Enhanced Inference for the Market Split Problem*

Tarik Hadzic
Cork Constraint Computation Centre
University College Cork
t.hadzic@4c.ucc.ie

Eoin O'Mahony
Cork Constraint Computation Centre
University College Cork
edom1@student.cs.ucc.ie

Barry O'Sullivan
Cork Constraint Computation Centre
University College Cork
b.osullivan@cs.ucc.ie

Meinolf Sellmann
Department of Computer Science
Brown University
sello@cs.brown.edu

Abstract

Inference in constraint programming is usually based on the deductions generated by individual constraints which are then communicated to other constraints through domain filtering. Frequently we find that this is a too coarse-grained form of communication since constraints could exchange more powerful forms of deductions that could help reduce the search effort. In this paper we propose a particular technique for enhancing inference in constraint programming, by generating deductions that involve tighter interleaving of constraints. We apply our method to the Market Split Problem and obtain massive speed-ups which brings a new order of Market Split Problems into the realm of solvability by means of constraint programming.

1 Introduction

Inference in constraint programming (CP) is usually based on the deductions generated by individual constraints which are then communicated through removal of values from variable domains. A constraint prunes a value from a domain which then invokes pruning in other constraints. This form of communication is very natural and has led to the design of efficient filtering algorithms and theoretical analysis of the inference power of various schemes. For example, in finite domain CP, variable domains are usually enumerated explicitly as part of the input. Then, the propagation is guaranteed to reach a fixedpoint in polynomial time whenever all constraints involved can be filtered in polynomial time.

There is a serious disadvantage, though. A removal of a variable assignment is an atomic amount of inference we can convey during search. This could be a too coarse form of communication in many domains. A constraint might be able to make powerful deductions from which other constraints might greatly benefit. However, such inferences are never communicated since they cannot be expressed as removals of a value in a domain. Conventional wisdom in the CP community is that generating and exchanging such coarser forms of inference is the most adequate approach: even though more powerful inferences would be made if individual constraints are more tightly interleaved, and even though this might result in a reduction in the size of the search tree, the execution time per search node would be much higher and hence the total search time would be longer.

The Market Split Problem, however, challenges this line of thought. As we demonstrate through our experiments, search trees traversed by the standard CP algorithms (enforcing arc or bounds consistency) are so huge that even models with 40 binary variables are out of reach. In this paper we therefore propose a new technique for enhancing inference between constraints. Our technique generates and compactly represents a number of deductions that involve reasoning about multiple constraints simultaneously. Such inferences are performed only once - before initiating the search. During search we use the stored inferences to reduce the search tree. We use *directed acyclic graphs* (in particular, we use multivalued decision diagrams) to represent constraints, and use *compatibility labels* to compute and store the inferences. We apply our method to the Market Split Problem and obtain massive gains in performance, which brings a new order of Market Split Problems into the

*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). Meinolf Sellman is supported by the National Science Foundation through the Career: Cornflower Project (award number 0644113).

realm of solvability by means of constraint programming.

The rest of the paper is organised as follows. In Section 2 we present related work. In Section 3 we present the technical background and notational conventions. In Section 4 we present the algorithmic core of our approach. In Section 5 we present an extensive experimental evaluation. We conclude in Section 6.

2 Related Work

There exist multiple techniques to enhance inference in constraint programming. For example, we could learn implied constraints. Constraint filtering can be viewed as a process in which an original constraint infers implied unary constraints. An enhanced reasoning process results from learning constraints that are not unary, and which are implied by multiple constraints. This approach is realized in mixed integer programming by adding valid inequalities, an idea which has boosted the power of mathematical programming considerably. A break-through of similar magnitude was achieved when SAT solvers began to learn non-unary clauses, so-called no-goods, which are added to the SAT formula.

Yet another way to enhance inference was introduced in [7] where individual subset-sum constraints were combined into a unique subset-sum, whose solutions were then represented as paths in a directed acyclic graph induced by a dynamic program. In this graph the removal of a variable assignment is triggered when all edges on a variable level, which are associated with the assignment value, have been removed. This principle was later also used in [6]. Pruning graph edges instead of values was also a key component of an approach to use a multi-valued decision diagram (MDD) rather than variable domains as a communication interface between constraints during propagation [2].

3 Background

In this section we provide the necessary background on preliminary concepts and notational conventions.

3.1 Constraint Satisfaction

A constraint satisfaction problem (CSP) is a triplet $\mathcal{P}(\mathcal{X}, \mathcal{D}, \mathcal{C})$ where $\mathcal{X}$ is a set of variables $\{x_1, \ldots, x_n\}$, $\mathcal{D}$ is a mapping of variables to sets of values and $\mathcal{C}$ is a set of constraints $\{C_1, \ldots, C_m\}$ that specify allowed combinations of values for subsets of variables. An assignment of a set of variables $\mathcal{X}$ is a set of pairs S such that $|\mathcal{X}| = |S|$ and for each $x \in \mathcal{X}$ there is a pair $(x, a) \in S$ such that $a \in \mathcal{D}(x)$. A constraint $C \in \mathcal{C}$ is *generalized arc consistent* (GAC) iff, when a variable in the scope of C is assigned any value, there exists an assignment of the other variables in C such that C is satisfied. This assignment is called a *domain support* for the value. Similarly, we call a *range support* an assignment satisfying C, but where values, instead of being taken from the domain of each variable ($a \in \mathcal{D}(x)$), can be any element between the minimum and maximum of this domain ($a \in [min(\mathcal{D}(x))..max(\mathcal{D}(x))]$). A constraint $C \in \mathcal{C}$ is *range consistent* (RC) iff, every value of every variable in the scope of C has a range support. A constraint $C \in \mathcal{C}$ is *bounds consistent* (BC) iff, for every variable x in the scope of C, $min(\mathcal{D}(x))$ and $max(\mathcal{D}(x))$ have a range support. The set of solutions Sol of a problem $\mathcal{P}$ is a subset of $\mathcal{D}(x_1) \times \ldots \times \mathcal{D}(x_n)$ such that every element in Sol satisfies all constraints $\mathcal{C}$.

3.2 Directed Acyclic Graphs

We use rooted *directed acyclic graphs* (DAGs) to represent solution spaces of constraints. Many well known knowledge representation forms can be viewed as DAGs, such as *multi-valued decision diagrams* (MDDs), acyclic automata, dynamic programming tables, etc. Throughout this paper we will use the term MDD since the implementation of our algorithms is based on MDDs. However, all the concepts and algorithms we present are equally applicable to general DAGs that might be used in other domains, such as dynamic programming tables. An MDD M is denoted as (V, E), where V is a set of vertices containing the special terminal vertex $\mathbf{1}$ and a root $r \in V$. Vertices are arranged in n layers $V_1, \ldots, V_n$, where each layer V_i is associated with variable x_i. Each edge $e \in E$ is denoted with a triple (u, u', a) of its parent node u, its child node u' and an associated value a. For each node we represent its child nodes and parent nodes using structures Ch and Pr, such that for edge (u, u', a) it holds $u' = Ch[u][a]$ and $u \in Pr[u'][a]$. Note that a vertex can have only one outgoing edge labeled with a specific label, while it can have multiple incoming edges with the same label.

We will denote with $p : u_1 \rightsquigarrow u_2$ any path in the MDD from u_1 to u_2. Also, edges between u and u' will be sometimes denoted as $e : u \rightarrow u'$. A value a of an edge $e(u, u', a)$ will be sometimes denoted as $v(e)$, while a partial assignment associated with path p will be denoted as $v(p)$. Every path corresponds to a unique assignment. Hence, the set of all solutions represented by the MDD is $Sol = \{v(p) \mid p : r \rightsquigarrow \mathbf{1}\}$. An example MDD is given on the left of Figure 1 which represents the solutions to the equation $2x_1 + x_2 + 2x_3 + 3x_4 + 4x_5 = 4$ for binary variables $x_1, \ldots, x_5$.

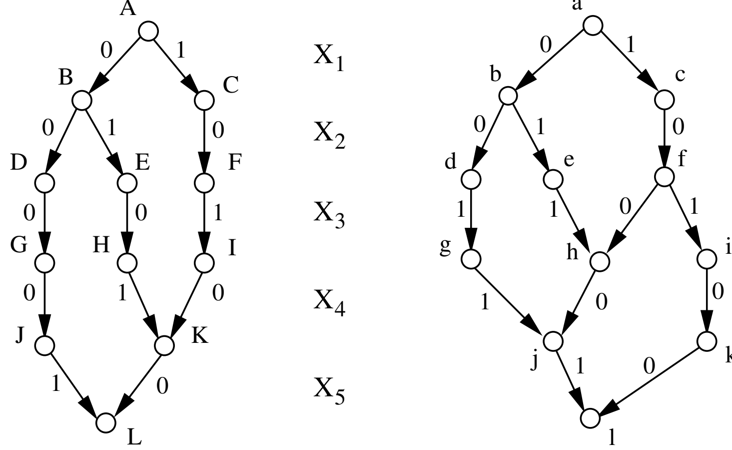

Figure 1. MDDs for Number Partitioning Constraints $2x_1 + x_2 + 2x_3 + 3x_4 + 4x_5 = 4$ **(left) and** $3x_1 + x_2 + 2x_3 + x_4 + 2x_5 = 5$ **(right).**

4 MDD-Based Inference

For a given problem specification $\mathcal{P}$, the first step of our scheme is to compile constraints $\{C_1, \ldots, C_m\}$ into MDDs $M_1, \ldots, M_m$, either exactly or approximately. Exact compilation is in general intractable, but for many important constraints, MDDs are surprisingly small. However, there are constraints, such as AllDiff that require MDDs that are exponential in size. In these cases, one can compile an over-approximation of such constraints [5]. This results in MDDs of tractable size that admit more solutions than the corresponding constraints. In an analogous way, one can generate a dynamic programming table that represent solutions to an equality [7]. In case that such table is prohibitively large, scaling and trimming techniques can be used to provide an over-approximation [6].

The resulting DAG structures can already be used during search to enhance propagation. If exact compilation is used, each DAG allows efficiently checking whether its corresponding constraint is GAC with respect to a current assignment. However, this still constitutes a standard inference scheme, where information exchanged between constraints are variable assignments.

4.1 Compatibility Labels

The critical aspect of our approach, however, is in the subsequent step. We initiate *exchange of inferences* between the various subsets of original constraints in a way that is not limited to the communication of domain values. As an extreme example, if we conjoin MDDs M and M', they would effectively exchange all the inferences they contain. This could result in dramatic increase of inference strength at the price of more memory consumption. The resulting conjunction might be infeasible even though all the values in all domains are supported. In this paper we suggest a specific technique for generating such inferences involving multiple MDDs. The technique depends critically on the notion of *vertex compatibility*.

Definition 1 (Compatibility)**.** *We are given an ordering of variables $x_1 < \cdots < x_n$ and a collection of k MDDs $M^1(V^1, E^1), \ldots, M^k(V^k, E^k)$. For a given variable x_i, let $V_i^1, \ldots, V_i^k$ denote layers in each MDD corresponding to x_i.*

- *We say that a combination of nodes $(u_1, \ldots, u_k) \in V_i^1 \times \ldots \times V_i^k$ is* prefix compatible *iff there exists an assignment to variables $x_1, \ldots, x_{i-1}$ such that the corresponding paths lead from the root to u_i in each MDD.*
- *Analogously, we say that a combination of nodes $(u_1, \ldots, u_k) \in V_i^1 \times \ldots \times V_i^k$ is* postfix compatible *iff there exists an assignment to variables $x_i, \ldots, x_n$ such that the corresponding paths lead from u_i to terminal* **1** *in each MDD.*
- *Finally, we say that a combination of nodes $(u_1, \ldots, u_k) \in V_i^1 \times \ldots \times V_i^k$ is* compatible *iff it is both pre- and postfix compatible.*

Example 1. *Consider MDDs for constraints $2x_1 + x_2 + 2x_3 + 3x_4 + 4x_5 = 4$ and $3x_1 + x_2 + 2x_3 + x_4 + 2x_5 = 5$ as shown in Figure 1. Compatibility labels for these two MDDs are given in Table 1. We see that the only nodes compatible with each other are those on the path which belongs to the only solution to both constraints,* $(1, 0, 1, 0, 0)$.

Note that in the previous definition, each MDD is assumed to contain a layer V_i for each variable x_i, even if

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**
pre	a	b	c	d	e	f	∅	∅	i	∅	k	l
post	a	c	c	f	∅	f	h	∅	i	j	k	l
comp	a	∅	c	∅	∅	f	∅	∅	i	∅	k	l
	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**
pre	A	B	C	D	E	F	∅	∅	I	∅	K	L
post	A	∅	B,C	∅	∅	C,F	∅	G	I	J	K	L
comp	A	∅	C	∅	∅	F	∅	∅	I	∅	K	L

Table 1. Compatibility Labels for the MDDs from Figure 1.

Algorithm 1: POSTFIX$(M_1, \ldots, M_k)$. Compute postfix-compatibility labels POST for combination of k MDDs.

```
   POST[i] = ∅, i = 1,...,n;
   POST[n+1] = {(1,...,1)};
1  foreach i = n+1,...,2 do
       foreach (u'_1,...,u'_k) ∈ POST[i] do
           foreach a ∈ D_{i-1} do
               foreach
               u_1 ∈ Pr[u'_1][a],...,u_k ∈ Pr[u'_k][a] do
                   POST[i-1] ←
                   POST[i-1] ∪ {(u_1,...,u_k)};
```

x_i is not in the scope of the corresponding constraint C. It is always possible to expand an MDD constructed over a subset of variables in the scope of C to include layers for each variable $\mathcal{X}$. This expansion incurs a worst-case linear increase in size. For every edge "skipping" a missing variable x_i, say an edge (u_1, u_2, a) with variable labels $var(u_1) = i-1$, $var(u_2) = i+1$, we insert one node in the i-th layer (say u_3). For each value a_j in domain $\mathcal{D}(x_i)$, we introduce an edge (u_3, u_2, a_j). In total we add one node and $|\mathcal{D}(x_i)|$ edges for each edge "skipping" V_i. Some of the newly created nodes are isomorphic and can be eliminated in the subsequent merging phase.

4.1.1 Computing Compatibility Labels

Algorithm 1 illustrates how to compute postfix-compatibility labels for a collection of MDDs $M_1, \ldots, M_k$. Initially, the combination of all terminal nodes is designated as postfix-compatible. The algorithm then traverses the MDD in a bottom-up manner, and for every compatible k-tuple of nodes in layer i, updates the set of compatible tuples in layer $i-1$.

The worst-case space and time complexity of our scheme is $O(\sum_{i=1}^{n} |V_i^1| \cdot \ldots \cdot |V_i^k|)$, where $|V_i^j|$ is the number of nodes at the i-th layer of the j-th MDD. The complexity grows exponentially with k, and could exceed available resources even for moderately small k or when the MDDs are large. There is a number of ways in which we might trade-off the strength of inference for reduced time and space requirements. One of the simplest ones that was particularly effective in our empirical evaluation, is to limit the number of layers for which we compute postfix labels. For a given layer threshold L, we iterate over layers $n+1$ to L in the line with label 1 of the algorithm.

Enhancing Search. Compatibility labels can be exploited to enhance inference in many different ways. In this paper we are using them during MDD-based search for a solution. For a given size of collection k, we compute for each layer i and each k-combination of MDDs $\{M_{j_1}, \ldots, M_{j_k}\}$ the set of postfix compatible nodes $POST[j_1, \ldots, j_k][i]$. After computing postfix labels for all k-combinations of initial MDDs, we initiate a DFS-based tree search algorithm MDDSAT, shown in Algorithm 2. We branch with respect to ordering $x_1, \ldots, x_n$, and at each choice point we reach an m-tuple of MDDs $(u_1, \ldots, u_m)$. If for any k-combination, $(j_1, \ldots, j_k)$, nodes $(u_{j_1}, \ldots, u_{j_k})$ are not postfix compatible, we backtrack in the line with label 1. This is the primary mechanism we use to exploit the labels. The algorithm is instantiated by calling MDDSAT$(r_1, \ldots, r_m, x, 0)$, where x is an empty set of assignments, and $(r_1, \ldots, r_m)$ are root nodes of corresponding MDDs $M_1, \ldots, M_m$. Note that we could compute postfix labels in more flexible ways, e.g. choosing only a few subsets of constraints for which label computations might be particularly useful. For the purpose of this paper however, it suffices to generate labels for all combinations of a specified length k.

4.1.2 Inference Guarantees

Algorithm 2 is a simple DFS exploration of the search space that uses MDDs and compatibility labels to backtrack. If no postfix labels are used, the algorithm backtracks after assigning x_i if each value in a domain of x_{i+1} lacks domain support in at least one constraint.

If postfix labels are used, then for a partial assignment p if the algorithm backtracks with respect to postfix labels

Algorithm 2: `MDDSAT` $(u_1, \ldots, u_m, x, var)$

```
if var = n + 1 then
    return true;
foreach {u_{j_1}, ..., u_{j_k}} ⊆ {u_1, ..., u_m} do
    if (u_{j_1}, ..., u_{j_k}) ∉ POST[j_1, ..., j_k][var] then
1       return false;
foreach a ∈ D_var do
    foreach i = 1, ..., m do
        if Ch[u_i][a] = null then
            try next value a;
    foreach i = 1, ..., m do
        u'_i = Ch[u_i][a];
    x[var] = a;
    if MDDSAT(u'_1, ..., u'_m, x, var + 1) = true then
        return true;
return false;
```

$(j_1, \ldots, j_k)$ then p has lost support in the conjunction of constraints $M_{j_1} \wedge \ldots \wedge M_{j_k}$. Hence, executing `MDDSAT` for MDD combinations of size k is equivalent to executing classical DFS search with respect to *conjunctions of MDDs* $\{M_{j_1} \wedge \ldots \wedge M_{j_k} \mid \{j_1, \ldots, j_k\} \subseteq \{1, \ldots, m\}\}$. Hence, we could view compatibility labels as implicit conjunctions of k-tuples of MDDs. While in the worst case we might expect the same space complexity, it is reasonable to expect that the labels require less space than an explicit MDD conjunction (which requires instantiating a number of auxiliary structures - at the very least one has to maintain edge connectivity information which is not present in compatibility labels). In our experience with binary labels ($k = 2$) constructing conjunctions of MDDs took significantly longer than building postfix labels.

Compatibility-Based Filtering. Another important way in which we can use compatibility labels is to *prune MDDs*. If a node in an MDD is not part of at least one compatible tuple in a combination of MDDs, it can be deleted. Namely, in that case there exists no solution to all constraints that passes through the node. This provides us with a filtering algorithm for MDDs (or other DAGs stemming from dynamic programs or approximation schemes). Nodes which are not part of at least one k-combination are removed. This may leave some nodes in other MDDs incompatible which can now be removed as well. In this way, we propagate information between constraints represented as MDDs. In a similar way, one can remove edges if they cannot be used to connect compatible tuples of nodes.

Computing compatibility labels and filtering can be performed *statically*, i.e. only at the root node of the search tree, much as the valid inequalities generated in mixed integer optimization. However, these inference steps can be also performed *dynamically* at each choice point, such as learning nogoods in SAT solving.

5 Experiments

We evaluated the performance of our algorithms on the decision version of the *Market Split Problem* (MSP) [3]. An MSP instance consists of m linear equalities over $n = (m-1) \cdot 10$ binary variables.

$$a_1^j x_1 + \ldots + a_n^j x_n = b^j, \; j = 1, \ldots, m.$$

For each linear equality coefficients are drawn randomly from the interval $[0, 99]$ and the right-hand side of the equation b^j is set to $\frac{1}{2} \cdot \sum_{i=1}^{n} a_i^j$. In the rest of the paper, we will use (m, n) to denote instances with m constraints and n variables.

The MSP problem was originally suggested in the operations research (OR) community to challenge standard solving algorithms based on exhaustive search. It proved to be an extremely hard to solve problem for standard integer programming methods even for the $(4, 30)$ instances. Only specialized approaches managed to solve larger instances by utilizing a basis reduction technique to transform the original problem over binary variables into a problem over unbounded integer variables [1]. However, the original problem formulation remains extremely challenging for exhaustive search algorithms. In the CP community MSP instances were used in [7] and [6] to evaluate propagation algorithms for knapsack constraints, managing to handle $(4, 30)$ and $(5, 40)$ instances respectively. Interestingly, both approaches involved the generation of a dynamic program (a DAG) for a linear combination of equality constraints before starting the search algorithm, and then maintaining the DAG during search to enhance propagation. The creation of a dynamic program could be seen as a preprocessing step in which inferences are generated and stored involving information from all constraints in the model.

To better understand and compare the performance of our approach, we first tried to solve MSP instances using standard state-of-the-art *constraint programming* techniques. We then evaluated a number of variations of our approach based on the MDD representation of constraints. For our evaluation we used publicly available instances from MIPLIB[1]. All experiments were executed on a Fedora 9 operating system, using dual Quad core Intel Xeon processor running at 2.66 GHz.

5.1 Standard CP Approaches

To evaluate standard CP approaches we used the CP system Choco[2] with two propagation algorithms: enforcing

[1] `http://miplib.zib.de/contrib/Markshare/`
[2] `http://choco-solver.net`

(generalized) arc consistency (GAC) and *bounds consistency* (BC) during search. All the tests involving standard CP approaches were run with a time limit of ten hours and twelve gigabytes of memory.

5.1.1 Generalized Arc Consistency

To enforce *generalized arc consistency* for the equality constraint (in CP vocabulary this is often referred to as a *sum constraint*), after each assignment the solver prunes all values from domains of unassigned variables that do not have domain support with respect to at least one constraint. To enforce this level of consistency in Choco, we use the *regular* constraint, which builds an automaton (a DAG equivalent to our MDD) encoding solutions for each equality. These automata are then updated during search to maintain GAC.

However, the memory requirements were too large to build automata even for the $(4, 30)$ instances. We therefore broke equalities into sub-expressions, and built automata for each sub-expression, linking them through shared variables corresponding to sums of corresponding subexpressions. The memory requirements were much smaller, and we were able to solve all $(4, 30)$ instances in on average 17.2 seconds and 391 choice points. However, we could not handle any of the $(5, 40)$ instances.

5.1.2 Bounds Consistency

Enforcing arc consistency for an equality constraint is NP-hard as it involves solving a subset-sum problem [4]. It also leads to large memory requirements as we have observed in previous experiment. A very popular alternative is to enforce *bounds consistency* (BC) instead. Enforcing BC for an equality $a_1x_1 + \ldots + a_nx_n = b$ is equivalent to removing values from unassigned domains if they violate one of the inequalities:

$$a_1x_1 + \ldots + a_nx_n \leq b,\ a_1x_1 + \ldots + a_nx_n \geq b.$$

This is a strictly weaker form of consistency, as many values having support in each inequality might not have support in the equality. However, it is usually a very effective propagation algorithm, as it is fast to propagate at each choice point. We discovered that this form of consistency uses huge number of choice points. Choco, enforcing bounds consistency with impact-based search, solves $(4, 30)$ instances in about 30 seconds, using on average approximately $3 \cdot 10^6$ choice points. It is able to solve four out of five $(5, 40)$ instances, taking on average 140 minutes and $0.7 \cdot 10^9$ choice points.

5.2 MDD-Based Search

Our MDD-based approaches were evaluated over two sets of $(5, 40)$ and $(6, 50)$ instances available at MIPLIB. As the first step, for each equality we generate an MDD using lexicographic variable ordering, and utilizing the BDD package BuDDy.[3] We present the MDD properties of these instances in Table 2. For each instance we indicate its feasibility, the total time required to generate the MDDs for individual constraints and the average number of nodes and edges in such MDDs. Note that we were able to generate MDDs for all the instances while Choco could not generate automata even for the $(4, 30)$ instances. This is surprising given that the MDDs we use and automata generated by Choco are equivalent DAGs. The difference might be due to the fact that the BDD package BuDDy is highly optimized and well engineered software tailored for efficient generation and manipulation of decision diagrams. In comparison, automata building in Choco is just an auxiliary procedure designed to support propagation of one of the many supported constraints.

Table 2. MDD properties of Market Split Problem instances.

	Feasible	Compile(s)	Nodes	Edges
5_0	no	3.91	15,604	29,205
5_1	no	4.27	15,218	28,504
5_2	no	3.82	14,588	27,300
5_3	yes	3.67	14,383	26,908
5_4	no	3.64	14,186	26,615
6_0	yes	10.64	26,234	49,956
6_1	no	10.68	25,659	48,843
6_2	yes	10.22	25,727	48,977
6_3	yes	10.03	25,706	48,971
6_4	no	10.24	25,608	48,867

5.2.1 Basic MDD Search

In order to better understand the effect of introducing compatibility labels, we first tested the basic version of our `MDDSAT` algorithm (Algorithm 2), which does not utilize the labels during search. Unlike the standard CP approaches, all the variables are assigned in the fixed order that respects the variable ordering in the MDD. The less flexible variable ordering may lead to a worst-case larger number of choice points, but requires much less execution time at each point. When assigning variable x_i, the basic `MDDSAT` algorithm considers only values from $\mathcal{D}(x_i)$ that have domain support with respect to each constraint; such values are given implicitly by labels on outgoing edges of the current nodes.

[3] http://sourceforge.net/projects/buddy

Table 3. Performance of Algorithm 2 over Market Split Problem instances. All times are given in (m:s) format.

	$k=0$		$k=2$			$k \in \{2,m\}$				
	Iterations	**Search**	**Iterations**	**Label**	**Search**	**Iterations**	**Label**	**L**	**Search**	**Compile**
5_0	$1.2 \cdot 10^9$	6:51	$6.5 \cdot 10^6$	0:13	0:04	$1.9 \cdot 10^6$	0:07	21	0:03	0:22
5_1	$1.5 \cdot 10^9$	6:52	$7.4 \cdot 10^6$	0:12	0:05	$1.9 \cdot 10^6$	0:07	21	0:03	0:27
5_2	$1.2 \cdot 10^9$	6:50	$5.8 \cdot 10^6$	0:11	0:04	$1.6 \cdot 10^6$	0:08	21	0:03	0:20
5_3	$0.26 \cdot 10^9$	1:26	$1.5 \cdot 10^6$	0:11	0:01	$0.4 \cdot 10^6$	0:08	21	0:01	0:22
5_4	$1.2 \cdot 10^9$	6:56	$7.6 \cdot 10^6$	0:10	0:05	$1.9 \cdot 10^6$	0:07	21	0:03	0:18
6_0	-	-	$2.5 \cdot 10^9$	0:47	42:00	$9.1 \cdot 10^7$	3:02	26	5:23	126:27
6_1	-	-	$3.0 \cdot 10^9$	0:45	49:00	$1.2 \cdot 10^8$	3:01	26	6:54	109:40
6_2	-	-	$1.9 \cdot 10^9$	0:45	31:00	$6.8 \cdot 10^7$	3:02	26	3:45	116:24
6_3	-	-	$2.3 \cdot 10^9$	0:45	39:00	$8.7 \cdot 10^7$	2:55	26	4:38	134:35
6_4	-	-	$2.9 \cdot 10^9$	0:45	47:00	$1.2 \cdot 10^8$	3:02	26	5:57	130:20

The results are shown in Table 3 under the $k = 0$ heading. We were able to solve $(5, 40)$ instances within seven minutes and 10^9 iterations on average. Note that instance 5_3 stands out, requiring about four times less iterations. The smaller number of iterations can be expected since this is the only satisfiable instance. The search did not terminate for $(6, 50)$ instances even after 24 hours of computation.

Note that with respect to the number of choice points, performance of the basic `MDDSAT` scheme is comparable to bounds consistency enforced by Choco. Both approaches require approximately 10^9 iterations. However, the processing of each choice point is much faster in our implementation, so our scheme takes on average 20 times less time (7min in comparison to 140min). Despite the similar number of iterations it is hard to draw reliable comparisons between the two schemes. On one side, basic `MDDSAT` achieves a stronger consistency level in the domain of each variable $\mathcal{D}(x_i)$ that is about to be assigned. In contrast, bounds consistency achieves a weaker consistency level but is enforced over all unassigned domains, rather than a single domain. Furthermore, `MDDSAT` branches with respect to a fixed variable ordering, while Choco is using a sophisticated impact-based variable ordering heuristic.

5.2.2 MDD Search With Compatibility Labels

In our second set of experiments we tested two versions of Algorithm 2 involving compatibility labels. The first version uses postfix labels of arity $k = 2$, while the second version also uses labels for the entire collection of constraints (i.e. using 2-ary and m-ary labels where $m = 5$ and $m = 6$ for $(5, 40)$ and $(6, 50)$ instances respectively). For each version we indicate the number of choice points, the time required to compute the labels and terminate search. The results are shown in Table 3 under $k = 2$ and $k \in \{2, m\}$ headings respectively. Note that the labeling time for the second version refers *only* to the generation of m-ary labels. Hence, the total labeling time for the second version of the algorithm is the sum of labeling times for both versions. Note that in order to get the total running times, one should also add the time necessary to generate the MDDs from Table 2.

The first version ($k = 2$) solves all the $(5, 40)$ instances within 20 seconds on average and using approximately $6 \cdot 10^6$ iterations. Around 4 seconds are spent on MDD generation, 12 seconds on computing postfix labels and 4 seconds on search. This a 20 fold speedup in comparison to the basic scheme (taking 7 minutes) and a 200 fold decrease in the number of choice points. Obviously, introducing 2-ary compatibility labels results in a dramatic reduction in search effort which offsets the time required to compute the labels. Furthermore, for the first time we can solve $(6, 50)$ instances, which require between 31 and 49 minutes of search time, taking more than $2 \cdot 10^9$ iterations on average. To the best of our knowledge, this is the first constraint programming approach capable of solving $(6, 50)$ instances.

The second version of the algorithm performs even better for the $(6, 50)$ instances. It significantly reduces the search time to between 4 and 7 minutes. The number of iterations is cut down to around 10^8 which is approximately 20 times less than in the first version. In total, the second version solves the $(6, 50)$ instances in less than 9 minutes.

Tradeoff Between Inference and Search. In Table 3 in the second version of the algorithm ($k \in \{2, m\}$) we indicate the level L up to which we are computing m-ary labels. As we mentioned previously, L denotes the smallest level until which the compatibility labels are computed. During search, the `MDDSAT` algorithm would be able to backtrack based on the m-ary variables only after assigning the first

$L-1$ variables. However, the smaller the L the more labels are generated, which consumes both time and memory. We discovered however that at some point, the computational effort invested in labeling would offset the speedups obtained during search. We found that setting $L = 21$ and $L = 26$ provides the best ratio between inference and search for $(5, 40)$ and $(6, 50)$ instances respectively, yielding the smallest total running times. Setting L to smaller values increases the label-computation times significantly more than decreasing the search time. In addition, the memory requirements increase significantly as well. For $L = 24$ in $(6, 50)$ instances we run out of memory.

Dynamic Propagation. Dynamic versions of our algorithms, involving recomputation of compatibility labels at each choice point are so far unable to produce competitive time results. While the number of iterations is reduced by several orders of magnitude in comparison to the static version (which computes labels only at the root node), the time spent per iteration is significantly longer. For example, the number of iterations in instance 5_0 is reduced from 6.5 million to only 200, but the solving time increases to 2m 30s.

5.2.3 Compilation Approach

As an extreme form of generating inferences in a preprocessing step, we can consider compiling the conjunction of all constraints, i.e. generating an MDD for $M_1 \wedge \ldots \wedge M_m$. The resulting MDD would represent the set of all solutions to the system of equalities and trivially provide an answer to the satisfiability query. For this purpose we, again, utilize the efficient BDD package BuDDy, conjoining the BDDs corresponding to each equality using the standard conjunction operation provided by the package. The results are surprisingly good. We are able to compile MDDs for $(5, 40)$ instances within 22 seconds on average. We are even able to compile $(6, 50)$ instances within 2 hours on average, using about 1.5 GB memory. The running times for the compilation are shown in the rightmost column of Table 3.

6 Conclusions

We presented a set of techniques for enhancing inference between constraints during search. We evaluated the performance of our techniques for the Market Split Problem and demonstrated significant improvements in speed obtained. The Market Split instances with 6 constraints and 50 variables are for the first time solvable within the constraint programming framework. In the future we plan to further explore various forms of inference techniques based on compatibility-labels.

Acknowledgments

We would like to thank Radoslaw Szymanek for useful discussions in the early stages of this work.

References

[1] K. Aardal, R. E. Bixby, C. A. J. Hurkens, A. K. Lenstra, and J. W. Smeltink. Market Split and Basis Reduction: Towards a Solution of the Cornuejols-Dawande Instances. *INFORMS Journal on Computing*, 12(3):192–202, 2000.

[2] H. R. Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemman. A Constraint Store Based on Multivalued Decision Diagrams. In C. Bessiere, editor, *Principles and Practice of Constraint Programming (CP 2007)*, Lecture Notes in Computer Science. Springer, 2007.

[3] G. Cornuéjols and M. Dawande. A class of hard small 0-1 programs. In *Proceedings of the 6th International IPCO Conference on Integer Programming and Combinatorial Optimization*, pages 284–293, London, UK, 1998. Springer-Verlag.

[4] M. R. Garey and D. S. Johnson. *Computers and Intractability-A Guide to the Theory of NP-Completeness*. W H Freeman & Co, 1979.

[5] T. Hadzic, J. N. Hooker, B. O'Sullivan, and P. Tiedemann. Approximate compilation of constraints into multivalued decision diagrams. In P. Stuckey, editor, *Proceedings of CP 2008*, volume 5202, pages 448–462. Springer-Verlag, 2008.

[6] M. Sellmann. The Practice of Approximated Consistency for Knapsack Constraints. In D. L. McGuinness and G. Ferguson, editors, *AAAI*, pages 179–184. AAAI Press / The MIT Press, 2004.

[7] M. A. Trick. A dynamic programming approach for consistency and propagation for knapsack constraints. *Annals of Operations Research*, 118(1):73 – 84, 02 2003.