Using MDDs to solve problems via their decompositions

Joachim Schimpf
Caulfield School of Information Technology
Monash University, Australia
joachim.schimpf@infotech.monash.edu.au

December 2009

Abstract

Multivalued Decision Diagrams (MDDs) are directed acyclic graphs that can be used to represent relations in a compact way. In finite-domain Constraint Programming, constraints can be represented in extension, i.e. by their truth tables, which can be encoded as MDDs. Recently, an algorithm has been proposed for efficiently computing Generalised Arc Consistency on the basis of such an MDD representation. In this report we investigate several ideas for how to exploit these new tools for solving hard constraint satisfaction and optimization problems. Our general approach consists in working on a number of subproblems separately, representing the results as MDDs, and using these MDDs to gain efficiency in the solution of the problem as a whole.

Contents

List of Figures

1 Motivation

A lot of research and development effort in Constraint Programming has gone into developing efficient solvers for certain classes of primitive constraints on one hand, and efficient specialised propagation algorithms for particular global constraints on the other.

But there has also been slow but steady progress in the area of propagation algorithms for general constraint relations. This was originally motivated by table constraints, but more recently, since the arrival of the Constraint Catalog with its over 200 entries, by a desire for more generic approaches.

As these generic propagation algorithms get understood better and become more efficient, the question arises: instead of modelling a problem in terms of the available, efficiently implemented global and primitive constraints, is it realistic or helpful to consider problem-specific constraints, and use generic propagation algorithm on those? In such a setting, the notion of a 'constraint' has two aspects:

1. a predicate in the problem model, used purely for its declarative semantics
2. a unit of reasoning (including, but not limited to, propagation)

This is of course related to the problem of translating a conceptual model to a design model. But this mapping has traditionally focused on mapping a description which is close to a problem, to a description in terms of constraints that best fit a particular solver. And, given the level of sophistication attained by solvers for restricted problem classes (LP, SAT, global FD constraints), it is unlikely that this will become unimportant.

Nevertheless, there may be a role for extracting subproblems from a conceptual model, and treating these subproblems as constraints in the second (design model) sense, by applying generic methods to propagate, locally search and/or optimise over them. In this scenario, the process of deriving a design model from a conceptual model can be expected to focus more on problem features like topology of the constraint graph, rather than having to be concerned with pleasing a particular solver.

At the very least, having a toolbox of methods that apply to generic constraints, is expected to help with research into principles of model transformation, without being biased too much towards models that suit particular solvers.

2 Consistency and related Properties

Let us first review some basic notions of consistency in the context of constraint satisfaction problems. We assume the usual setting where we have *variables* with *finite domains*, and *constraints* between those variables. These form a *constraint network* in whose *satisfiability* we are interested. Variables can be assigned values from their domain. For a constraint, an *assignment* of values to its variable is called a *valid tuple* if it satisfies the constraint. An assignment Y=b is a *support* of an assignment X=a if they occur together in a valid tuple.

Arc Consistency (AC) (for binary constraints): every value in the domain of one variable is supported by a value in the other.

Generalised Arc Consistency (GAC) (for n-ary constraints): every value in the domain of one variable is supported by a valid tuple.

Beyond that, we have more powerful notions, although they are not representable in terms of domains alone:

Path Consistency (PC) (for binary constraints): every valid pair of assignments of two variables can be extended to a valid triplet.

k-Consistency (for nary constraints): every valid k-1 tuple can be extended to a valid k-tuple. (Strong k-consistency is j-consistency for all $j \leq k$).

Some implications of the binary notions (AC/PC)

- AC implies satisfiability for cycle-free constraint networks. Moreover, given the arc-consistent network, actual solutions can be found by the following algorithm: pick an arbitrary variable and an arbitrary value in its domain; pick a neighbour of an already assigned variable, and pick a value from its domain that is compatible with the already assigned neighbour (this exists, otherwise the connecting constraint would not have been arc-consistent); continue until everything is assigned. In practice, we usually establish AC again after every assignment: the constraint network then falls apart into arc-consistent, cycle-free sub-networks, which can then be treated recursively, i.e. the next variable/value pair to be assigned can again be arbitrarily chosen.
- PC implies satisfiability for arbitrary networks with binary constraints and binary domains (does this have any practical application?)

Properties of GAC n-ary constraints:

- cycle-free networks (where constraints only overlap in one variable) of GAC constraints are GAC as a whole.
- However, there are probably no interesting cycle-free networks (two constraints sharing two variables already form a cycle), although they can occur after some variables are already instantiated.
- The notion of cycle-freeness can still be useful when looking at the macro-structure, where we may have (cyclic) subproblems, that are loosely connected (via a small number of shared variables) to other cyclic subproblems. If this macro-structure is cycle-free, then all information from one sub-problem can be transferred via the domain of a single connecting variable (or the allowed tuples in the case of several connecting variables) to the other subproblem.

If we regard the whole problem as a single n-ary constraint, and are able to establish GAC on this constraint, then this

- implies that there exists a solution (actually at least as many solutions as the size of the largest domain, and at most the product of the domain sizes)

- but it may not contribute much towards finding an actual solution: e.g. in the n-queens problem, all domain values are typically possible for all variables, so finding an actual solution after achieving GAC is just as hard as before. Thus the effort for computing GAC can be easily wasted - this is the motivation for the idea of approximations in Generalised Propagation (GP, section 7.1). GAC is only useful when the (remaining) problem is already quite tight, and failure or domain reduction can be inferred. This idea is used in GP's consistent-approximation, which stops after the existence of a single solution has been established.

3 Arc Consistency Algorithms (AC/GAC)

We review arc consistency algorithms here because they are relevant on two levels: One one hand, we ideally want to achieve arc-consistency within the subproblems we consider, and on the other, we might like to create a similar form of consistency among the subproblems.

3.1 AC on binary constraints

Arc consistency algorithms can be classified into coarse-grained and fine-grained ones. In order to re-establish consistency, the fine-grained ones require precise information about which values were removed from which domain, while the coarse-grained ones only need to know which variable had its domain reduced. An overview of binary arc consistency algorithms can be found in [Bes06].

Practical arc consistency algorithms for binary constraints are the coarse-grained AC-3 [Mac77] and AC-3.1 (aka AC-2001) [BRYZ05], and the fine-grained AC-6 [BC93] and AC-7 [BFR95].

AC-3.1 differs from AC-3 in that it maintains a table of supports, which enables it to achieve the same (optimal) worst-case complexity as the fine-grained algorithms. Another refinements is to exploit support symmetry by heuristically trying trying to first find tuples that support two hitherto unsupported values [vD02], and thus reduce checks.

In the fine-grained group, AC-6 can be seen as a lazy variant of AC-4 (which is known to be impractical due to memory requirements and the fact that its average complexity is the same as its worst case complexity), and AC-7 improves on that by exploiting constraint symmetry.

3.2 GAC via binary AC

Arc consistency on binary constraints has been extensively studied. Its particular importance is based on the observation that n-ary relations can be encoded as binary relations [SS05].

Dual encoding: replace the original problem, containing n-ary constraints, by its dual. This has one variable per original n-ary constraint, ranging over the valid tuples. In addition, one binary constraint for each case where an original variable was shared between two original constraints. These dual constraints make sure that only such (original) tuples can be combined that have the same value for the connecting original variable.

Hidden variable encoding: keep the original variables, and add dual variables for each original constraint. A dual variable's value is an index for a tuple that satisfies the constraint. Set up (element) constraints that ensure that values of the original variables are compatible with those tuple indices (dual variable values) where the tuple has that value.

3.3 GAC on n-ary constraints

Algorithms for Generalised Arc Consistency, and candidates for comparison, are

- Map to a cycle-free binary network, and use binary AC algorithms. [SS05] proposes specialisations of generic AC algorithms adapted to deal with the special structure resulting from binary encodings of n-ary constraints.
- GAC3 - systematically look for supports for every value of every variable. There are variants for how exactly those supports are sought.
- GAC-scheme-allowed - a fine-grained algorithm with complex indexing of the constraint table and support maintenance. Variants exist with respect to the index data structure. Moreover, there are variants for constraints defined by negative tables, or via predicates.
- GAC-scheme on (test-only) predicates. This is also applicable to constraints defined by tables of forbidden tuples (using hashing).
- MDD based algorithm - coarse-grained, no support maintenance, but shortcuts based on mdd structure.
- Topological-Branch-and-Bound (TBB, 3.4.3) as used in GP (section 7.1), mainly aimed at constraints defined via a generator program.

Most work on table constraints is on tables of allowed tuples. The advantage here is that

- we can expend effort on building static index structures a priori, ideally offline.
- these index structures can typically be shared between multiple instances of the constraint.

3.3.1 GAC-Schema and Variants for table constraints

[BR97] presents an algorithm to achieve GAC on arbitrary constraints. It is loosely based on the ideas of the binary algorithm of AC-7, but much more involved. It is called a schema because there are different variants depending on whether the constraint is defined by (i) an *allowed*-table of all solution tuples (ii) a *forbidden*-table of non-solution tuples (iii) a predicate for which no semantics is known, but the availability of an effective *test* is assumed.

For the prevailing case of allowed-tables, the data structures used are: a table last(x,a) giving the last tuple scanned for a particular variable and argument value; and a table next(x,a,t) giving the next tuple with x=a after tuple t (we'll call this the NextSame data structure). The algorithm iterates over allowed tuples, ignoring the invalid ones (i.e. invalid with respect to the domains).

[LR05] is looking at table constraints only, and improves the indexing of these tables with the result of making GAC-schema more efficient and eliminate pathological cases of exponential (in the constraint arity) complexity. The key is in additional indexing of the allowed-table structure This is an index pointing directly to the next tuple with a particular value at a particular argument position. However, this is significantly bigger (by a factor of the domain size) than the tuple table it indexes.

[Lho04] invents a clever data structure called *hologram* tuples to reduce the Bessiere/Regin tables by a factor of domain size. The idea is essentially to "fold" or flatten the extra table dimension into the existing dimension, losing some precision, but only slowing down the related operations by a constant factor of no more than the domain size. This weaker index can still achieve the exponential savings of the original one.

[GJMN07] look at further alternative data structures for use with GAC-Schema on table constraint, namely NextDifference lists, and Tries. NextDifference lists add another index the size of the NextSame structure. They record for every tuple t with x=a the next tuple that has a different value for x. The memory requirement is between the simple and the hologram structures. The authors exploit this structure in an unnecessarily naive way only, and thus find it not competitive. Their experiments with tries are promising, they are found to be similar or better than holograms.

Their trie algorithm uses one trie for every variable in the constraint's scope, because of the variable ordering problem. Each of these tries starts with a different variable. This makes it cheap to establish support for every variable and value in the constraint's scope. A supporting tuple is a path from root to leaf, and the current supports are remembered as the corresponding leaves. When support is lost, search is resumed from that leaf (which requires parent-pointers in the trie).

3.4 GAC on non-table constraints

As we intend to compute GAC on non-table constraints, i.e. constraints defined by a test predicate or a subproblem, the following aspects become relevant:

- Indexes, if any, must be built dynamically.
- We may have only a test procedure. In that case, we can build an enumeration procedure over the domain, but without any further knowledge about the constraint, it is just generate-and-test and unlikely to be efficient.
- We may have an efficient enumeration procedure for all tuples.
- We may have an enumeration procedure that can take partial information (domains) into account and find the next valid tuple under these domain restrictions efficiently. (For instance, a compiled Prolog fact base in ECLiPSe is indexed on any one argument, so it can enumerate efficiently all possible supports for the value an argument is instantiated to).
- We may have less-than-GAC propagation on the subproblem. Two scenarios: (i) We know the structure, e.g. logical combination of GAC components. TBB (3.4.3) can efficiently deal with a disjunction of GAC components (an extension of enumerating solutions), but a dedicated handling of

the logical combinators may be more efficient (see 7.2). (ii) We don't know the structure: in that case, we can take advantage of the reduced domains from that weak propagation, and are otherwise in the same situation as without it.

- The terminology of table-oriented procedures is not so useful: *valid* for tuples within the current domains, *allowed* for tuples satisfying the constraint. We are rather in a situation where the subproblem is further constrained by the current domains.
- We may want to know some characteristics of the subproblem solver: is ground test cheap? Is partial instantiation solving cheaper than full solving, i.e. should we look for support for one value at a time, or generate at random?
- We have to be careful with solving subproblems with different domain restrictions: they may generate solutions in different orders, which may be a problem with some of the GAC-algorithm's ordering assumptions.

3.4.1 GAC-Schema on predicates

[BR97] looks also at the case where the constraint is defined not by a table but by a predicate.

This requires only a test for the constraint (on a fully instantiated tuple). For every variable and value, support is sought. When looking for support, all valid tuples (that have the value of interest) get enumerated in lexicographic order. A quite involved algorithm is used to make sure that the same tuple is never checked twice. This is tricky because the tuples are not enumerated once lexicographically, but separately for every single var/value for which support is sought.

3.4.2 GAC-Schema on subproblems

In [BR99], Bessiere and Regin look at applying GAC-Schema to subproblems that themselves are stated in terms of (global) constraints. They expressly advertise this as a tools for prototyping specialised propagators for combinations of constraints, with the rationale that prototyping can show whether a substantial reduction of search space can be achieved, and thus justify the effort of developing a dedicated, efficient propagator algorithm.

Alternatives for special cases are (i) precompute all allowed tuples, and use GAC-Schema+allowed, (ii) precompute all forbidden tuples, and use GAC-Schema+forbidden. However, in general these are not realistic options, and the GAC-Schema+predicate is too naive since it uses the defining predicate only to test for allowed tuples.

3.4.3 Topological Branch-and-Bound

Topological Branch-and-Bound (TBB) is an algorithm originally developed for Generalised Propagation, which can also be used to compute GAC. It proceeds by constructing a *most general solution* to a subproblem, using a branch-and-bound technique.

- Only requires solution enumeration capability.
- Does not require the enumerator to deliver fully instantiated tuples. Will generalise whatever partial domain restrictions the subproblem solver finds (if they are not GAC, the result will not be GAC).
- Successively finds solution tuples that differ in at least one variable assignment. This isn't very efficient for table constraints, but can work well for disjunctive subproblems.
- A notinstance-constraint cuts off search branches that are subsumed. This is not really relevant to table constraints, but very important for operating on subproblems.

This is further discussed in section 7.1.

4 Multivalued Decision Diagrams

4.1 Introduction

Multivalued Decision Diagrams (MDD) are a way of representing a relation or a truth function. They are a generalisation of Binary Decision Diagrams (BDD) which have been used with much success in logic circuit optimisation.

For our purposes, an **MDD** is a directed acyclic graph (DAG) with a root node and (one or two) terminal nodes, where each path from the root to a terminal node represents a (set of) variable assignments and their truth value. Most frequently, we have only one terminal node and represent only success tuples.

An MDD can be thought of as derived from a complete search tree for a constraint problem in the following way. Consider a search tree like the one in figure 1 which represents the search for all solutions of the 6-queens problem. By removing all the failed branches, we obtain a decision tree (or *trie*) in figure 2 that represents the solution set. In this example, no further simplification is possible.

Looking at the similar example of the 5-queens problem (the trie in figure 3) with 10 solutions, we see that the variable assignments at the bottom of the tree can be shared. If that is done systematically, we obtain the MDD in figure 4. While the space savings are modest in this example, they are potentially exponential. As an example, figure 5 shows a representation of all 120 solutions of an alldifferent constraint with 5 variables.

We see that one property of such an MDD is that every horizontal level corresponds to one variable and its possible values.

4.2 MDD Construction

MDDs can be constructed in several ways:

- During search
- From a table of tuples
- By combining other MDDs

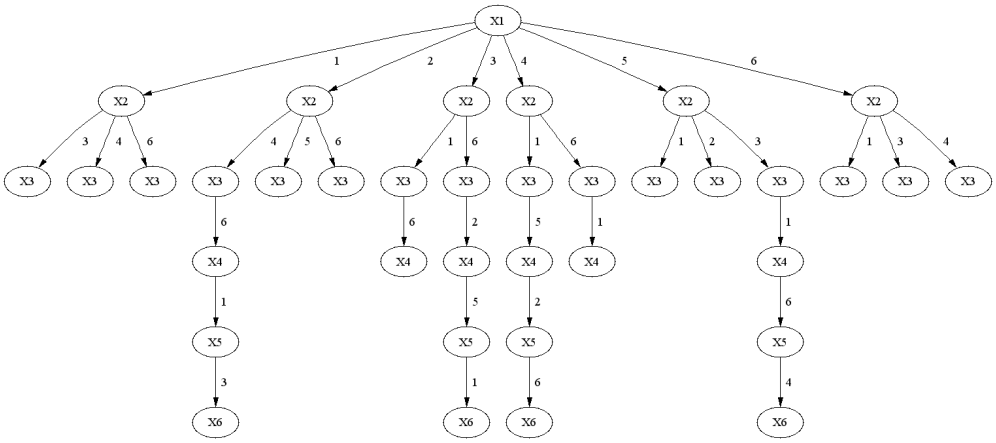

Figure 1: Search Tree (6 queens)

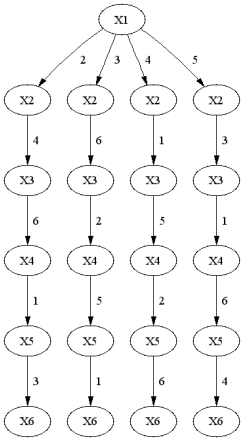

Figure 2: Decision tree (trie, 6 queens)

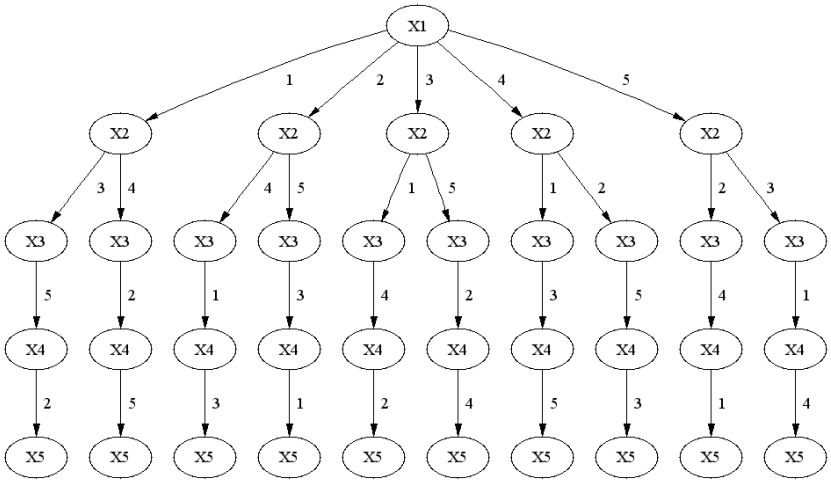

Figure 3: Decision tree (trie, 5 queens)

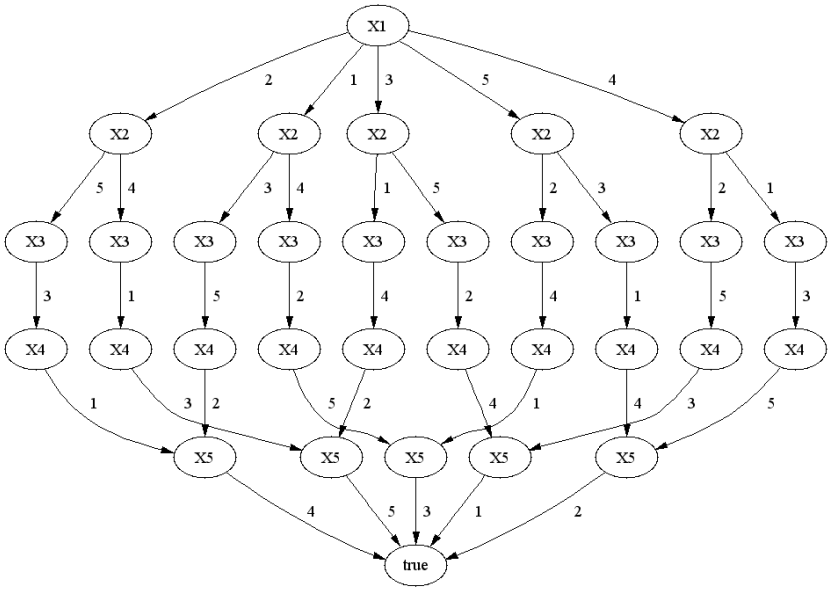

Figure 4: MDD (5 queens)

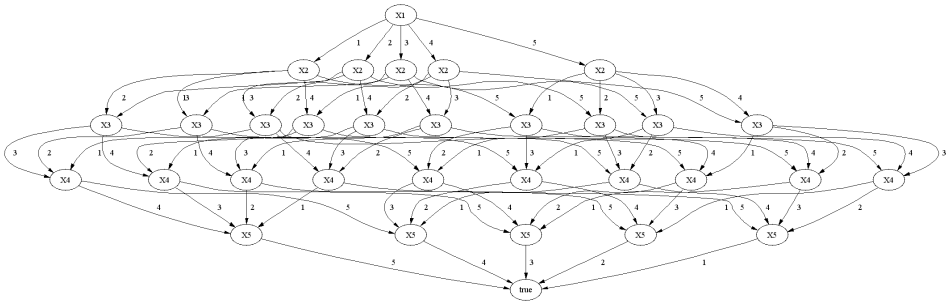

Figure 5: MDD (alldifferent over 5 variables)

```
mdd_findall(Xs, MDD) :-
        mdd_init(Root, MDD),
        ( mdd_findall(Xs, 1, Root, MDD) -> true ; mdd_false(Root) ).

mdd_findall(Xs, K, ThisNode, MDD) :-
        ( K > arity(Xs) ->
            mdd_true(ThisNode)
        ;
            arg(K, Xs, X),           % variable selection
            K1 is K+1,
            findall(X-SubNode, (
                    indomain(X),     % value selection
                    mdd_findall(Xs, K1, SubNode, MDD)
                ), Arcs),
            Arcs = [_|_],            % fail if no solution
            mdd_mk_node(K, Arcs, ThisNode, MDD)
        ).
```

Figure 6: Making an MDD during constraint-based search

- Top down or bottom-up

For the construction algorithms below, we assume that we have a non-backtrackable representation for MDDs, which can be built up incrementally by adding nodes one by one (mdd_init and mdd_mk_node).

4.2.1 MDD Construction during search

An MDD can be conveniently constructed during a standard constrained search process. The algorithm is given in figure 6. We assume as input an array of constrained variables Xs, and construct an MDD representing all solutions. The MDD variable order is the order in which the variables were selected during search. Construction is bottom-up, and we assume that we have an efficient mechanism (within mdd_mk_node) to detect reusable nodes, i.e. nodes that refer to the same variable index K and have the same outgoing value edges, leading to the same descendant nodes. What the algorithm does is essentially

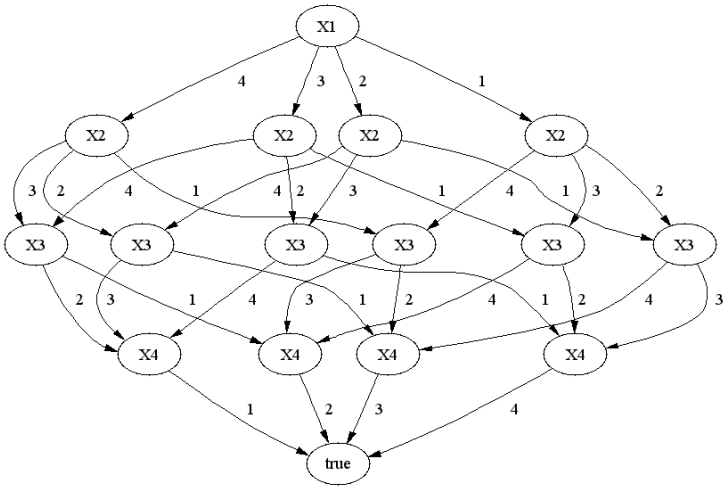

Figure 7: MDD paths leading to equivalent subtrees (alldifferent)

to record all branches in the search tree that lead to success, and represent this tree in compact MDD form.

This constructions of course takes as much time and effort as finding all solutions. In [HHOT08] it is suggested to exploit constraint properties to predict the outcome of a partial instantiation to detect equivalent subtrees without fully exploring. Figure 7 shows the example of the alldifferent constraint, where the 3 paths that consume the set of values 1,2,3 are known to lead to the same sub-MDD, without having to construct this sub-MDD 3 times. However, the technique relies on particular semantic properties of constraints, and thus seems to have limited applicability.

4.2.2 MDD Construction from tuple table

When a table of valid tuples is given, an MDD can be constructed more efficiently using algorithm 8. It assumes as input an ordered table of tuples, although the order is only relevant in the sense that tuples with a common prefix must be adjacent. The MDD is constructed with a variable order corresponding to the tuple argument order from left to right. The complexity of this algorithm is O(NumberOfTuples*Arity).

4.2.3 MDD Combination

MDDs can be conjoined (intersected), but in the worst case the result may be exponentially larger than the inputs. Corresponding code is provided in the accompanying library 9.1.9. In principle, one can solve a problem by starting with MDDs for its individual constraints, and conjoining them pairwise until an MDD for the whole problem is reached. The latter can then be used to efficiently enumerate the solutions. The problem is of course that even if the initial and the final MDD are compact, it is likely that the intermediate MDDs explode in size.

```
mdd_from_ordered_tuples(Tuples, MDD) :-
        mdd_init(Root, MDD),
        arg(1, Tuples, DummyTuple),
        level_node(Tuples, DummyTuple, 1, 1, _N, MDD, Root).

% Make sub-MDD node for the sub-table with common prefix
% PrefixTuple[1..K-1], starting at Tuples[I1].
% K in 1..Arity+1 (if K=1, PrefixTuple is dummy).
% I1 is the next tuple number to consider.
level_node(Tuples, PrefixTuple, K, I1, I2, MDD, Node) :-
        ( K > arity(PrefixTuple) ->
            I2 is I1+1,
            mdd_true(Node)
        ;
            collect_arcs(Tuples, PrefixTuple, K, I1, I2, MDD, Arcs),
            % Now Arcs == []
            mdd_mk_node(K, Arcs, Node, MDD)
        ).

% Collect arcs for all the different values at argument position
% K with identical prefix 1..K-1.
collect_arcs(Tuples, PrefixTuple, K, I1, I3, MDD, Arcs) :-
        (
            I1 =< arity(Tuples),
            arg(I1, Tuples, Tuple),
            same_prefix(Tuple, PrefixTuple, K)
        ->
            arg(K, Tuple, Xk),
            Arcs = [Xk-SubNode|Arcs1],
            K1 is K+1,
            % make mdd for sub-table with prefix Tuple[1..K]
            level_node(Tuples, Tuple, K1, I1, I2, MDD, SubNode),
            collect_arcs(Tuples, PrefixTuple, K, I2, I3, MDD, Arcs1)
        ;
            Arcs = [], I3=I1
        ).

% Two tuples are identical on position 1..K-1
same_prefix(Tuple, PrefixTuple, K) :-
        ( for(I,1,K-1), param(Tuple,PrefixTuple) do
            arg(I, PrefixTuple, X),
            arg(I, Tuple, X)
        ).
```

Figure 8: Making an MDD from a table of tuples

4.2.4 Top-down Construction

In [HHOT08], an alternative way of MDD construction is put forward. One starts with an approximate MDD (e.g. one that that allows all solutions). This approximate MDD can then refined by node-splitting until the solution set is represented precisely. This generalises to conjunction: further constraints can be incorporated by further refining the existing MDD in the same way.

4.3 MDD Properties and Comparison

An MDD can be seen as

- a summary of (the successful branches of) a search tree
- a description of a tuple set
- an indexing structure for valid tuples

The description is often more compact than a simple table of valid tuples, since common prefixes and suffixes are shared. It potentially is exponentially more compact than a table. It is also more compact than a decision tree (where only prefixes are shared).

To enable sharing of common prefixes and suffixes, it is necessary that the variable order in the MDD is the same for every path. But this has the disadvantage is that the MDD is order-sensitive in several ways. Different orders can lead to vastly different diagrams, and thus different diagram sizes for the same represented set.

An MDD is therefore not necessarily a good choice when the consequences of an arbitrary partial variable assignment are to be computed. It doesn't have any indexing-properties when only variable values "in the middle" of the diagram are known.

An MDD is a combination of an indexing structure and a set representation: When values for the top k variables are given, then the top k MDD layers serve a an index, and the bottom layers as a representation for the resulting set.

When we disregard the above considerations of variable order and its influence on MDD size, we can observe that MDDs are good at compactly representing certain solution sets but not others. In particular, solution number is not a good indicator, as can be seen by comparing the MDDs for

- a single solution over 5 variables (5 nodes)
- alldifferent constraint on 5 variables, domain size 5 (120 solutions, 32 nodes, figure 5)
- N-queens on 5 variables (10 solutions, 32 nodes, figure 4)
- all solutions of 5 variables, domain size 5 (3125 solutions, 5 nodes)

It would be interesting to investigate in more detail what constraints make simple MDDs, and which don't.

Because of the relationship with search trees, one could suspect that that MDDs constructed during a search with a good, compact heuristic search tree would also lead to the most compact MDDs, but the following table shows that

Queens		Left-to-right		Middle-out	
N	Solutions	Backtracks	MDD Nodes	Backtracks	MDD Nodes
7	40	48	147	40	141
8	92	172	287	138	307
9	352	622	971	466	1102
10	724	2086	2452	1340	2533
11	2680	8868	8002	5047	8911

Figure 9: N-queens search tree vs MDD size

this is not necessarily the case. Surprisingly, in the N-queens example in figure 9, a better search strategy with significantly fewer backtracks leads to larger MDDs to represent the same solution set.

4.4 MDD Uses

Once an MDD is constructed, in can be used in a variety of ways:

- Efficiently testing a variable assignment against the constraint that the MDD represents.
- Efficiently enumerating all valid assignments under the constraint in represents (e.g. as underlying data structure for the MDDC algorithm, section 4.5)
- Composition of MDDs: conjunction and disjunction of constraints.
- Use for guiding search in the context of a larger problem.

4.5 Propagation with MDDs

The following algorithm is the MDDC algorithm from [CY08]. An actual implementation can be found in our *mdd* library (section 9.1.9). The algorithm works on a fixed (precomputed) MDD representation of the constraint to be propagated, and rebuilds the supported domains whenever any support may have been lost. Its principles are:

- Traverse the MDD depth-first. The nodes correspond to variables and the edges to their values.
- Ignore branches that are not allowed according to the current variable domains (but don't actually compute a simplified MDD).
- During traversal, collect all supported domain values. These are those that lie on an allowed path to a solution (the true-node).
- Remember MDD nodes that lead to a solution, to avoid traversing shared sub-MDDs repeatedly. This memory is only kept for the duration of the propagation step, since it may be invalidated by the future loss of domain values caused by other constraints.

- Remember MDD nodes that lead to failure, to avoid traversing shared sub-MDDs repeatedly. This memory is kept for future propagation steps, and reset only on backtracking. A special sparse-set data structure is used to speed up resetting these marks on backtracking.

The algorithm has shown to be competitive with other approaches for GAC on n-ary constraints, even though it is of the coarse-grained type.

4.6 Approximate MDDs

The authors of [HHOT08] suggest the use of approximate MDDs to limit the space requirements. In particular, it is suggested to maintain MDDs of limited *width* (number of nodes in one level of an MDD). Such MDDs would be approximate in the sense that they allow too many solutions, but could be refined as needed (using node splitting as mentioned in 4.2.4) in order to exclude solutions whose exclusion is important for efficiency.

4.7 Constrained Decision Diagrams

So far we have only considered variable=value pairs on the edges of an MDD. A simple generalisation is to allow other unary constraints, e.g. $X = 3$, $X >= 5$, $X \in 2..7$, $even(X)$ etc. while maintaining the condition that branches must be mutually exclusive. While this can broaden the number of solution sets that an MDD can efficiently represent, it raises questions with regard to canonicality of such representations, makes operations for combining such MDDs more complex and unpredictable in the size of their result representation.

In [CY05], the notion of Constrained Decision Diagrams is introduced. There are actually two separate ideas here, which the authors don't clearly separate:

- the use of more general *splitting constraints* on the branches of the decision diagram.
- the interpretation of the decision diagram in the context of other constraints.

The first idea, although initially appealing, is fraught with many problems, and indeed not pursued in much depth in [CY05]. We will just discuss it briefly and then proceed to the second one. Like an MDD, a CDD is a rooted DAG with a terminal node for true (and possibly one for false), but the edges are annotated with more general *splitting constraints*. An MDD is then a special case of a CDD where the splitting constraints are of the form `X=c`. With the general definition, splitting constraints with a common parent do not have to be mutually exclusive. Also, variables can occur repeatedly in different levels of the diagram. A CDD is said to be reduced if it does not have sub-CDDs that have the same variable set and the same semantics. Reducedness is claimed not necessary for many applications.

4.7.1 Using MDD/CDDs in the context of other constraints

If an MDD is constructed during all-solution search of a general constraint problem, the resulting MDD represents the original constraints plus the outcome

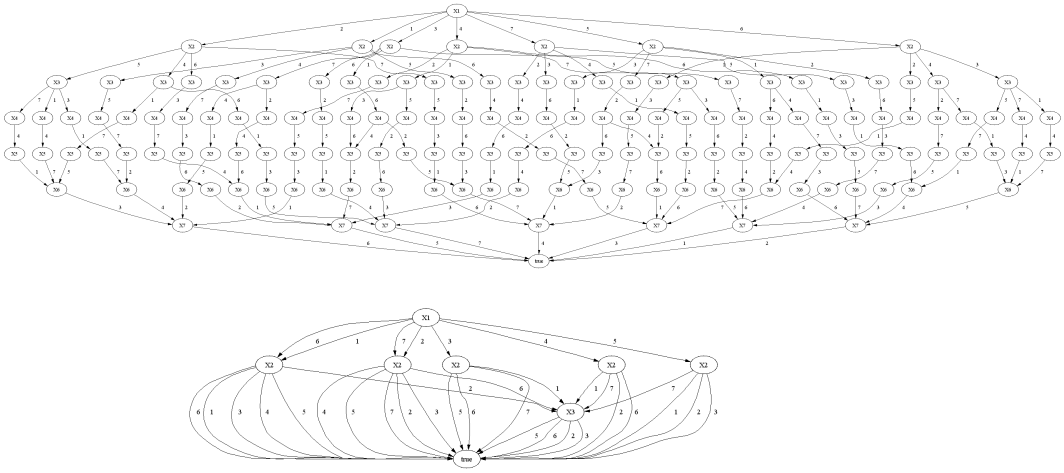

Figure 10: MDD vs. CDD for 7-queens

of the solution process. In particular, this MDD can be used *instead of* the original constraints to describe the same problem.

If however, an MDD is always interpreted in the context of the original constraints, then a lot of the information in the MDD is redundant (replaced by propagation of the original constraints), and the MDD can be replaced by a much simpler one. The number of cases that need to be distinguished at each particular node is greatly reduced, and nodes can be merged even when they differ in outgoing edges that are active only in certain propagation contexts. As a consequence, there are much more opportunities to merge nodes than there are in a standard MDD. Figure 10 shows the striking contrast between the MDD and CDD for the 7-queens problem.

This reduced MDD then only needs to capture the additional information that was learnt during search and which is not obvious from the original constraints. It can be interpreted as either

- a redundant constraint
- an oracle that avoids searching subtrees that don't lead to solutions

In other words, such a CDDs can be used to remember search results in a compact way.

Unfortunately, contrary to what seems to be suggested in [CY08], the MDDC algorithm is not applicable to such CDDs: the way nodes are merged in CDDs precludes the use of the node-marking techniques that are essential to MDDC.

4.8 Related Techniques

There are a number of other, related, techniques that can be used to represent extensional constraints or a sub-problem's solution set.

Compressed decision trees as explored in [KW07]

Algorithms from Boolean function minimization for example the Espresso algorithm [RSV87], which is an efficient approximate variant of the better known exponential Quine-McCluskey algorithm.

Finite Automata can be used to represent the set of valid tuples in a constraint as the accepted language of an automaton. Nondeterministic automata are compact, but making them deterministic, (or constructing an MDD from them) may cause exponential growth. It would be interesting to explore the right balance between space saved by the deterministic representation, and time wasted working with the nondeterministic representation.

Binary Decision Diagrams can be used with appropriate binary encodings of a CSP. The paper [HHO08] compares MDDs, BDDs and minimized DFAs, and found MDDs and MDFAs to be very similar, with BDDs competitive only if a log-encoding of the CSP is used.

5 Subproblem Selection

The selection of suitable subproblems for preprocessing is a difficult problem in itself. To begin with, it is not even clear what properties a good subproblem should have. Intuitive ideas are that subproblems should

- be tightly connected clusters of variables/constraints
- or, alternatively, connect parts of the problem that are not close in terms of the original constraints
- cover cycles in the constraint graph
- be simple enough in some sense, e.g. have a compactly representable and efficiently computable set of solutions

In many cases, there may be an obvious problem-specific structure, which can be described by the modeller, and then be exploited. Examples of such topologies are

- chains (or trees) of somewhat overlapping subproblems
- many subproblems connected by a single global constraint
- global constraints that overlap in many variables

To find such structures automatically in a larger problem, we consider two basic approaches: the decomposition of the constraint graph based on its topology; and the clustering of individual constraints, not necessarily based on topology. Further degrees of freedom in subproblem selection are:

- require them to be pure conjunctions of original constraints, or allow them to contain projections of the original constraints.
- do or do not allow them to have overlapping variables
- do or do not require that all original constraints must be covered by a subproblem

- do or do not allow original constraints to belong to more than one subproblem

5.1 Decomposition

When making subproblem by decomposition of the whole problem, we usually assume that variables should be in the same subproblem when they are connected by many constraints. This may or may not be a valid assumption.

Treewidth Treewidth is an important concept in complexity theory ([ST93], [Dec03]), and various algorithms for computing tree decompositions of graphs exist. Computing a tree decomposition with minimal tree-width is NP-hard. Nevertheless, an approximate algorithm may be usable.

Spectral Clustering Spectral clustering is a technique that uses eigenvalues to measure the importance of nodes in a network. One can start with an adjacency matrix that is real-valued to express constraint tightness. The dominant eigenvector is then found by a power iteration algorithm. The nodes (variables) with the highest values can be separated out as the first subproblem. The procedure is then repeated with the remainder of the constraint graph. This is somewhat related to e.g. the Google page-rank algorithm.

Greedy Techniques It is not clear whether algorithms of great theoretical sophistication are necessary to obtain a useful decomposition, keeping in mind that constraints may or may not tightly link the variables that they are involved with.

It is easy to devise other algorithms that may be just as appropriate.

- variable-oriented: maximum-cardinality ordering: start from the most connected variable, and incrementally add the variables that are most connected to the set collected so far.
- constraint-oriented: start from one constraint and add other constraints with maximum overlap, minimum number of new variables, or similar heuristics.

These techniques give an ordering, which then still needs to be partitioned by cutting "narrow" connections - however, no such obvious cutting opportunities may exist, and even if they do, they may not lead to clusters of manageable size.

5.2 Clustering

An alternative to decomposition is to simply group conjunctions of constraints in the hope that we can improve propagation behaviour of the conjunction.

Topological Clustering An obvious approach is again the use of the constraint graph topology. For instance, it seems natural to group constraints that overlap in many variables, or constaints that form tight cycles. Cycle elimination can be very important, since cycles lead to cycles during constraint propagation, which in turn lead to time consuming fixpoint iterations.

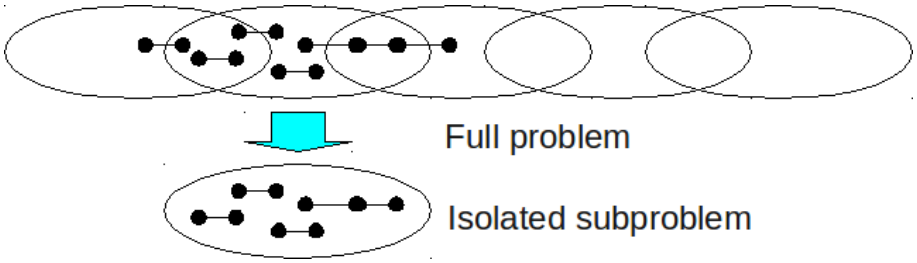

Figure 11: Problems with chain topology

Impact-based Clustering A different idea is to ignore the static constraint graph completely, and instead try to predict which variables affect which others via propagation. By observing the effect of test-instantiations, one can reconstruct an "effective" constraint graph, and use this as a heuristic for grouping variables into subproblems.

Creating new Connections It is actually not clear at all that we want to cluster adjacent constraints or variables. Since in the end we are generating redundant constraints to improve propagation, it might be much more important to connect distant variables with these new constraints. This essentially strengthens consistency by introducing a flavour of path consistency.

6 Experiments

6.1 Random Problems

We conducted experiments with random problems of a chain-like topology (figure 11), where each sub-problem overlaps with two others (except the ones at the ends of the chain, which overlap only with one). The overlap consists in a number of variables shared between two subproblems. The problem generator is parameterised by *NSub* (number of subproblems in the chain), *NSVar* (number variables per subproblem), *NOlap* (number of variables in every overlap), *D* (domain size), *PCX* (probability of constraint existence), and *PCS* (probability of each constraint being satisfied). The problems are generated by generating *NSub* standard random problems (using parameters *D*, *NSVar*, *PCS*, *PCX*). To achieve the desired topology, we fixed PCX to 1.0, i.e. there is a constraint between any two subproblem variables. We then combined adjacent subproblems by equating *NOlap* overlap variables, and removed half the constraints among the overlap variables in order to maintain the original even constraint density within the subproblems.

In the following result tables, the parameters are given as *D-NSub-NSVar-NOlap-PCS*. We generated 20 problem instances for each parameter set. The next column (%sat) gives the percentage of satisfiable problems in that class. The rest of the columns give the cumulative timings in milliseconds for the 20 problems in the class. We used a timeout of 10000ms, so the maximum time that can occur here is 200000. A $>$ sign indicates that at least one of the instances caused a timeout (and contributes 10000 to the total timing). For satisfiable problems, we measure the time to find the first solution. We looked at chains of

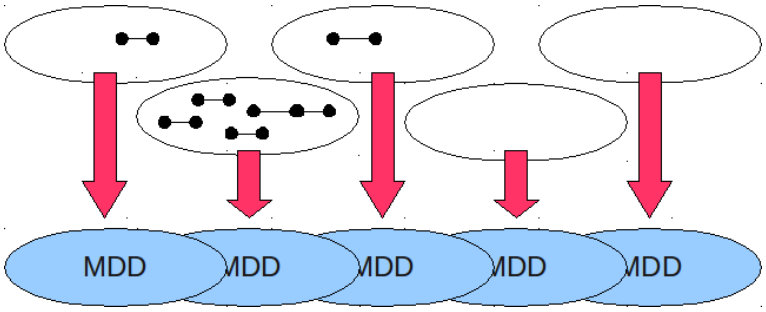

Figure 12: Replacing subproblems fully by MDD constraints

50 and 40 subproblem, where each subproblem consists of 9 resp. 12 variables, and which overlap in 2-4 variables. We omit results with 1-variable overlaps because they are not interesting with respect to to MDDs.

6.2 Algorithms

Our objective was to investigate algorithms that exploit the subproblem structure by computing solution sets represented as MDDs. For purposes of comparison, we also looked at the simpler idea of just probing subproblems for satisfiability. All algorithms have in common that they first preprocess the problem based on the subproblem structure, and then proceed to search over the full problem. The search method used for the final search on the full problem is always the same (first-fail for variable selection, and simple enumeration for values). In particular, this last stage it is independent of the problem structure, because we are interested in a preprocessing technique that can improve behaviour on structured problems in an orthogonal way. We always compare against the *naive* algorithm without any preprocessing.
Our algorithms are labelled with a 6-letter code, describing the the different characteristics of subproblem preprocessing, which we detail in the following.

Information collected from subproblems (1st letter) We looked at three alternatives here: (p) checking subproblems for satisfiability only; (f) building MDDs for whole subproblems; (o) building MDDs only for subproblem overlaps.

Satisfiability checking simply means that a subproblem is test-labelled, and early failure is detected if there is no solution.

The full MDD variants compute all solutions for every subproblem, represent these as MDDs, and then solve the full problem using GAC global constraints made from these MDDs (using the mddc algorithm, see section 4.5). The basic algorithm is sketched in figure 12. The details are dependent on the choice of the other parameters.

The overlap MDD variant computes all solutions for every subproblem, but only builds an MDD to represent all valid tuples for those variables that overlap with the next neighbour problem. The basic algorithm is sketched in figure 13.

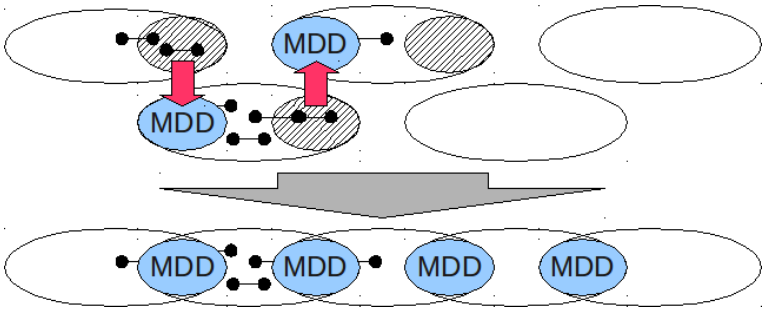

Figure 13: Creating MDD constraints for the overlaps only

We begin at one end of the subproblem chain, and compute all solution tuples for the variables in the overlap with the next subproblem. These are represented as an MDD, and set up as an additional constraint for the solution of the next subproblem, which in turn computes all solution tuples for the overlap with the following subproblem. Eventually, the subproblems with the additional MDD constraints are joined into a full problem representation, and solved as usual.

Starting domains used in subproblem solving (2nd+3rd letter) We looked at different starting domains for subproblem solving: (i) the initial variable domains, only reduced by the constraints among the subproblem variables; (s) the initial variable domains, reduced by the constraints of the subproblems already considered (the left half of the subproblem chain); (f) the initial variable domains, reduced by the constraints of the full problem setup. In addition to the above, we either further strengthen the domains using the MDD constraints created so far (s) or not (w).

Constraints visible to subproblem solving (4th letter) We looked at different constraint setups for the subproblems: We either consider: (i) only the internal constraints among subproblem variables; (s) the internal constraints plus the ones for the subproblems already considered (the left half of the subproblem chain); (s) all original constraints (the full chain). No matter how many constraints are considered, we always only label subproblem variables.

Full problem setup (5th+6th letter) The last variations are about how the full problem is set up using the results of the subproblem preprocessing: (b) batch mode, where all MDDs are computed first, then they are set up as constraints; (i) incremental mode, where the MDD constraints are set up as soon as they become available; (n) for no MDD constraints. Furthermore, we distinguish whether our full problem setup contains (m) MDD constraints only; (b) or both original and MDD constraints; (o) for original constraints only.

6.3 Results

All experiments were done using ECLiPSe 6.0, its IC finite domain solver, and the libraries random_csp, ic_ac3 and mdd (see section 9), on a Linux PC with Intel Core i7 920 2.67GHz processor.

Parameters	%sat	naive	pswcio	pfwcno	pswsio	pfwsno
5-50-9-2-0.65	0	630	110	710	70	710
5-50-9-2-0.70	0	1050	1090	1520	470	1090
5-50-9-2-0.75	0	>136380	>137110	>137120	>98180	>34890
5-50-9-2-0.80	100	2130	3380	3410	3700	4780
5-50-9-2-0.85	100	2020	3210	3210	3170	3930
5-50-9-3-0.65	0	640	110	730	110	740
5-50-9-3-0.70	0	880	1050	1470	250	950
5-50-9-3-0.75	0	4180	6250	6210	5000	2960
5-50-9-3-0.80	100	>12650	>13920	>13950	>15390	>17690
5-50-9-3-0.85	100	1840	3090	3060	3510	4600
5-50-9-4-0.65	0	510	80	670	70	670
5-50-9-4-0.70	0	860	1390	1760	310	940
5-50-9-4-0.75	0	1370	3490	3530	2470	1680
5-50-9-4-0.80	75	>94130	>97730	>97500	>99850	>103290
5-50-9-4-0.85	100	1360	3470	3050	3950	5670
5-40-12-2-0.70	0	1230	150	1150	150	1140
5-40-12-2-0.75	0	1660	520	1430	470	1320
5-40-12-2-0.80	0	>69720	>68400	>68520	>48810	9170
5-40-12-2-0.85	100	5100	7060	7100	7160	8410
5-40-12-3-0.70	0	1130	160	1060	210	1110
5-40-12-3-0.75	0	1910	410	1280	380	1350
5-40-12-3-0.80	0	3800	7380	7330	3810	2610
5-40-12-3-0.85	100	5730	7580	7630	9290	12020
5-40-12-4-0.70	0	1030	190	1030	160	1140
5-40-12-4-0.75	0	1640	520	1280	400	1370
5-40-12-4-0.80	0	2390	6210	6230	1580	3090
5-40-12-4-0.85	100	>44480	>49360	>49380	>53490	>60060

Figure 14: Variants with satisfiability checking only

Variants with satisfiability checking only Table 14 compares 4 variants of subproblem satisfiability checking algorithms: All algorithms check subproblems for satisfiability and cause early termination if an unsatisfiable one is discovered. Clearly, this will never help on satisfiable problems, as any overhead is wasted in this case. The variants are:

pswcio: One subproblem at a time is set up, checked for satisfiability, and if successful, added to the full problem setup. For each new subproblem, any domain reductions caused by already set up neighbours is taken into account.

pfwcno: The whole problem is set up first, the subproblems are extracted one by one and checked for satisfiability. For each subproblem, any domain reductions caused by its neighbours is taken into account.

pswsio: One subproblem at a time is added to the full problem setup, and then all its variables are test-labelled to check for satisfiability. *All* constraints set up so far remain active, i.e. the subproblem is not isolated from the rest.

pfwsno: The whole problem is set up first, then each set of subproblem variables is test-labeled individually to check for satisfiability. However, in this process, *all* constraints remain active, i.e. the subproblem is not isolated from the rest.

None of the variants can be singled out as clearly superiour, and the wide spread of results for what is supposed to be one basic technique is remarkable. For satisfiable problems, none of the algorithms improve things, but pswcio has the least negative impact. For the easy unsatisfiable problems, the simplest algorithm (pswio) performs best, probably because it has the minimum overhead over the naive method. However it doesn't work well on the hard nonsatisfiable ones, where pfwsno is faster. This is explained by the fact that pfwsno takes into account the largest number of constraints and therefore prunes better, but as it causes propagation sequences outside the subproblem, its runtime is less predictable than an algorithm that works on an isolated subproblem.

Variants of non-isolated full MDD generation Table 15 compares 4 variants of algorithms that compute full MDDs for the subproblems. They have in common that the subproblems are not solved in isolation, but with all or half of the original chain constraints being active.

ffssim: The whole problem is set up first, then subproblems are labelled and solved for all solutions, which are collected into MDDs. All original constraints are allowed to propagate. As soon as an MDD is produced, it is added as a constraint to a separate full problem setup, which is eventually used for full solving.

ffwsim: The same as ffssim, but the conjunction of MDDs generated so far is exploited to strengthen the subproblem domains before subproblem solving.

fsssib: This is one of the most straightforward variants: the subproblems are added one by one, and each time immediately labelled and solved for all solutions, which are collected into an MDD. This happens in the presence of the half chain of subproblems set up so far, with both the original constraints and the already generated MDD constraints.

fsssim: Similar to above, but the original constraints are removed and replaced by their MDD equivalent when it becomes available. At any time, we have a chain of MDD constraints. The next subproblem is added with its original constraints, then solved for all solutions (with the MDD constraints active), and the resulting MDD is used to replace the original subproblem constraints.

Remarkably, the algorithms that work on a full setup of original constraints (ffwsim, ffssim) perform much worse than those that work with only the "chain so far". There are two opposing effects: when working with the full set, there should be less subproblem solutions and smaller MDDs, but on the other hand, unpredictably long propagation sequences are likely. The fsssib and fsssim algorithms strengthen the "chain so far" before extending it with a new subproblem. This presumably causes the new subproblem to solve faster and not cause

Parameters	%sat	naive	ffwsim	ffssim	fsssib	fsssim
5-50-9-2-0.65	0	630	750	710	130	60
5-50-9-2-0.70	0	1050	1340	1080	240	260
5-50-9-2-0.75	0	>136380	18990	11780	4900	3900
5-50-9-2-0.80	100	2130	>200000	>200000	125820	111070
5-50-9-2-0.85	100	2020	>200000	>200000	>200000	>200000
5-50-9-3-0.65	0	640	740	760	70	60
5-50-9-3-0.70	0	880	960	950	170	180
5-50-9-3-0.75	0	4180	6060	3700	1310	1090
5-50-9-3-0.80	100	>12650	>200000	>200000	>193420	145780
5-50-9-3-0.85	100	1840	>200000	>200000	>200000	>200000
5-50-9-4-0.65	0	510	720	720	80	80
5-50-9-4-0.70	0	860	950	930	210	200
5-50-9-4-0.75	0	1370	3110	2370	910	740
5-50-9-4-0.80	75	>94130	>174250	>168840	>154240	>102010
5-50-9-4-0.85	100	1360	>200000	>200000	>200000	>200000
5-40-12-2-0.70	0	1230	1120	1140	160	170
5-40-12-2-0.75	0	1660	1360	1280	300	320
5-40-12-2-0.80	0	>69720	8090	4930	1840	1680
5-40-12-2-0.85	100	5100	>200000	>200000	>200000	>200000
5-40-12-3-0.70	0	1130	1110	1140	200	160
5-40-12-3-0.75	0	1910	1340	1350	330	310
5-40-12-3-0.80	0	3800	4150	3350	1160	1040
5-40-12-3-0.85	100	5730	>200000	>200000	>200000	>200000
5-40-12-4-0.70	0	1030	1100	1120	190	130
5-40-12-4-0.75	0	1640	1350	1330	310	270
5-40-12-4-0.80	0	2390	3420	3080	1080	1030
5-40-12-4-0.85	100	>44480	>200000	>200000	>200000	>200000

Figure 15: Variants of non-isolated full MDD generation

too much constraint activity outside the subproblem. Of the two better algorithms, fsssim is faster, presumably because it removes the redundant original constraints.

Variants for full MDDs with reduced SP domains The common characteristic of this group of algorithms is that they solve each subproblem in complete isolation, only letting its internal constraints propagate. They differ in the way the starting domains for the subproblems are obtained, and whether the full problem setup contains the original constraints.

ffscim: Subproblems solved in isolation, using domains resulting from a full setup of the original problem, plus strengthened by MDD chain so far. Full problem solved with MDD constraints only.

ffwcim: Subproblems solved in isolation, using domains resulting from a full setup of the original problem. Full problem solved with MDD constraints only.

fsscim: Subproblems solved in isolation, with initial domains that are strength-

Parameters	%sat	naive	ffscim	ffwcim	fsscim	ffscib
5-50-9-2-0.65	0	630	700	720	50	720
5-50-9-2-0.70	0	1050	900	1080	240	930
5-50-9-2-0.75	0	>136380	4480	6300	3880	4600
5-50-9-2-0.80	100	2130	48610	49080	48120	49890
5-50-9-2-0.85	100	2020	176030	175790	175330	175550
5-50-9-3-0.65	0	640	730	720	90	720
5-50-9-3-0.70	0	880	830	960	200	840
5-50-9-3-0.75	0	4180	2360	3290	1750	2490
5-50-9-3-0.80	100	>12650	48030	48540	47400	49840
5-50-9-3-0.85	100	1840	171450	171320	170780	172110
5-50-9-4-0.65	0	510	660	680	40	680
5-50-9-4-0.70	0	860	840	970	170	840
5-50-9-4-0.75	0	1370	1630	2170	1090	1700
5-50-9-4-0.80	75	>94130	>96810	>97240	>96360	>108130
5-50-9-4-0.85	100	1360	172220	172010	171710	174020
5-40-12-2-0.70	0	1230	1130	1090	220	1130
5-40-12-2-0.75	0	1660	1260	1380	340	1250
5-40-12-2-0.80	0	>69720	2810	4580	1920	2880
5-40-12-2-0.85	100	5100	>186240	>186340	>185320	>187250
5-40-12-3-0.70	0	1130	1070	1080	200	1040
5-40-12-3-0.75	0	1910	1200	1330	320	1190
5-40-12-3-0.80	0	3800	2090	3030	1240	2120
5-40-12-3-0.85	100	5730	175590	176580	174700	179380
5-40-12-4-0.70	0	1030	1050	1050	160	1050
5-40-12-4-0.75	0	1640	1160	1330	310	1160
5-40-12-4-0.80	0	2390	2050	2820	1160	2100
5-40-12-4-0.85	100	>44480	>182580	>184070	>182160	>188480

Figure 16: Variants for full MDDs with reduced SP domains

ened by MDD chain so far. Full problem solved with MDD constraints only. Original problem never set up.

ffscib: Subproblems solved in isolation, using domains resulting from a full setup of the original problem, plus strengthened by MDD chain so far. Full problem solved with both original and MDD constraints.

The results show that having an initial full setup of the original constraints does not bring any benefit, although it is not clear whether this would still hold when the original constraints were more complex than binary. What seems to be clear is that using the partial setup of the MDD constraints for reducing the initial domains of the next subproblem to process, is very important. Third, we see from comparing ffscim and fsscib that removing the redundant original constraints gives a small speedup.

Variants for full MDDs on isolated subproblems with initial domains In figure 17, we compare three variants that compute full MDDs on isolated subproblems without using the original constraints of neighbouring subproblem

Parameters	%sat	naive	fiwcbm	fiwcim	fsscim
5-50-9-2-0.65	0	630	50	90	50
5-50-9-2-0.70	0	1050	2730	450	240
5-50-9-2-0.75	0	>136380	15120	5720	3880
5-50-9-2-0.80	100	2130	48460	48630	48120
5-50-9-2-0.85	100	2020	174390	175680	175330
5-50-9-3-0.65	0	640	80	70	90
5-50-9-3-0.70	0	880	2230	310	200
5-50-9-3-0.75	0	4180	15350	2670	1750
5-50-9-3-0.80	100	>12650	47980	48100	47400
5-50-9-3-0.85	100	1840	169340	170550	170780
5-50-9-4-0.65	0	510	70	60	40
5-50-9-4-0.70	0	860	3080	390	170
5-50-9-4-0.75	0	1370	14860	1600	1090
5-50-9-4-0.80	75	>94130	>96840	>96760	>96360
5-50-9-4-0.85	100	1360	170630	171880	171710
5-40-12-2-0.70	0	1230	160	200	220
5-40-12-2-0.75	0	1660	620	490	340
5-40-12-2-0.80	0	>69720	26580	3710	1920
5-40-12-2-0.85	100	5100	>184650	>185780	>185320
5-40-12-3-0.70	0	1130	180	180	200
5-40-12-3-0.75	0	1910	630	470	320
5-40-12-3-0.80	0	3800	26490	2220	1240
5-40-12-3-0.85	100	5730	174640	175740	174700
5-40-12-4-0.70	0	1030	180	170	160
5-40-12-4-0.75	0	1640	660	480	310
5-40-12-4-0.80	0	2390	27610	1980	1160
5-40-12-4-0.85	100	>44480	>182010	>182720	>182160

Figure 17: Variants for full MDDs on isolated subproblems with initial domains

to obtain reduced starting domains. None of these algorithms ever fully sets up the original problem.

fiwcbm: First, all subproblems are solved in isolation, with initial domains. Then the full problem is set up with MDD constraints only (batch mode), and solved.

fiwcim: Subproblems are solved in isolation, with initial domains. The resulting MDDs are incrementally added to a full problem setup. Finally, the full problem is solved, with MDD constraints only.

fsscim: Like fiwcim, but subproblem domains are strengthened by the MDD chain so far.

Algorithm fiwcbm sets up the full problem after all the MDDs have been computed (batch), while fiwcim and fsscim interleave this setup with the MDD computation (incremental). The results show that the incremental methods are almost always better. Only for easy unsatisfiable problems is incrementality worse (because they are detected as infeasible in the subproblem solution process). For satisfiable problems, the strategy hardly makes any difference. We

Parameters	%sat	naive	pswsio	fsscim	ofwcib
5-50-9-2-0.65	0	630	70	50	110
5-50-9-2-0.70	0	1050	470	240	230
5-50-9-2-0.75	0	>136380	>98180	3880	2140
5-50-9-2-0.80	100	2130	3700	48120	12920
5-50-9-2-0.85	100	2020	3170	175330	13730
5-50-9-3-0.65	0	640	110	90	120
5-50-9-3-0.70	0	880	250	200	170
5-50-9-3-0.75	0	4180	5000	1750	730
5-50-9-3-0.80	100	>12650	>15390	47400	>30900
5-50-9-3-0.85	100	1840	3510	170780	32960
5-50-9-4-0.65	0	510	70	40	120
5-50-9-4-0.70	0	860	310	170	220
5-50-9-4-0.75	0	1370	2470	1090	630
5-50-9-4-0.80	75	>94130	>99850	>96360	>77120
5-50-9-4-0.85	100	1360	3950	171710	71500
5-40-12-2-0.70	0	1230	150	220	180
5-40-12-2-0.75	0	1660	470	340	350
5-40-12-2-0.80	0	>69720	>48810	1920	1440
5-40-12-2-0.85	100	5100	7160	>185320	>33610
5-40-12-3-0.70	0	1130	210	200	190
5-40-12-3-0.75	0	1910	380	320	330
5-40-12-3-0.80	0	3800	3810	1240	990
5-40-12-3-0.85	100	5730	9290	174700	>68180
5-40-12-4-0.70	0	1030	160	160	190
5-40-12-4-0.75	0	1640	400	310	290
5-40-12-4-0.80	0	2390	1580	1160	1000
5-40-12-4-0.85	100	>44480	>53490	>182160	>91470

Figure 18: Satisfiability testing vs. full MDDs vs. overlap MDDs

see again that strengthening the subproblem starting domains via the MDD chain so far is helpful.

Level of Information Extraction In figure 18 we try to appraise what kind of information is worth extracting from the subproblems. We compare naive solving, a simple checking algorithm, and two MDD variants.

pswsio: The best of algorithms that just check for subproblem satisfiability.

fsscim: The best of the algorithms that extract full subproblem MDDs.

ofwcib: The algorithm sketched in figure 13, which extracts MDDs only for the variables in which subproblems overlap. It proceeds along the chain, solving each subproblem, taking into account the MDD result from the previous subproblem, and computing an MDD representing all solutions for the overlap with the next subproblem. The full problem is then set up with the original constraints plus all MDD constraints as redundant constraints.

Parameters	sat	naive	pswsio	fsscim	ofwcib
5-50-9-3-0.80	true	timeout	timeout	2410	1080
5-50-9-4-0.80	true	timeout	timeout	3350	1240
5-50-9-4-0.80	true	timeout	timeout	timeout	6520
5-50-9-4-0.80	true	1980	2550	2430	1150
5-40-12-2-0.85	true	1780	1900	9950	1250
5-40-12-4-0.85	true	6380	7580	8720	3210
5-40-12-4-0.85	true	4490	5190	timeout	3860
5-40-12-4-0.85	true	7960	8910	9440	4340

Figure 19: Satisfiable instances that benefit from subproblem solving

Obviously, some unsatisfiable problems fail because one of the subproblems is already unsatisfiable. With satisfiability checking, the whole problem only has to be solved if all these checks fail. As the pswsio column shows, this approach works well for the more constrained problems (PCS at most 0.75) or when the overlap between the subproblems is small - in those cases, building MDDs is a waste of time.

A big improvement is obtained via MDDs for the hard unsatisfiable problems (PCS 0.75 to 0.8) and for some very hard satisfiable ones (see figure 19 for those individual instances). It also seems that the benefit can usually be obtained from the smaller overlap MDDs, and full MDDs are not necessary. This is good news and makes the approach more realistic in terms of MDD sizes.

However, almost all satisfiable problems suffer heavy performance penalties due to MDD construction (although less so with overlap MDDs).

6.4 General Conclusions

We have observed quite surprising differences in behaviour resulting from what initially seemed minor choices in the implementation.

Applicability Unsurprisingly, different techniques work for different problem classes. According to constraint tightness we can conclude

- The very easy unsatisfiable problems (PCS below 0.7) work best with a naive technique, and all treatment of subproblem structure just adds overhead.
- The harder unsatisfiable ones (PCS 0.7 to 0.75) can benefit from checking subproblems individually for unsatisfiability, because of the chance to detect failure without needing to address the problem as a whole.
- For any satisfiable problem, satisfiability checking alone is obviously always just an overhead and slows things down.
- Hard unsatisfiable problems (PCS 0.75 to 0.8) aren't helped by satisfiability checking. But this is the class that benefits most from subproblem solving, and the gains can be dramatic.
- For hard satisfiable problems (PCS 0.8 to 0.85), subproblem solving adds overhead, and only in very rare cases can we see a gain from it. However,

like for hard unsatisfiable problems, in these cases the gain can be dramatic (solvable versus timeout).

- Easier satisfiable problems are dramatically slowed down by any of our forms of subproblem treatment.

As a consequence, in any practical application of this method one would have to make sure that it is only applied to the really hard problem instances. This might be achievable by simply limiting MDD size or preprocessing time. We also note that, in optimization via the branch-and-bound method, the proof of optimality relies precisely on the solution of a tightened problem that is just no longer feasible. At this stage the problem is likely to be non-trivially infeasible, and thus hopefully a good candidate for our technique.

Subproblem definition In terms of subproblem definition, we had tried three techniques: Setting subproblems up completely independently; taking into account some or all of the original constraints; reducing starting domains with the help of non-subproblem constraints. The variants that consider neighbouring constraints can be seen as an extended form of singleton arc consistency, where more than one variable is being assigned. The drawback of this solution is that solving the subproblem involves more constraints and a larger constraint networks, and potentially long propagation sequences with fixpointing iterations. Our results seem to show that this is a serious problem, and that it seems better to treat subproblems in isolation.

This still leaves the option of exploiting information from the global problem when setting up the subproblem, by taking into account any propagation that already happened in the (fully or partially set up) global problem for reducing starting domains. According to our results, it is not helpful to use propagation from the original constraints (or to even set up the full problem with the original constraints). On the other hand, setting up MDD constraints as soon as the MDDs become available, and using the propagation results from the chain established so far, does provide an advantage.

6.5 Caveats

There are a number of caveats with these measurements. A major one is that the order of constraint waking and propagation is essentially undefined for finite domain propagation solvers. In practice, none of the common strategies (waking queue or stack) is likely to produce the optimal propagation sequence, which would (in the case of a chain-topology) probably consist in systematic sweeps all the way to the end of the chain, then back to the other end, until a fixpoint is reached. Since a subproblem propagator can cause waking of both its left and right neighbour, the actual activation order of the constraints in the chain will be rather chaotic.

Moreover, we have only explored a faction of the parameter space, in particular it is not very clear how chain length, overlap size and domain size affect the choice of algorithm. Overlap size will clearly have impact on whether full MDDs are competitive with overlap MDDs. Larger domain sizes may lead to larger MDDs which eventually become a memory problem.

7 Related Approaches

7.1 Generalised Propagation

7.1.1 Overview of GP

Generalised Propagation (GP) is a paradigm that establishes a framework for using a nondeterministically defined subproblems as active constraints within a bigger problem. It is thus closely related to our topic of investigation.

Let us first recall the better known constraint programming frameworks:

- CLP(X), where we have an effective solver for conjunctions of primitive constraints over domain X (at least equality), plus disjunction (search).
- CSP, where the variables are restricted to finite domains, but we have arbitrary constraint relations. Solving is by inferring domain information from individual constraints (propagation), and search.
- SAT, where we are restricted to the Boolean domain, and clauses as constraints. Solving is by unit propagation, nogood learning, and search.

In [LW93], the authors put forward *Generalised Propagation* (GP) as a flexible scheme which

- allows arbitrary predicates (and thus subproblems) to be treated as constraints
- subsumes, but does not rely on, a notion of domains
- is instead parameterised by the language of primitive constraints, e.g. GP(fd) for finite domain programming
- supports approximation of the constraint language, i.e. inference can be restricted to something weaker than expressible in the language of primitive constraints

The central step in GP is the extraction of *Approximation Constraints* from a subproblem. *Approximation Constraints* are primitive constraints that are a consequence of all the constraints they approximate, e.g. the domain constraint X in 1,2,3 approximates X=1, X=2 and X=3.

GP can be implemented using a Topological Branch-and-Bound algorithm, whose outline is as follows:

```
gp_propagate:
    Find a solution S1 in terms of Approximation Constraints
    Cover := S1
    Repeat until fixpoint
        Add a constraint disallowing instances of Cover
        Find a solution Si in terms of Approximation Constraints
        Cover := generalize(Cover,Si)
    Return Cover as propagation result
```

The idea is to collect alternative solutions, and compute their generalisations, until a cover of the whole solution set is obtained. If finite domain membership constraints are used as approximation constraints, this algorithm computes GAC on the problem it is applied to.

Approx. Language	Behaviour
$true/false$	consistency (can only express dis/entailment)
$X_i = c_i$	exactly one solution tuple (can only express a unique solution)
$X = c \wedge ...$	entailed instantiations
$X = f(\_) \wedge ...$	entailed partial instantiations
$X = Y \wedge ...$	entailed equalities
$X >= c \wedge ...$	entailed bounds
$X \in Domain \wedge ...$	entailed domains, equivalent to conjunction of disjunction of X=c assignments
$X_i = c_i \vee ...$	disjunction of tuples, or table constraint, or tuple domains, equivalent to disjunction of conjunction of X=c assignments

Figure 20: Approximation Languages for Generalized Propagation

The relevance of the GP framework for our purposes is that it extends the notion of domain-consistency in two directions: towards the weaker, and towards the stronger side. This is simply achieved by choosing different approximation constraints. Table 20 shows a number of "approximation languages", from weaker to stronger.

The original implementation of GP (ECLiPSe's library(propia) [ea08]) provides a subset of these, namely

- consistency
- unique solution
- a combination of entailed instantiations, equalities and optionally full domains

The introduction of MDDs for the representation of results allows us to extend this towards approximations that are stronger than just the domain cover.

7.1.2 More Precise Approximation in GP

The introduction of MDDs for the representation of results would allow us to extract more precise information from a subproblem.

In the extreme case, the Approximation Constraint is then a constraint represented by the table of satisfied tuples, represented as an MDD. Moreover, MDDs of other limited widths can be used conveniently to realise representations that trade space versus precision in the spirit of [HHOT08] and [AHHT07]. Interestingly, the established technique of inferring domains can be seen as a special case of this, using an approximate MDD of width 1 (where every node's outgoing edges are independent of those of its ancestors in the MDD).

7.1.3 Efficiency improvements in GP

MDDs can be used to improve efficiency even with the less precise approximation constraints, like the domain cover.

If the results of the initial propagation step are retained in the form of a (precise) MDD that represents all solutions, then there is no need for further

application of the TBB algorithm: it can be replaced by the MDDC algorithm (4.5) on the constructed MDD. Note however that this is not true when the MDD only represents a cover of subproblem propagation results, as these are not necessarily solutions of the subproblem.

In the case of approximate MDDs, the TBB algorithm can be used together with MDDC (4.5), thus reducing search in subsequent iterations of TBB. Alternatively, a variant of TBB could be devised that uses the MDD as a guide to its search process. This would effectively systematically seek support for the previously supported solution space approximation, rather than blindly recomputing the new cover from scratch. This is interesting because the naive TBB algorithm may search large spaces that don't lead to new solutions, because the notinstance constraint is constraint that propagates only weakly.

A future research topic would be to devise good heuristics to guide the the search from the MDD: variable selection, domain splitting and value selection could all benefit from the information contained in the MDD that resulted from the previous search iteration. Classical techniques like shaving can probably also benefit from this additional information.

7.2 GAC on Combinations of Constraints

The previously mentioned work [Lho04] explores ideas for computing GAC on combinations of constraints, in particular

Disjunction: If any sub-constraint is unsatisfiable, replace the disjunction with the other one. For the variables in the intersection, find support in one or the other sub-constraint. The only interesting observation here is that this is cheaper than solving both subproblems and intersecting the results. The known techniques for making this incremental can be used.

Negation: Find supports by failing to find a conflicting assignment for a given X=a for the negated constraint.

Conjunction: Find support in one constraint, then a matching one (with the same intersection assignments) in the other. If both are table constraints, then the same (e.g. hologram) data structure used for their individual GAC-schema propagators can be used to find these partially instantiated supports quite efficiently.

The requirement for the sub-constraints is that they provide the following operations: getSupport (find any support (solution) for one X=a); getNextSupport (find next support for one given instantiation X=a after a given support); getCommonSupport (find any support for a given set Xi=ai); getConflict (find an assignment that violates the constraint). An earlier paper investigating GAC on conjunctions is [KB01].

7.3 MDD-based constraint store

[AHHT07] investigates the idea of using a (single) MDD as the constraint store, replacing the constraint store consisting only of the variable domains. To keep memory requirements manageable, the MDD can be restricted in width (the width being the maximum number of nodes on any level of the MDD). With a

width of 1, the MDD-store degenerates into the classical domain store. With unlimited width, it can represent the problem completely (as an extensional representation of all solution tuples).

7.4 Relationship with Propagator-Generators

There is a body work on the generation of efficient constraint propagators from formal descriptions of the constraints, in particular automata ([CB04], [BCP04]). These techniques require the constraint to be specified in a particular formalism, while we are interested in subproblems that are arbitrary combination of of other constraints.

7.5 Relation to Singleton Arc Consistency

Singleton Arc Consistency works by trying to set individual variables to values from their domain, let the constraint system propagate, and if that leads to failure, eliminate the value. This is similar to "looking for support" in a GAC algorithm, except that since we can't positively establish support, we try to establish non-support instead. Only in case the remaining subproblems (after instantiating one variable) have GAC-strength propagation, will the variable indeed become arc-consistent by using this method.

8 Future Work

The central aim of this research was the exploitation of problem structure, in particular with respect to the generation of useful redundant constraints. The space of possible approaches and their variants is huge, and we have only looked at a very small area. We have seen in our experiments that small algorithmic differences can have big impacts on behaviour and usefulness of a technique, meaning that careful analysis is necessary. We have also seen that a smart technique can help with hard problems, but is likely to be a waste of time on easier ones. This suggests that an a priori estimate, or an on-the-fly estimate of problem hardness would be helpful loin deciding which algorithm to employ for solving.

In our opinion, automatic clustering methods are not necessarily the most pressing research area. For the practitioner, it would be more helpful to have tools that allow effective propagation strengthening, even if the subproblems are manually selected.

9 Software by-products

A number of ECL^{i}PS^e libraries have been created in the course of this work.

9.1 Solvers and Constraints

9.1.1 fdd

A simple integer finite domain solver with deletion notifications, for experimenting with fine-grained (or AC-4/6/7 style) algorithms. These type of algorithms

do their propagation based on the knowledge of which values have been removed from which variable's domain. This solver provides that information, while the standard solvers in ECL^{i}PS^e don't do this.

9.1.2 sd/fd/ic_notifiers

An auxiliary library that adds value-deletion notification on top of a solver that does not provide this feature (such as *fdd*). This is not as efficient as generating the notifications directly in the solver, because the information is derived by comparing old and new domains, but at least the overhead can be shared among several constraints.

9.1.3 ac3

Basic AC-3 algorithm for extensionally represented binary constraints, rebuilding the supported domain every time a support may have been lost. For lib(fdd).

9.1.4 ac3_dir

Variant of AC-3 algorithm with suppression of redundant backwards-propagation. For lib(fdd).

9.1.5 ac31

Coarse-grained algorithm like AC-3, known as AC-3.1, remembering supports and thus achieving optimal worst case complexity [BRYZ05]. For lib(fdd).

9.1.6 ac6

Fine-grained algorithm, remembering supports and incrementally re-establishing lost supports [BC93]. For lib(fdd).

9.1.7 gac3

A naive implementation of GAC (for lib(sd)) which essentially rebuilds the supported domains from scratch, using

```
( foreach(X,Xs) do
    findall(X, ( indomain(X), test(c(Xs)) ), SupportedValues),
    X :: SupportedValues
)
```

9.1.8 gac_schema

An implementation of GAC-schema [BR97] for allowed-tuple and predicate representation. For lib(sd) with lib(sd_notifiers).

9.1.9 mdd

The basic algorithms and tools for working with MDDs: *mdd_findall* to collect all solutions of a goal as an MDD (figure 6); *mdd_from_ordered_tuples* to convert a table of tuples into an MDD (figure 8); *mddc* implements the MDD-based GAC constraint propagator using the algorithm from [CY08]; *mdd_and* to conjoin two MDDs; *mdd_to_graph* to convert an MDD into a standard graph representation that can be manipulated and displayed;

9.1.10 sparse_sets

An auxiliary library for the MDD propagation algorithm from [CY08]. It implements a data type to store sets of integers, with the characteristic that the operations of member insertion, membership testing, and in particular resetting to an arbitrary earlier state (backtracking) are all constant-time operations.

9.2 Problem Analysis

9.2.1 constgraph

A dummy solver that builds a constraint graph, but does no semantic processing of the constraints or variables. The representation can then be queried for connectedness information. This library is the basis for topology-based decomposition techniques.

9.2.2 impact

Infrastructure for analysing propagation effects for the purpose of estimating connectedness in a constraint graph. This is based on collecting the variables that are affected in the course of each propagation sequence.

9.3 Benchmarking

9.3.1 random_csp

A library to generate random CSP problems, both binary and n-ary. For binary problems, the representation of a whole problem is a hash table, mapping a pair of variable indices to the corresponding constraint. A binary constraint is also a hash table, containing the allowed value pairs, which is a convenient representation for validity testing. For n-ary constraints, constraints are simple tuple tables, since the various propagation algorithms typically compute their own specialised index structures over this data.

9.3.2 read_xml

A library to read a subset of XCSP 2.1 format [RL09] (extensional constraints), and either convert them to Prolog tables or set up constraints directly.

9.3.3 Various

Various experimental solving strategies for testing, not packaged in library form.

References

[AHHT07] Henrik Reif Andersen, Tarik Hadzic, John N. Hooker, and Peter Tiedemann. A constraint store based on multivalued decision diagrams. In Bessiere [Bes07], pages 118–132.

[BC93] Christian Bessière and Marie-Odile Cordier. Arc-consistency and arc-consistency again. In *AAAI*, pages 108–113, 1993.

[BCP04] Nicolas Beldiceanu, Mats Carlsson, and Thierry Petit. Deriving filtering algorithms from constraint checkers. In Mark Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2004.

[Bes06] Christian Bessiere. Constraint propagation. Technical Report 06020, LIRMM, March 2006.

[Bes07] Christian Bessiere, editor. *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*. Springer, 2007.

[BFR95] Christian Bessière, Eugene C. Freuder, and Jean-Charles Régin. Using inference to reduce arc consistency computation. In *IJCAI (1)*, pages 592–599, 1995.

[BR97] Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks: Preliminary results. In *IJCAI (1)*, pages 398–404, 1997.

[BR99] Christian Bessière and Jean-Charles Régin. Enforcing arc consistency on global constraints by solving subproblems on the fly. In Joxan Jaffar, editor, *CP*, volume 1713 of *Lecture Notes in Computer Science*, pages 103–117. Springer, 1999.

[BRYZ05] Christian Bessière, Jean-Charles Régin, Roland H. C. Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *Artif. Intell.*, 165(2):165–185, 2005.

[CB04] Mats Carlsson and Nicolas Beldiceanu. From constraints to finite automata to filtering algorithms. In David A. Schmidt, editor, *ESOP*, volume 2986 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2004.

[CY05] Kenil C. K. Cheng and Roland H. C. Yap. Constrained decision diagrams. In Veloso and Kambhampati [VK05], pages 366–371.

[CY08] Kenil C. K. Cheng and Roland H. C. Yap. Maintaining generalized arc consistency on ad hoc r-ary constraints. In Stuckey [Stu08], pages 509–523.

[Dec03] Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

[ea08] Pascal Brisset et al. *ECLiPSe 6.0 Constraint Library Manual*, September 2008.

[GJMN07] Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Data structures for generalised arc consistency for extensional constraints. In *AAAI*, pages 191–197. AAAI Press, 2007.

[HHO08] Tarik Hadzic, Esben Rune Hansen, and Barry O'Sullivan. On automata, mdds and bdds in constraint satisfaction. In *Proceedings of the ECAI 2008 Workshop on Inference Methods based on Graphical Structures of Knowledge*, July 2008.

[HHOT08] Tarik Hadzic, John N. Hooker, Barry O'Sullivan, and Peter Tiedemann. Approximate compilation of constraints into multivalued decision diagrams. In Stuckey [Stu08], pages 448–462.

[KB01] George Katsirelos and Fahiem Bacchus. GAC on conjunctions of constraints. In Toby Walsh, editor, *CP*, volume 2239 of *Lecture Notes in Computer Science*, pages 610–614. Springer, 2001.

[KW07] George Katsirelos and Toby Walsh. A compression algorithm for large arity extensional constraints. In Bessiere [Bes07], pages 379–393.

[Lho04] Olivier Lhomme. Arc-consistency filtering algorithms for logical combinations of constraints. In Jean-Charles Régin and Michel Rueher, editors, *CPAIOR*, volume 3011 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2004.

[LR05] Olivier Lhomme and Jean-Charles Régin. A fast arc consistency algorithm for n-ary constraints. In Veloso and Kambhampati [VK05], pages 405–410.

[LW93] T. Le Provost and M.G. Wallace. Generalized constraint propagation over the CLP Scheme. *Journal of Logic Programming*, 16(3-4):319–359, July 1993. Special Issue on Constraint Logic Programming.

[Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[RL09] Olivier Roussel and Christophe Lecoutre. XML representation of constraint networks: Format XCSP 2.1. *CoRR*, abs/0902.2362, 2009.

[RSV87] Richard L. Rudell and Alberto L. Sangiovanni-Vincentelli. Multiple-valued minimization for pla optimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 6(5):727–750, 1987.

[SS05] Kostas Stergiou and Nikos Samaras. Binary encodings of non-binary constraint satisfaction problems: Algorithms and experimental results. *J. Artif. Intell. Res. (JAIR)*, 24:641–684, 2005.

[ST93] P. D. Seymour and R. Thomas. Graph searching and a min-max theorem for tree-width. *Journal of Combinatorial Theory, Series B*, 58(1):22 – 33, 1993.

[Stu08] Peter J. Stuckey, editor. *Principles and Practice of Constraint Programming, 14th International Conference, CP 2008, Sydney, Australia, September 14-18, 2008. Proceedings*, volume 5202 of *Lecture Notes in Computer Science*. Springer, 2008.

[vD02] Marc R. C. van Dongen. AC-3_d an efficient arc-consistency algorithm with a low space-complexity. In Pascal Van Hentenryck, editor, *CP*, volume 2470 of *Lecture Notes in Computer Science*, pages 755–760. Springer, 2002.

[VK05] Manuela M. Veloso and Subbarao Kambhampati, editors. *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*. AAAI Press / The MIT Press, 2005.