Approximate Compilation of Constraints into Multivalued Decision Diagrams

Tarik Hadzic[1], J. 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 detection of equivalent paths in the MDD. Although the algorithm is quite general, it can be adapted to exploit constraint structure by specializing the path equivalence test 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]. Because the full MDD can grow too large for practical use, we are particularly concerned to generate *approximate* MDDs that are limited in size but useful in applications. We are also concerned with generating approximate MDDs under tight time requirements, since some of the constraints might be known only during user interaction.

We propose a general compilation method, based on iterative refinement, for building approximate MDDs—and exact MDDs as a special case. We begin with a trivial MDD and progressively refine it to represent the constraint set more accurately. The key refinement technique is *vertex splitting*.

Recent work introduces various types of vertex splitting to create an MDD for use as a constraint store [6] and for postoptimality analysis [7]. However, these splitting techniques are ad hoc and based on the particular characteristics of the application. Because it is important to exploit the structure of particular constraints for the sake of efficiency, different splitting techniques were invented for each new constraint class. This experience has taught us, however, how to view the splitting operation from a broader

perspective. We present here a general splitting technique for MDD compilation, and a principled method for relaxing the technique to obtain approximate MDDs.

Although the vertex splitting algorithm is quite general, it is readily adapted to take advantage of special structure in constraints. The algorithm checks whether paths in the MDD leading to the same vertex, which correspond to partial assignments to the variables, are *equivalent*. If not, the vertex is "split" by generating multiple copies of the vertex that correspond to equivalence classes of paths. The key to this operation is recognizing when two paths are equivalent, and the equivalence test can be specialized to particular types of constraints. Two paths are viewed as equivalent when they pass the test for each constraint in the problem.

The same equivalence checking operation 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 paths 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.

After some preliminaries we describe below a basic top-down approach to MDD compilation and how it relies on equivalent path detection. The efficiency of the top-down approach depends on caching paths that have already been generated, and therefore on the precision with which equivalent paths can be detected. We then indicate how equivalence detection can be specialized to the semantic properties of particular constraints, which results in a process we call *semantic caching*.

We then present a general vertex splitting algorithm that likewise uses the semantic properties of constraints to check equivalence. Repeated application of vertex splitting leads to *semantic refinement* of the MDD. This can be superior to semantic caching in a top-down algorithm when processing a *set* of constraints. It can also be superior to a standard bottom-up compilation method in which MDDs are created for individual constraints and then conjoined. At this point we show how constraint-specific distance measures can be introduced to obtain approximate MDDs in an iterative process.

We conclude with empirical evaluation. We first illustrate a tradeoff between approximation quality and size that can be achieved through approximate compilation. We then demonstrate the benefits of approximate refining as an exact compilation method. Finally, we demonstrate the value of our approach in one of the primary applications of MDDs, interactive configuration.

2 Multivalued Decision Diagrams

A *multivalued decision diagram* (MDD) can be viewed as a branching tree in which isomorphic subtrees have been superimposed. The tree is constructed to find feasible solutions of a constraint set containing finite-domain variables $x_1, \ldots, x_n$. The tree

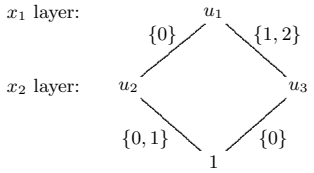

Fig. 1. MDD for $2x_1 + 3x_2 \leq 4$ with domains $x_i \in \{0, 1, 2\}$.

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 superimposed 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 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 superimposed. To make this precise, let each vertex u in layer j of the MDD correspond to the function $f_u : D_j \times \cdots \times D_n \to \{0, 1\}$ 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.

3 Top-Down Compilation of MDDs

An MDD can be created for a given constraint set in a recursive fashion from the top down. The key to doing so is to recognize when partial assignments are tautologous, infeasible, or equivalent.

Suppose that we are in the midst of constructing an MDD for constraint set S and have reached layer $k - 1$. Thus we have a directed acyclic graph M with root r and layers $1, \ldots, k-1$ corresponding to variables $x_1, \ldots, x_{k-1}$. A path $p = (u_1, \ldots, u_{k-1})$

Algorithm 1: `Compile`(path p, int i): Compiles constraint set S into an MDD. The initial call is `Compile`$(\emptyset, 1)$

```
if p ≡_S 0 then
    return False;
if p ≡_S 1 then
    return True(i) vertex;
key = compute-key(p);
result = cache-lookup(key);
if result ≠ null then
    return result;
Let v_1, ..., v_k be the values in D_i;
result = get-vertex(i, Compile(p × {v_1}, i + 1), ..., Compile(p × {v_k}, i + 1));
cache-insert(key, result);
return result ;
```

from r to any vertex in the bottom layer of M is identified with the cartesian product $\prod_{i=1}^{k-2} D_{u_i,u_{i+1}}$. Any tuple in p is a partial assignment to the variables, specifically to variables $x_1, \ldots, x_{k-1}$.

We define a path p from r to the bottom layer of M to be *infeasible* for a constraint set S ($p \equiv_S 0$) if it cannot be completed to a feasible assignment; that is, no tuple in $p \times \prod_{i=k}^{n} D_i$ satisfies S. Also p is *tautologous* ($p \equiv_S 1$) if every completion is feasible; that is, $p \times \prod_{i=k}^{n} D_i$ satisfies S. Finally, p_1 and p_2 are *equivalent* ($p_1 \equiv_S p_2$) if they have the same feasible completions. Then $p_1 \equiv_S p_2$ if for every $p \in \prod_{i=k}^{n} D_i$, $p_1 \times p$ satisfies S if and only if $p_2 \times p$ satisfies S.

We suppose that tests for infeasibility, tautology, and equivalence are available. These tests are *sound* if every "yes" answer is correct, *complete* if every "yes" answer is recognized, and *efficient* if the test can be efficiently computed in some sense. In this case we can build an MDD in output-optimal time and space using a procedure similar to Algorithm 1.

The algorithm traverses partial assignments using depth-first search and creates a vertex of the MDD after all of its children have been created. If the current path p is infeasible, we do not create an edge to the corresponding vertex. If p is tautologous, we create a True(i) vertex that roots a subtree representing $D_{i+1} \times \cdots \times D_n$. We then check whether a path equivalent to p has already been traversed to the current level. We do this by computing a key for p and checking whether the cache contains a path with this key. If so, we retrieve the corresponding vertex.

4 Semantic Caching

The running time of the above top-down algorithm depends critically on the completeness and efficiency of the tests for infeasibility, tautology and equivalence. If the tests are incomplete, the algorithm may traverse equivalent or infeasible parts of the search space. Inefficient tests obviously increase the running time as well.

In many cases there is little prospect of having both efficient and complete tests. An infeasibility test is essentially a satisfiability check and is NP-hard for many interesting

constraints. Equivalence testing is strictly harder. It subsumes infeasibility checking, and it can be hard when infeasibility detection is easy (see the discussion of inequality constraints below).

Two recent approaches to knowledge compilation enhance equivalence detection by exploiting independencies among variables [3, 8]. These techniques can accelerate compilation into binary decision diagrams, free BDDs, deterministic decomposable negation normal form (d-DNNF), and/or MDDs. They recognize two partial assignments p_1, p_2 as equivalent when they assign same values to *critical* variables. In [8] [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.

While using variable independencies to accelerate knowledge compilation can be effective, we argue that further improvement can be obtained by exploiting the *semantics* of constraints, even when no variable independencies are present. The more we know about the structure of a constraint, the more we can enhance equivalence detection. This is particularly true for highly-structured global constraints in constraint programming models. We illustrate this idea by discussing compilation of MDDs for inequality, equality, and alldiff constraints.

For purposes of this section, we will assume that we branch on x_i by fixing x_i to one value at a time, as in Algorithm 1. This means that a path p in the MDD is a single tuple of values, and D_{uv} is a singleton for each edge (u, v).

Inequality An inequality constraint has the form $ax \leq b$, where $ax = \sum_{i=1}^{n} a_i x_i$ and each x_i is a finite domain integer variable. 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 a as $a(p) = \sum_{i=1}^{k-1} a_i v_i$.

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

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

The following infeasibility test is both complete and efficient:

$$p \equiv_C 0 \Leftrightarrow a(p) + \mathit{SP}(D_k \times \ldots \times D_n) > b \tag{1}$$

where $\mathit{SP}(D_i \times \ldots \times D_n) = \sum_{i=1}^{n} \min\{a_i v \mid v \in D_i\}$ is the *shortest path* in the remaining solution space. If every D_i contains 0, this reduces to $p \equiv_C 0 \Leftrightarrow a(p) > b$.

An analogous tautology test is also complete and efficient:

$$p \equiv_C 1 \Leftrightarrow a(p) + \mathit{LP}(D_k \times \ldots \times D_n) \leq b \tag{2}$$

where $\mathit{LP}(D_k \times \ldots \times D_n) = \sum_{i=1}^{n} \max\{a_i v \mid v \in D_i\}$ is the *longest path* in the remaining solution space.

Equality The equivalence test

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

for an equality constraint C is complete and efficient. The infeasibility test is essentially a subset sum problem:

$$p \equiv_C 0 \Leftrightarrow a(p) + a(p') = b \text{ for no } p' \tag{4}$$

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

$$p \equiv_C 1 \Leftrightarrow \mathit{SP}(D_i \times \ldots \times D_n) = \mathit{LP}(D_i \times \ldots \times D_n) \tag{5}$$

Alldiff If path $p = (u_1, \ldots, u_{k-1})$, we define $p(x_i)$ to be the singleton containing the value of x_i in the path; that is, $p(x_i) = D_{u_i u_{i+1}}$ for $i = 1, \ldots, k-1$. Assuming that the values $p_1(x_1), \ldots, p_1(x_{k-1})$ already assigned are distinct, and similarly for $p_2(x_1), \ldots, p_2(x_{k-1})$, we can use the following complete and efficient equivalence test for an alldiff constraint C:

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

where $D(p) = \bigcup_{i=1}^{k-1} p(x_i)$ is the set of values used in the path p. Again assuming that $p(x_1), \ldots, p(x_{k-1})$ are distinct, we have the complete and efficient infeasibility test

$$p \equiv_C 0 \Leftrightarrow \text{alldiff}(x_k, \ldots, x_n) \text{ has no solution when each } x_i \text{ has domain } D_i \setminus D(p)$$

Under the same assumption, a complete and efficient tautology test is

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

The above equivalence detection rules indicate what to store as a key in the cache. In case of inequality or equality constraints, the key is a *constant* corresponding to a cost, and for an alldiff constraint it is a *set* of assigned values.

Semantic caching is designed to exploit the structure of a particular constraint in the context of a top-down compilation algorithm. When we wish to compile a set of constraints, the equivalence test can be run separately for each constraint. Two paths are then regarded as equivalent if they are equivalent for every constraint. If all of the individual tests are complete, then this provides a complete equivalence test for the constraint set as a whole.

5 Incremental Refinement

Semantic caching can improve the performance of top-down compilation by reducing the search space. However, if equivalence detection is incomplete, the top-down algorithm can revisit parts of the search space multiple times. This problem is compounded

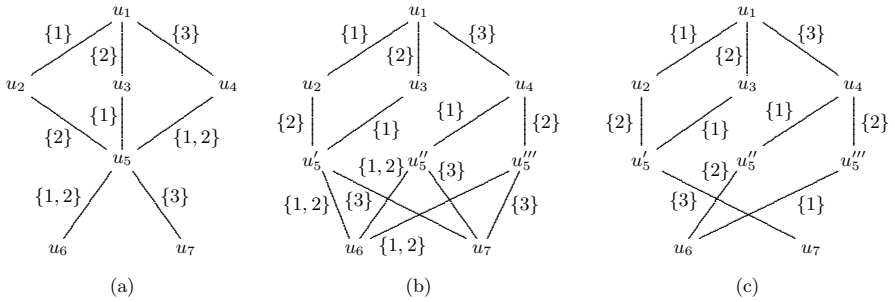

Fig. 2. (a) Part of an MDD just before splitting vertex u_5 with respect to an alldiff constraint. (b) The edges coming into vertex u_5 have been partitioned into three equivalence classes, and u_5 split into three vertices to receive them. (c) After the split we can prune some infeasible values.

when there are several constraints, because the combined equivalence tests are likely to be even weaker for the constraint set as a whole than for the individual constraints.

Another difficulty is that even if each individual constraint generates a small number of cache keys, the number of cache entries for the constraint set can grow exponentially. Each key for the constraint set is a tuple of keys for the individual constraints.

There are also difficulties with the standard bottom-up approach to compiling a constraint set. It compiles each constraint individually into an MDD, and then repeatedly applies a pairwise conjunction operator to create an MDD for the entire set. Each individual MDD is conjoined with an MDD that represents the constraints already conjoined. Unfortunately, these intermediate MDDs can explode in size.

We therefore propose an alternate approach to MDD compilation that can potentially avoid the large intermediate requirements of standard bottom-up compilation and tries to cope with incomplete equivalence detection. We suggest an incremental *semantic refinement* method that starts from a tautological MDD $True(0)$, and every time it detects two nonequivalent paths ending in the same vertex, it *splits* the vertex. As the process continues, the MDD is progressively refined until it accurately represents the constraint set.

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.

The definition of path equivalence is the same as before, except that we consider only completions that are part of the current MDD. That is, two paths p_1, p_2 terminating at u are equivalent for constraint C when for any path p from u to 1 in the MDD, $p_1 \times p$ satisfies C if and only if $p_2 \times p$ satisfies C. It is convenient to say that two edges (u_1, u), (u_2, u) are nonequivalent when some path from r to u through (u_1, u) is not equivalent

Algorithm 2: `SemanticRefine`(M, C)

Data: MDD M, constraint C
Result: Refined MDD
foreach vertex $u \in M$ **do**
  foreach $e \in In(u)$ **do**
    if $e \equiv_C 0$ **then**
      delete e from M and from $In(u)$;
  if $In(u) = \emptyset$ **then**
    delete u from M;
  else if $In(u) \not\equiv_C 1$ **then**
    Partition $In(u)$ into sets $E_1, \ldots, E_m$ such that for each $e, e' \in E_i$, $e \equiv_C e'$;
    `Split`$(M, u, E_1, \ldots, E_m)$;
return M;

Algorithm 3: `Split`$(M, u, E_1, \ldots, E_m)$

Data: MDD M, vertex u
for $i = 1 \ldots m$ **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 $D_{u_i u'} = D_{uu'}$;
  for $(u', u) \in E_i$ **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;

to some path from r to u through (u_2, u). Then we split a vertex u when at least two edges coming into u are nonequivalent.

Infeasibility is similarly defined. A path p from r to u is infeasible for C if $p \times p'$ is feasible for no path p' from u to 1 in the MDD. Tautology becomes *satisfaction*. A path p from r to u satisfies C if $p \times p'$ satisfies C for every path p' in the MDD from u to 1.

The overall splitting procedure is given as Algorithm 2, where $In(u)$ denotes the set of edges coming into vertex u. The subprocedure `Split` appears as Algorithm 3.

Note that if there is no MDD substructure under a vertex u in layer k (i.e., the paths from u to 1 correspond to the cartesian product $D_k \times \ldots \times D_n$) a vertex split at layer k corresponds to creating branches at layer $k-1$. Since there is only one child vertex $True(k+1)$ and all outgoing edge values are available, there is no need to add children explicitly. Every value in the current domain of $var(u)$ is valid. Thus if there is no substructure, splitting behaves the same as branching in a search tree. However, if there is substructure, splitting helps to reuse it.

If vertices are processed from top down, and if the same equivalence detection is used, the incremental refinement procedure (Algorithm 2) explores the same solution space as Algorithm 1. However, incremental refinement generates all the vertices it

Algorithm 4: `Compile`

```
Data: A set S of constraints
Result: MDD Representation of Solution Space
M ← True(0);
foreach C ∈ S do
    M ← SemanticRefine(M, C);
    if M = ∅ then
        return false;
return M;
```

visits, while Algorithm 1, due to its depth-first traversal, visits the same vertices but does not generate them until proving that they are part of the MDD.

Incremental refinement can behave very differently, however, on a *set* of constraints. It explores the same solution space if it checks equivalence by checking it with respect to all constraints:

$$p_1 \not\equiv_S p_2 \Leftrightarrow p_1 \not\equiv_C p_2 \text{ for some } C \in S \tag{7}$$

But an alternative method is to process each constraint individually, in a semantic version of bottom-up compilation. This differs from the usual bottom-up approach, in that there is no need to create an MDD for each constraint and then conjoin the MDDs. It suffices to create an MDD, say M, for the first constraint, refine it with respect to the second constraint by performing `SemanticRefine`(M, C_2), and so forth until all constraints are processed (Algorithm 4).

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 considering all constraints simultaneously, thus taking into account interactions among the constraints. We also provide approximation guarantees with respect to each constraint.

The basic idea is to regard two partial assignments 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 path equivalence in the same way as before. Distance measures are specialized to each type of constraint. For an inequality

Algorithm 5: `ApproxRefine`$(M, S, d^S_{\max}, T_{\max})$

Data: Starting MDD M, constraint set S, set $d^S_{\max}$ of distance thresholds, MDD size limit $T_{\max}$
Result: MDD Representation of Solution Space
while $S \neq \emptyset$ **do**
  foreach $C \in S$ **do**
    if M subsumes C **then**
      $S \leftarrow S \setminus \{C\}$;
    else if $M = \emptyset$ **then**
      return false;
    else
      $M \leftarrow$ `SemanticRefine`(M, C);
      if $|M| > T_{max}$ **then**
        return M
return M;

constraint $ax \leq b$, the distance could be

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

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

$$\text{distance}_{A}(p_1, p_2) = |D(p_1) \triangle D(p_2)|$$

where $S_1 \triangle S_2 = (S_1 \setminus S_2) \cup (S_2 \setminus S_1)$.

When equivalence is detected in this fashion, Algorithm 2 becomes an approximate MDD compiler. 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 $d^C_{\max}$. An approximate refining procedure appears as Algorithm 5. It refines the MDD with respect to each constraint. Then any constraints subsumed by the MDD are dropped, and the process is repeated. (An MDD *subsumes* a constraint when all paths in the MDD satisfy the constraint.) Computation is terminated early if a ceiling on the MDD size is reached.

The overall procedure begins with a trivial MDD (consisting of the single vertex *True*(0) and refines it using Algorithm 5. The process is then repeated with lower distance thresholds, obtained from the previous thresholds perhaps by a multiplicative or additive factor. Each refined MDD allows for more precise information to be collected in the next iteration which improves the accuracy of equivalence and infeasibility detection. In fact, it can be advantageous to compute an exact MDD through a sequence of approximations in which the distance thresholds are gradually reduced to zero.

Algorithm 6: `Overall Approximate Compilation`

Data: Constraint set S
$M \leftarrow True(0)$ **while** $|M| \leq T_{max}$ and M is inexact **do**
  $d^S_{\max} \leftarrow$ `getNewDistanceThresholds`(M, C);
  $M \leftarrow$ `ApproxRefine`$(M, S, d^A_{\max}, T_{max})$;
  $M \leftarrow$ `reduce`(M)

7 Experiments

In this section we will show how the presented techniques perform in practice for a selection of applications. All experiments using the techniques in this paper are done using our own MDD compiler and generic MDD-manipulation package, written in Java. Comparisons to standard compilation techniques makes use of CLab [9].

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.

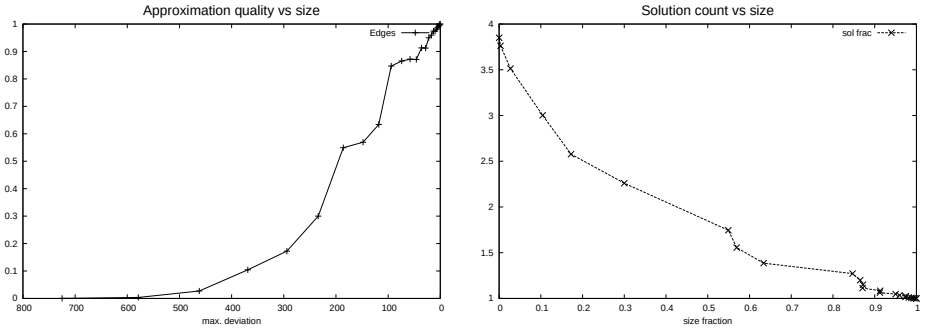

Fig. 3. The two plots above tracks the progress of the approximate compilation process of 5 random separable inequalities, with 15 variables over domains of size 3 and matrix elements from the range -100 to 100. The leftmost plot shows the approximative distance achieved on the horizontal-axis and the 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).

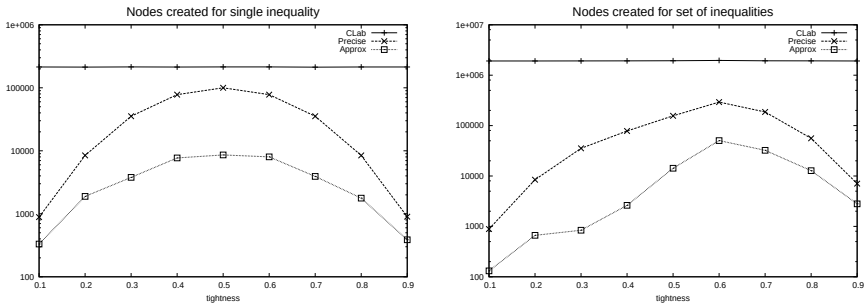

Fig. 4. (a) Total number of vertices created during compilation of a single random linear inequality over 18 variables with binary domains. The coefficients range from 0 to 100000. (b) Total number of vertices created during compilation of 10 random linear inequalities over 18 variables with binary domains.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

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.

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 [4] 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.

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 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 last scenario 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 [10]. 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

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

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

we report, for each number of constraints m, the median of these values over the 10 generated models.

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.

References

1. Vempaty, N.R.: Solving constraint satisfaction problems using finite state automata. In: Proceedings of the Tenth National Conference on Artificial Intelligence. (1992) 453–458
2. Hadzic, T., Andersen, H.R.: A BDD-based Polytime Algorithm for Cost-Bounded Interactive Configuration. In: Proceedings of AAAI'06. (2006)
3. Mateescu, R., Dechter, R.: Compiling constraint networks into AND/OR multi-valued decision diagrams (AOMDDs). In Benhamou, F., ed.: CP. Volume 4204 of Lecture Notes in Computer Science., Springer (2006) 329–343
4. Darwiche, A., Marquis, P.: A knowledge compilation map. Journal of Artificial Intelligence Research **17** (2002) 229–264
5. Hadzic, T., Hooker, J.N.: Cost-bounded binary decision diagrams for 0-1 programming. In Hentenryck, P.V., Wolsey, L.A., eds.: Proceedings of 4th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2007. (2007) 84–98
6. Andersen, H.R., Hadzic, T., Hooker, J.N., Tiedemman, P.: A Constraint Store Based on Multivalued Decision Diagrams. In Bessiere, C., ed.: Principles and Practice of Constraint Programming (CP 2007). Lecture Notes in Computer Science, Springer (2007)
7. Hadzic, T., Hooker, J.: Postoptimality analysis for integer programming using binary decision diagrams. Technical report, Carnegie Mellon University (2006) Presented at GICOLAG workshop (Global Optimization: Integrating Convexity, Optimization, Logic Programming, and Computational Algebraic Geometry), Vienna.
8. Huang, J., Darwiche, A.: DPLL with a trace: From SAT to knowledge compilation. In Kaelbling, L.P., Saffiotti, A., eds.: IJCAI, Professional Book Center (2005) 156–162
9. Jensen, R.M.: CLab: A C++ library for fast backtrack-free interactive product configuration. `http://www.itu.dk/people/rmj/clab/`(2007)
10. CLib: Configuration benchmarks library. Available online at: http://www.itu.dk/research/cla/externals/clib/ (2007)