E_m)$

```
Data: MDD M, vertex u
for i = 1 ... m do
    Create a new vertex u_i in u's layer of M;
    for edges (u, u') of M do
        Add edge (u_i, u') to M with D_{u_i u'} = D_{u u'};
    for (u', u) ∈ E_i do
        Remove edge (u', u) from M;
        Add edge (u', u_i) to M with D_{u' u_i} = D_{E_i};
for edges (u, u') of M do
    Remove edge (u, u') from M;
```

infeasibility detection. The previously described entailment detection is now done prior to the vertex splitting. Since reduction is not done during splitting, this is performed just before returning the MDD. In our experiments we will rely on this vertex splitting based algorithm to implement the pair-wise conjunction of Algorithm 2.

6 Approximate Compilation

Even when we can compile an MDD for a constraint set using iterative refinement, the MDD may be too large or too hard to compute for practical purposes. This may occur, for example, in an online setting where there is insufficient time or memory to compute an exact MDD. We therefore propose to modify iterative refinement for *approximate semantic compilation*. For given memory and time restrictions we compile an MDD that represents a superset of the solution space. In particular, we produce a sequence of approximate MDDs, each a refinement of the last in the sense that it represents a smaller

superset of the solution space. Each approximate MDD is created by taking all constraints into consideration, thus taking advantage of interactions among the constraints. We also provide approximation guarantees with respect to each constraint.

The basic idea is to regard two partial assignments p_1, p_2 as equivalent for algorithmic purposes when their *distance* is below a threshold $d^C_{\max}$. Thus the equivalence test becomes

$$p_1 \equiv_C p_2 \Leftrightarrow \text{distance}_C(p_1, p_2) \leq d^C_{\max}.$$

A definition of edge equivalence is induced from equivalence of partial assignments in the same way as before. Distance measures are specialized to each type of constraint. For an inequality constraint $\sum_{i=1}^{n} f(x_i) \leq b$, the distance will be

$$\text{distance}_{\leq}(p_1, p_2) = |a(p_1) - a(p_2)|$$

and similarly for an equality constraint. For Alldiff constraints we can use symmetric difference as a measure of distance:

$$\text{distance}_A(p_1, p_2) = |D(p_1) \triangle D(p_2)|.$$

where $S_1 \triangle S_2 = (S_1 \setminus S_2) \cup (S_2 \setminus S_1)$. Other distance measures could be used as well.

When equivalence is detected in this fashion, Algorithm 1,2 and 3 becomes approximate MDD compilers. The resulting MDD guarantees that any two paths entering the same vertex differ by at most $d^C_{\max}$ with respect to constraint C. If the infeasibility test is complete, then we create no infeasible vertices, and the number of redundant equivalence classes can be limited as desired by adjusting the bound $d^C_{\max}$.

The overall procedure for approximate compilation begins with a trivial MDD M (consisting of the single vertex *True*(1)) and some initial large distance. It then refines M using `SplitCompile` for each of the constraints in S using the distance based equivalence tests. The process is then repeated with lower distance thresholds, obtained from the previous thresholds by, for example, a multiplicative or additive factor.

The advantage of this approach, regardless of whether the goal is exact or approximate compilation, is that after one distance is processed, the resulting MDD takes all constraints into consideration. This means that we at any time have a bound on the degree of violation on each constraint in the current MDD. In addition, it allows obvious inconsistencies to be removed from the solution space at an earlier time, preventing the corresponding equivalence classes from taking up computatation time in subsequent steps. Therefore it can in fact be advantageous to compute an exact MDD through a sequence of approximations in which the distance thresholds are gradually reduced to zero.

7 Experiments

In this section we will show how the presented techniques perform in practice for a selection of applications. Implementation of techniques discussed in this paper is using our own MDD compiler and generic MDD-manipulation package, written in Java. Comparisons to standard compilation techniques makes use of CLab [11].

7.1 Approximation Quality Tradeoff

In the first set of experiments we evaluate the overall quality of our approximation scheme. For an MDD M, and a constraint C we create an approximate MDD M_{apx} with increasing precision (decreasing maximal distance threshold $d^C_{\max}$) and without size limit ($T_{max} = \infty$). For each $d^C_{\max}$ we generate the approximate MDD and report its number of edges and solutions. The results are shown in Figure 3, and we can for example see that the solution count decreases super-linearly as a function of MDD size.

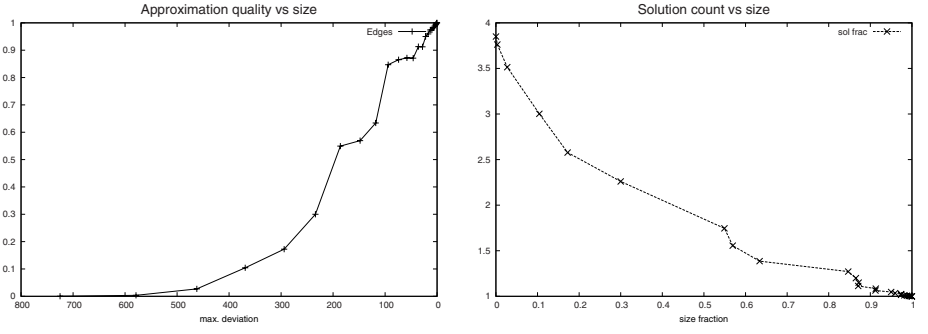

Fig. 3. The two plots above tracks the progress of the approximate compilation process of 5 random separable inequalities, with 15 variables over domains of size 3 and matrix elements from the range -100 to 100. The leftmost plot shows the approximatedistance achieved on the horizontal-axis and the size of the MDD on the vertical axis. Since the instance consists of inequalities, the distance corresponds to an upper bound on how much the longest path can violate the bound. The rightmost plot shows the trade-off achieved between solution count and MDD size.

7.2 Approximate Refining for Exact Compilation

In the second set of experiments we illustrate how approximative refining can be a competitive method for exact compilation. We postulated previously that the use of approximate refinement steps with distance thresholds gradually reducing to 0 might be beneficial for exact compilation. We therefore compared the CLab compilation approach, precise refining, and approximate refining for a single randomly generated linear inequality, as well as for a set of linear inequalities over binary variables. Compiling a single inequality might be relevant for assisting a standard compiler (such as CLab) in compiling individual rules, while the set of linear inequalities illustrates behavior when we have weak equivalence detection due to a lack of strong semantics. The results are shown in Figures 4(a) and 4(b).

For a single inequality, we can observe that the number of vertices created by CLab is nearly unaffected by tightness. This is due to the mechanism used in CLab for constructing the BDD for an inequality, which compiles a BDD for each bit of the left-hand side and then builds the BDD by comparing these with the bit representation of the right-hand side. We can also see that both precise and approximate compilation outperform CLab in the number of vertices generated. With regard to time (which is not shown), the approximate compiler outperforms CLab on tightness less than 0.2 and greater than 0.8.

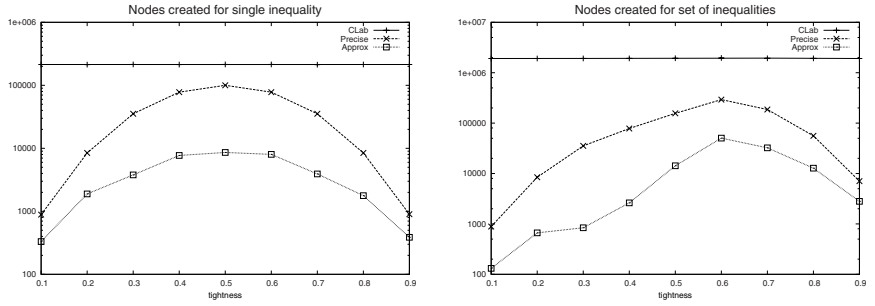

Fig. 4. (a) Total number of vertices created during compilation of a single random linear inequality over 18 variables with binary domains. The coefficients range from 0 to 100000. (b) Total number of vertices created during compilation of 10 random linear inequalities over 18 variables with binary domains. The coefficients range from 0 to 100000. Instances with tightness up to and including 0.4 are unsatisfiable.

The second experiment considers a set of linear inequalities. Again we observe that CLab is almost unaffected by the tightness of the constraints. This is again due to the construction mechanism mentioned before. In fact more than 99% of the vertices created by CLab are generated during the construction of BDDs for individual inequalities, and most of these vertices are created before considering the right-hand side of each constraint. The precise vertex-splitting based compiler produces far fewer vertices, while approximate compilation reduces this number even further, clearly outperforming precise compilation. With regards to time (not shown), the (approximate) vertex-splitting compiler is fastest up to and including tightness 0.5 and again for tightness greater than 0.8. CLab is fastest between 0.5 and 0.8. Note, however, that CLab is based on highly optimized C code, while our Java implementation is far from optimized.

7.3 Interactive Configuration

In our final set of experiments we assess the usefulness of approximative MDD compilation for one of its main application areas: interactive configuration. We consider a scenario where an MDD M^{init} is given for an initially compiled configuration instance along with a set S of external (resource) constraints, which have not been compiled either because the resulting MDD is too large, or the constraints are not known at the time of compilation.

In the presence of external constraints, it is NP-hard to prune all *non-GAC* values; that is, values that are not *generalized arc consistent* with respect to the conjunction $M^{init} \wedge S$ of all constraints. A user is therefore exposed to backtracking, because he is presented with non-GAC values as valid options due to incomplete (but time efficient) pruning algorithms. This often occurs in practice[1] and is regarded negatively. We therefore explore whether approximate compilation can be used to remove non-GAC values while still observing strict time and memory limitations.

[1] Think of buying an airplane ticket online and getting the message, "There is no flight on selected dates. Please go back and try again."

After each user assignment, we compute initial valid domains, and while the user is assessing available options we refine the existing MDD with respect to S to get refinement M^{apx}. This MDD is then additionally cost-pruned with respect to each constraint $C \in S$, in the sense of cost-bounded configuration [2], and the domains displayed to the user are updated. As an alternative to approximate compilation, we consider computing valid domains only with respect to the initial MDD M^{init}, or with respect to M^{init} after cost pruning. We abbreviate the first scenario as *ApxP* (approximation + cost pruning) and denote the other two as *Init* and *InitP*, respectively. The approach of the last scenario in itself leads to strictly more pruning than in the case of standard CSP propagation, in which the MDD and the constraints in S are posted individually as global constraints.

For the initial MDD M^{init} we loaded an MDD representing the real-world configuration instance "PC" (a personal computer configuration problem), available in the CLib benchmark suite [12]. It has 45 finite-domain variables of up to 33 domain values and 4875 vertices. We then generated a set S of external constraints. For each $m \in \{2, 3, \ldots, 13\}$ we generated 10 models of m random separable inequalities, each with a tightness $t = 0.5$. For a separable cost expression $\sum_i c_i(x_i)$ we set the right-hand side bound to $b = min_c + (max_c - min_c) \cdot t$, where min_c and max_c are the minimal and the maximal value of the cost function c. We set the maximal vertex size threshold T_{max} to 5000. For each set of separable inequalities we measured a number of parameters averaged over 100 interaction simulations (where in each simulation we randomly simulated user assignments until there was only one solution left). In Table 1 we report, for each number of constraints m, the median of these values over the 10 generated models.

Table 1. Effect of approximate compilation on reducing the non-GAC values in user interaction. Column m indicates the number of external constraints $\mathcal{C}$. M^{apx} is the maximal size of an approximate MDD encountered. M^e is the size of the MDD representing entire conjunction exactly $M^{init} \wedge \mathcal{C}$. Columns *Init*, *InitP* and *ApxP* denote the probability of selecting non-GAC value for the three scenarios previously described. Column *Subsume* indicates the average subsumption depth, i.e. after how many assignments does approximate MDD become exact. Finally, columns *Refine* and *Reduce* indicate the number of seconds spent for generating approximate M^{apx} and subsequent elimination of redundant equivalence classes.

m	M^{apx}	M^e	*Init(%)*	*InitP(%)*	*ApxP(%)*	*Subsume*	*Refine(s)*	*Reduce(s)*
13	7894	732	18	7	0.5	1.17	1.27	0.48
12	5838	253	19	6.9	0	0	1.07	0.48
11	5616	872	18	6.3	0	0	0.75	0.43
10	6081	2471	18	6.8	0.1	1	0.89	0.39
9	5031	258	19	5.1	0	0	0.86	0.36
8	7474	3676	16	4.8	0.02	1.45	0.94	0.40
7	6925	2849	16	4.7	0.02	1.43	0.70	0.31
6	6827	7797	14	2.5	0.02	1.62	0.64	0.28
5	7112	17965	11	2.0	0.01	2.17	0.56	0.24
4	7336	25030	11	1.8	0.02	2.42	0.48	0.21
3	7957	35092	9.8	0.82	0.006	2.56	0.42	0.18
2	7231	43108	6.2	0.22	0.0002	3.08	0.29	0.13

The probability of selecting a non-GAC value was assessed by comparing, for every unassigned variable, the size of the domains represented to the user (D^{init}, D^{initP}, and D^{apxP}) against the actual number of non-GAC values D^e. More precisely, if domain D_i is shown to the user, but only a subset D_i^e of values are GAC, then we compute the probability $\frac{|D_i|-|D_i^e|}{|D_i|}$ of selecting a non-GAC value with respect to a single variable. We then average the probability over all unassigned variables and repeat this for every assignment in a simulation. If U_j was the set of unassigned variables at interaction step j, and there were a total of k assignments when the solution was completely specified, we compute:

$$\frac{1}{k}\sum_{j=1}^{k}\frac{1}{|U_j|}\cdot\sum_{i\in U_j}\frac{|D_i|-|D_i^e|}{|D_i|}$$

as the probability of selecting a non-GAC value in an interaction simulation.

Table 1 indicates that approximate compilation almost entirely eliminates the probability of backtracking. On average, scenario *ApxP* using approximate compilation reduces by several orders of magnitude the probability of selecting a non-GAC value, compared to the *InitP* and especially the *Init* scenario. While *InitP* performs well for a smaller number of constraints (below 1% for two constraints), the probability of backtracking increases with the number of constraints (7% for 13 constraints). Computing domains over initial MDD in *Init* scenario leads to a significant backtracking probability that increases with the number of constraints, up to 19%. Subsumption depth for approximate compilation is very shallow. After an average of 1-3 assignments, the MDD becomes exact. Since we fixed the tightness of individual constraints, the overall tightness of the solution space increases with the number of constraints. As a result, exact MDDs get increasingly smaller, while approximate MDDs are relatively stable. The combined running time for refinement and reduction phase is usually below 1.5 seconds, which is more than acceptable in our interaction setting: we first show initial domains, and while the user is investigating those, we further refine based on an approximate MDD.

8 Conclusions

We presented an incremental refinement algorithm based on vertex splitting, for approximate compilation of constraint satisfaction models into MDDs. The presented approach utilizes the semantics of constraints and a notion of distance to obtain approximate MDDs. Our empirical evaluation demonstrated that approximate refinement can be a competitive compilation method and that significant reductions in backtracking can be made by approximately compiling external constraints during interactive configuration.

Acknowledgments. Tarik Hadzic is supported by an IRCSET/Embark Initiative Postdoctoral Fellowship Scheme. Barry O'Sullivan is supported by Science Foundation Ireland (Grant Number 05/IN/I886).

References

1. Vempaty, N.R.: Solving constraint satisfaction problems using finite state automata. In: Proceedings of the Tenth National Conference on Artificial Intelligence, pp. 453–458 (1992)
2. Hadzic, T., Andersen, H.R.: A BDD-based Polytime Algorithm for Cost-Bounded Interactive Configuration. In: Proceedings of AAAI 2006 (2006)
3. Mateescu, R., Dechter, R.: Compiling constraint networks into AND/OR multi-valued decision diagrams (AOMDDs). In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 329–343. Springer, Heidelberg (2006)
4. Darwiche, A., Marquis, P.: A knowledge compilation map. Journal of Artificial Intelligence Research 17, 229–264 (2002)
5. Hadzic, T., Hooker, J.N.: Cost-bounded binary decision diagrams for 0-1 programming. In: Hentenryck, P.V., Wolsey, L.A. (eds.) CPAIOR 2007. LNCS, vol. 4510, pp. 84–98. Springer, Heidelberg (2007)
6. Ravi, K., Somenzi, F.: High-density reachability analysis. In: ICCAD 1995: Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design, Washington, DC, USA, pp. 154–158. IEEE Computer Society, Los Alamitos (1995)
7. Ravi, K., McMillan, K.L., Shiple, T.R., Somenzi, F.: Approximation and decomposition of binary decision diagrams. In: Design Automation Conference, pp. 445–450 (1998)
8. Andersen, H.R., Hadzic, T., Hooker, J.N., Tiedemman, P.: A Constraint Store Based on Multivalued Decision Diagrams. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 118–132. Springer, Heidelberg (2007)
9. Huang, J., Darwiche, A.: DPLL with a trace: From SAT to knowledge compilation. In: Kaelbling, L.P., Saffiotti, A. (eds.) IJCAI, pp. 156–162. Professional Book Center (2005)
10. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers (1986)
11. Jensen, R.M.: CLab: A C++ library for fast backtrack-free interactive product configuration (2007), `http://www.itu.dk/people/rmj/clab/`
12. CLib: Configuration benchmarks library (2007), `http://www.itu.dk/research/cla/externals/clib/`