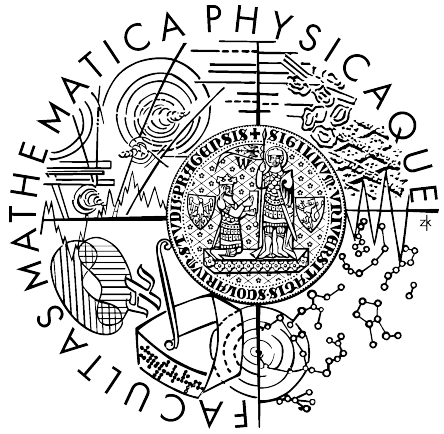
Charles University in Prague

Faculty of Mathematics and Physics



MASTER THESIS

Martin Molnár

Filtering Algorithms for Tabular Constraints

Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Doc. RNDr. Roman Barták, Ph.D.

Study programme: Computer Science, General Computer Science, Nonprocedural Programming and Artificial Intelligence

2010

I would like to thank my supervisor, Mr. Roman Barták, for being available during his vacation, for his leadership, support and many valuable comments on the content of this thesis and experiments.

I declare that I have written this thesis independently and by using the cited sources exclusively. *Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradne s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.*

In Prague, august 5th, 2010
V Praze, 5. srpna 2010 Martin Molnár

Table of Contents

Title:	Filtering Algorithms for Tabular Constraints
Author:	Martin Molnár
Department:	Department of Theoretical Computer Science and Mathematical Logics
Supervisor:	Doc. RNDr. Roman Barták Ph.D.
Supervisor's e-mail address:	roman.bartak@mff.cuni.cz

Abstract: The thesis studies an implementation of arc-consistency filtering algorithms for constraints defined in extension. We propose a new concept of binarization for decomposing high-arity ad-hoc constraints into networks of binary constraints. A theory proving correctness of the binarization is developed. We study the existing algorithms from the perspective of our binarization concept and propose possible binarization schemes for ad-hoc constraints defined in some of the common forms. In the thesis we also propose the filtering algorithms for the elementary constraints. A compound propagator then uses the elementary constraint filtering algorithms to propagate over the high-arity constraint. Finally, we experimentally evaluate the proposed approaches on constraints generated when solving the planning problems.

Keywords: constraint satisfaction, arc consistency, constraints defined in extension, binarization

Název práce:	Filtrační algoritmy pro tabulární podmínky
Autor:	Martin Molnár
Katedra:	Katedra teoretické informatiky a matematické logiky
Vedoucí diplomové práce:	Doc. RNDr. Roman Barták Ph.D.
e-mail vedoucího:	roman.bartak@mff.cuni.cz

Abstrakt: Předložená práce se zabývá implementací filtračních algoritmů hranové konzistence pro extenzivně definované podmínky. Zavádíme zde nový koncept binarizace pro rozkládání více-árních ad hoc podmínek na síť binárních podmínek. Je zde také rozpracována teorie pro dokázání správnosti této binarizace. V práci studujeme existující algoritmy z pohledu našeho konceptu binarizace a navrhujeme binarizace pro ad hoc podmínky definované vybranými běžnými způsoby. V práci také navrhujeme filtrační algoritmy pro dílčí podmínky. Složený propagátor pak používá tyto dílčí filtrační algoritmy pro propagaci přes více-ární podmínky. Konečně, navrhované postupy experimentálně ověřujeme na podmínkách generovaných plánovacími.

Klíčová slova: programování s omezujícími podmínkami, hranová konzistence, tabulární podmínky, binarizace

1 Introduction

Computers are very useful in solving many problems but it is necessary to describe the problems formally. One of the commonly used formalisms is constraint satisfaction. The specification of a *constraint satisfaction problem (CSP)* contains *variables* and *constraints*. Each variable has a finite set of possible values called the *initial domain*. The variables are bound by *constraints*, which allow/deny some combinations of values to be assigned to participating variables. Constraints can be defined using a mathematical formula or another similar way (*intensionally*). The alternative definition of the constraint is by an enumeration of the allowed (*compatible*) tuples of values. Such a constraint is typically defined in a table, hence the name *tabular constraints*. Other common terms for tabular constraints are "*ad hoc constraints*" and "*constraints defined in extension*". A *solution* of the CSP is such a variables' assignment that respects each constraint.

CSP solvers typically use backtracking search to find a solution. The time complexity of the search depends exponentially on the domain sizes, therefore any domain restrictions are desirable. One of the restriction techniques, the *arc consistency (AC)*, uses the following reasoning: If a value is pruned, all tuples containing the value are ignored. Tuples that are not ignored and are compatible (with respect to a given constraint) are called *feasible*. If the value is a part of a solution, it must be also part of some feasible tuple in all related constraints. Therefore the value, which is not part of any feasible tuple, can be pruned. The AC prunes all such values.

In this thesis we study arc consistency algorithms for tabular constraints. Our goal is to modify an existing or create a new AC algorithm for a specific class of problems. The constraints in these problems typically have higher arities. Our philosophy will be to exploit some particular properties of these problems by breaking the high-arity constraints into structures of constraints of small arities, typically binary constraints. The second step, exploiting some other characteristics of our problem class, will be the selection of pruning algorithms for these new constraints.

We will develop a necessary theory and present various algorithms for pruning binary constraints. Then we will examine different structures and combinations of the algorithms to find the most suited for our class of problems.

2 Constraint satisfaction

2.1 Preliminaries

As written in the introduction, the constraint satisfaction problem (CSP) consists of variables and constraints. Variables as nodes and constraints as hyper-edges together form a hyper-graph called a *constraint network*.

Each variable has its initial domain defined. The domains are finite sets of integers. Because the same number can be the value of many variables, we use the notation (variable, value) to refer to a concrete value.

When defining the constraints, the ordered set of variables bound by the constraint is called the constraint's *scope* and the size of the scope is called *arity*. The set of those tuples of values, for which the constraint holds, is called a *constraint domain*.

The CSP solver uses search techniques to find a solution. However, some ways of reasoning can be used before the search in order to deny some values that cannot be part of the solution. These are called *consistency* techniques and they can narrow the search space significantly. If the search is backtracking-like (by extending partial valuations) the consistency technique can be applied after each value selection and hence narrowing the search space even more. The consistency algorithm is *sound* if it removes only values that cannot appear in any solution. The algorithm is *complete* if it removes each value that can be excluded using the reasoning the consistency algorithm is based on.

Depending on the reasoning, many consistency levels can be defined. The more sophisticated consistency levels are also the more time-consuming. The most commonly used consistency level is *arc consistency*. We will define it for problems consisting of binary constraints:

Definition 2.1: Arc consistency

> Let C be a constraint of the CSP and let V_1 and V_2 be the variables bound by the constraint C. Let x_1 be a value in the domain of V_1.
>
> We say the value x_1 is *supported* in the constraint C iff there is a value x_2 in the domain of V_2 such that (x_1, x_2) conforms the constraint.
>
> We say that the arc (V_i, V_j) is *consistent* iff each value x_i in the domain of V_i is supported in the constraint binding V_i and V_j.
>
> We say that the CSP is *arc consistent* iff each arc is consistent.

We can analogically define *generalized arc consistency (GAC)* for constraints of higher arities. The only significant difference is the definition of support: while a supporting value is needed in the presented binary version, a supporting tuple is required by the GAC. Since there will be no risk of

misinterpretation, we will use the term arc consistency (*AC*) also for generalized arc consistency.

It is obvious that unsupported value cannot be part of any solution. The goal of the arc consistency algorithms is to remove (*prune*) all unsupported values; to restrict the domains of a CSP to an equivalent arc consistent form. The supported values are called *feasible*. A tuple is called *feasible* iff it is compatible (element of the constraint domain), and none of the values in the tuple is pruned.

There are two approaches to maintain arc consistency that differ in the specification of flaws making the problem inconsistent:

1. The *fine grained* algorithms consider the problem to be arc consistent, when there are no unsupported values.
2. The *coarse grained* algorithms are trying to reach the state in which all arcs are consistent.

In both cases, correcting a flaw can cause new flaws, so there is a queue of flaws and the algorithms are iteratively correcting flaws until the queue is empty.

2.2 Classical fine grained algorithms

The fine grained algorithms ensure that each value has a support. During the run of the algorithm, a value *(V, v)* may lost its supports in the variable *W*. Such value is called *pending (with respect to variable W)* until a new support for *v* is found. The pending values are stored in a queue. The fine grained AC algorithms pop a pending value from the queue and ensure that the pending value is still supported. The algorithms differ in a way the pending value is processed. For different algorithms, we will describe processing the pending value *(V, v)* with respect to variable *W*. First, we will present algorithm AC-4 introduced in [19].

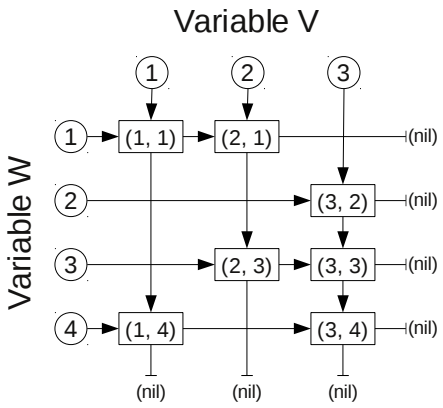


Figure 1: Constraint domain in sparse matrix

2.2.1 AC-4 algorithm

Figure 1 shows a constraint domain represented as a system of linked lists, each of which contains the tuples supporting one of the values. In fact, this structure (known as a *sparse matrix*) is not part of the AC-4 algorithm specification, on the other hand the structure is the result of a natural implementation of the algorithm. We are describing the AC-4 algorithm from this aspect because of a better understanding of later algorithms.

For now, we will consider only the linked lists composing the columns of the matrix. Each of these linked lists represents the set of tuples supporting one value of the variable *V*. Let us denote *slist[v]* the list for value *v*. Some of the tuples may contain pruned values, such tuples are not feasible. The algorithm needs to detect, whether at least one of the listed tuples is feasible. In order

to do that, the algorithm maintains the number of feasible tuples for each value. For value *v*, we will denote it *csup[v]*.

There are separate *slist* and *csup* structures for each arc (for each constraint in both directions). Note that when processing the pending value of variable *W* with respect to variable V (the other direction of the constraint binding *V* and *W*), lists in *slist* form the rows of the matrix in figure 1.

Let us consider processing a pending value *(V, v)* that has lost a support in the variable *W*. The number of supports the value has, stored in *csup[v]*, is decremented by 1. If *csup[v]* becomes zero, the value *(V, v)* has become unsupported and it is pruned. The tuples containing the pruned value are not feasible anymore. Let *(v, x)* be such a tuple in the constraint binding *V* with variable *X*. Then the value *(X, x)* becomes pending because it has lost support *v* in *V*. The algorithm needs to detect all such values *(X, x)*. In order to do that, the algorithm scans the lists *slist[v]* corresponding to all constraints binding the variable V. All feasible values from the lists *slist* are enqueued as pending values. On the other hand, if *csup[v]* is not zero after the decrementation, the value *v* is still supported and no pending values are generated. This way, the queue of pending values eventually becomes empty and the algorithm terminates.

We will denote the number of constraints *e* and the size of the largest domain *d*. There are *2e* arcs with their respective *csup* arrays. Each of the arrays has *O(d)* elements and each element has the initial value at most *d*. Therefore the sum of all elements in all *csup* arrays is $O(ed^2)$. This is an estimate of the number of decrementations the algorithm makes. All operations leading to one decrementation require a constant time. Thus the time complexity of the AC-4 algorithm is $O(ed^2)$.

2.2.2 AC-6 algorithm

The AC-4 algorithm kept the number of feasible supports (*csup* arrays) in order to determine, when the value becomes unsupported. The next classical algorithm, called AC-6 [8], will keep the smallest feasible support of each value instead. The smallest supports are stored in *ssup* array. (We will consider the supporting tuples instead of the supporting values for future generalization to non-binary constraints.) To describe the algorithm we need to define an operation of processing the pending value.

The value became pending because it has lost the smallest support. To process the pending value, the algorithm searches for the next smallest support. The searching starts at the lost support and the vertical links in figure 1 are used until a feasible support is found. In such a case, the found support is set to *ssup*. On the other hand, if the end of the linked list is reached, the value has no support and it is pruned. All smallest supports containing the pruned value become invalid. The values, which have been supported by the invalidated supports, become pending. The list of all smallest-support appearances of the given value is maintained by the algorithm in order to find the invalidated supports effectively. The lists are stored in a array called *lbss* and every time the algorithm alters the *ssup* array, the *lbss* array is modified accordingly. Again, the algorithm terminates when there is no

pending value.

Each tuple can be the smallest support at most once and the overhead of becoming the smallest support is constant. Thus the AC-6 algorithm requires $O(ed^2)$ time.

2.2.3 AC-7 algorithm

To improve the AC-6 algorithm we can exploit a symmetry called *constraint bidirectionality*: If *(V, v)* supports *(W, w)* then *(W, w)* supports *(V, v)*. When the smallest support for *(V, v)* is lost in the AC-6, the algorithm searches for the next smallest support using the vertical links in the *v*-th column of the sparse matrix. In fact, we do not need to search for the smallest support. Any support is sufficient to confirm that the value is feasible. The algorithm has the *lbss* available. The *lbss[v]* may contain some of the tuples of the *v*-th column of the sparse matrix. (In fact, *lbss[v]* contains those tuples that are being smallest supports for values of the variable *W*.) If *lbss[v]* is non-empty, its elements are actually supports of the value *v* because of the symmetry. The algorithm called AC-7 tries to find a support this way. On the other hand, if the list is empty, the AC-7 algorithm must search for a support in the AC-6 way.

The worst-case complexities are the same as complexities of the AC-6 algorithm, on the other hand, according to Bessière et. al [4] the AC-7 algorithm can save up to ed^2 constraint checks.

2.3 The AC-3 algorithm and it's variants

2.3.1 Algorithms AC-3 and AC-3.1

The previous algorithms treated each value individually. As opposed to those fine grained algorithms, we will now present the algorithms dealing with the whole constraints.

AC-3 algorithm

The oldest of the AC algorithms described in this thesis is called AC-3. The elementary step of the AC-3 algorithm [18] is a Revise procedure. It makes the given arc (V_i, V_j) consistent by excluding those values from the domain of V_i, which have no support in the domain of V_j. This exclusion can make arcs (V_k, V_i) for $k \neq j$ inconsistent. The AC-3 algorithm maintains a queue of possibly inconsistent arcs. The queue is initialized by a full enumeration of arcs. The arcs affected by the revision are then added into the queue. The next arc for revision is selected from the other end of the queue until the queue is empty.

The Revise procedure consumes $O(d^2)$ time and can be called at most $O(ed)$ times. Therefore, the time complexity of the AC-3 is $O(ed^3)$. Besides the constraint domain representation, the algorithm uses only $O(1)$ space.

AC-3.1 algorithm

The AC-3.1 algorithm [22] fixes the ineffectiveness of the AC-3 regarding the subsequent calls of the Revise procedure on the same arc. For each value, the algorithm remembers the smallest support. Since supports are only disappearing, when support t becomes invalid, search for a new support doesn't need to start from scratch but it can continue from the position of t. This guaranties that during all searches for support of x_i each value of V_j will be tested at most once. Hence, in the worst case, the time complexity is $O(ed^2)$ and the space complexity is $O(ed)$.

2.3.2 Propagators

The only difference between the AC-3 and AC-3.1 algorithms is the implementation of the Revise procedure. Disregarding the implementation, we will introduce an abstraction of the Revise procedure. The abstraction is called a *propagator*. The propagator is a procedure related to a constraint that prunes the unsupported values of the variables in the constraint's scope. The constraints defined by a mathematical formula may have their own propagators that are more efficient than the Revise propagator of the AC-3 or the AC-3.1. Note that such specialization is hardly conceivable in fine grained algorithms.

We will now describe the “interface” of the propagators: Each propagator has an ordered set of variables called *scope* (the size of the scope is propagator's *arity*). Just like in AC-3, when a domain of some variable in the scope changes[1], the propagator is enqueued for execution. By pruning other domains, the propagator can then reestablish consistency, which could have been corrupted by the domain change.

The propagator can have its *internal state*, which is passed to it as in/out argument. The solver engine executes the propagator with that input internal state, which was the output internal state of the previous call of the propagator. Assuming backtracking involved, the previous call is the one represented by the parent node in the backtrack tree. Therefore the internal state must be cloned in order to branch the search.

When examining the propagator call, we will denote $state_{IN}$ and $state_{OUT}$ the input and output internal state respectively and $dom_{IN}[i]$ and $dom_{OUT}[i]$ the input and output domain of the i-th variable in the propagator's scope.

As a special case of the propagator we define an *unidirectional binary propagator* as the binary propagator that prunes only the second variable of its scope. We will call *source* and *target variable* the first and the second variable respectively.

1 In general, the propagators can choose a condition on which they are executed. Besides the domain change, it can be the change of the minimum or the maximum of domain. Other common condition is when the domain becomes singleton. In this theses we will consider only the change of domain as the condition.

2.3.3 Propagator properties

We will now introduce several properties of the propagators.

Definition 2.2: Contraction

We say the propagator is *contracting* iff $\forall i$: $dom_{OUT}[i] \subseteq dom_{IN}[i]$.

Definition 2.3: Idempotency

Denote $dom_1[i]$ and $dom_2[i]$ the i-th input and output domain of the propagator's call respectively. Consider the subsequent call of the propagator on input domains $dom_2[i]$ and denote $dom_3[i]$ the i-th output domain.

We say the propagator is *idempotent* iff $\forall i$: $dom_2[i] = dom_3[i]$.

In other words the propagator is idempotent if the subsequent propagator call does not prune anything. There is no need to call the propagator again until some of the domains are pruned by other propagators. Note that we can make a non-idempotent propagator idempotent by iterating calls of the propagator until the domains reach a fixed point.

Recall that we call feasible such tuples that are compatible and the tuple's elements are in the current domains of their respective variables. Then the set of feasible tuples is $C \cap (dom[1] \times ... \times dom[a])$, where C is the constraint domain, a is the arity of the constraint and $dom[i]$ are the current domains of variables.

Definition 2.4: Soundness

We say the propagator is *sound* iff it never prunes a value from solution.

We say the propagator is *AC-sound* iff it never prunes a feasible tuple.

Soundness is more general than AC-soundness: every AC-sound propagator is sound.

Definition 2.5: AC-completeness

We say an idempotent propagator is *AC-complete* iff it prunes all unsupported values, as defined in the definition of AC (2.1).

A non-idempotent propagator is *AC-complete* iff the idempotent version of the propagator (iterating until a fixed point) is AC-complete.

Fact 2.6: Existence of the AC-sound and AC-complete propagator

Consider an a-ary constraint with the constraint domain C. Let us have a propagator that sets $dom_{OUT}[i]$ as i-th projection of the set $F \coloneqq C \cap (dom_{IN}[1] \times ... \times dom_{IN}[a])$. Then the propagator is contracting, idempotent, AC-sound and AC-complete. Moreover, any propagator that is idempotent, AC-sound and AC-complete is in fact equivalent to the above propagator.
(sketch of) proof: **contraction** is trivial consequence of properties of the projections.

AC-soundness: $F = C \cap (dom_{IN}[1] \times ... \times dom_{IN}[a]) \subseteq dom_{OUT}[1] \times ... \times dom_{OUT}[a]$ because of properties of the projections. The set F is the set of feasible tuples. Therefore no feasible tuple was pruned.

Idempotency: For any sets A, B, C: $C \cap A \subseteq B \Rightarrow C \cap A \subseteq C \cap B$ and $B \subseteq A \Rightarrow C \cap B \subseteq C \cap A$. Let $A \coloneqq dom_{IN}[1] \times ... \times dom_{IN}[a]$, $B \coloneqq dom_{OUT}[1] \times ... \times dom_{OUT}[a]$, $C \coloneqq C$. Then $B \subseteq A$ is equivalent with the contraction and we already used $C \cap A \subseteq B$ in the section about AC-soundness. Therefore both consequences $C \cap A \subseteq C \cap B$ and $C \cap B \subseteq C \cap A$ hold, hence $C \cap A = C \cap B$. The first call of the propagator computes projections of the set $C \cap A$ and the second call computes projections of the same set $C \cap B$. Therefore the returned output domains are the same.

AC-completeness: Let us consider a value v that is in the (respective) output domain. The the value is in the projection of F. In the set F, there is a tuple containing v that was a reason for including v into the projection. The tuple is feasible, thus can be used as a support. Hence all values of the output domains are supported.

Uniqueness of the propagator: We will denote P our projective propagator. Let us consider an idempotent propagator Q (idempotency is needed because we will rely on the idempotent version of the definition of AC-completeness). If the propagators P and Q are different, there is a situation, in which they return different output domains. Let us consider a value that is feasible according to Q and pruned by P. If the value is supported, then P is not AC-sound. On the other hand, if the value is not supported, then Q is not AC-complete. We will now focus on the reverse situation: the value is pruned by Q and feasible according to P. For the same reasons, either Q is not AC-sound or P is not AC-complete. We know that P is AC-sound and AC-complete. Therefore Q cannot be both AC-sound and AC-complete. ***QED***

Definition 2.7: Entailment

We say domains $dom_F[...]$ form a *fixed point* of an AC-complete propagator iff the propagator does no pruning when called on $dom_F[...]$ set as input domains (the call returns exactly $dom_F[...]$ as the output domains).

We say an AC-complete propagator is *entailed* with respect to current domains $dom_C[...]$ iff any subsets $dom_F[...] \subseteq dom_C[...]$ form a fixed point of the propagator.

A propagator that became entailed need not be called anymore because it will not prune any values (Of course, backtracking takes back the entailment, the propagator needs to be called again then). The propagator is entailed for domains $dom_C[...]$ iff
$C \cap (dom_C[1] \times ... \times dom_C[a]) = (dom_C[1] \times ... \times dom_C[a])$.

Having a constraint solver build on the AC-3 scheme, all involved propagators must be contracting and sound in order to find a solution. Moreover all propagators must be AC-complete for the solver to maintain arc consistency.

The last aspect of the propagator interface we will define is the returned result. The result of the propagator call is one of the following:

- Failure, meaning there is no solution in the current branch of the backtrack tree.
- Entailment, signaling that any subsequent call of the propagator cannot prune anything.
- Dirty, returned when some values were pruned
- Clean, telling that no values were prune

2.4 Remarks on time complexities

Note that any AC algorithm must at least read the constraint domains. Therefore $O(ed^2)$ is a theoretical lower bound for the worst-case time complexity the AC algorithm.

The time complexities stated up to this point measure the time needed to prune all the values. On the other hand no backtracking was assumed. We will call this kind of complexity a *single branch time complexity*, because it measures the time spent in one branch of the search tree. We can also measure the time requirements of a single propagator call in the coarse grained algorithms.

Other time requirements, such as the time spent during the whole search, are hard to estimate because many factors out of the scope of arc consistency are involved. Among others the shape of the search tree, the efficiency of a heuristics used to select values, the section of the search tree the solution is in, etc.

In this thesis, we will estimate both the single-propagator-call and the single-branch time complexities. All the time-complexity estimates will be the worst-case complexities.

3 A new concept of binarization

The philosophy of this thesis is to create the propagators for the complex constraints by the composition of the simpler propagators. For this we need an instrument for decomposing the complex constraint into a set of simple constraints. These elementary constraints will be typically binary. In this chapter we will build the necessary formal background.

3.1.1 Binarization

In the constraint satisfaction theory, replacing arbitrary-arity constraints by binary constraints is called binarization. There are two classical methods: the hidden variable method [21] and the dual graph transformation [14]. These methods were developed mainly for the theoretical reasons. The existence of such transformation allows to generalize the results of research on binary constraints to arbitrary-arity constraints, or to focus the research on binary constraints only with pleading that any problem can be described by binary constraints only. On the other hand, our concept does not try to dispose of the arbitrary-arity constraints. Our goal is to describe the internal structure of the constraints in order to implement an efficient filtering algorithm.

We will now describe the hidden variable binarization. A new variable is introduced for each constraint in the CSP. Let us focus on a single constraint C binding variables $V_1, V_2, \ldots, V_n$. The constraint C is replaced by the new variable W and n binary constraints binding W with each V_i. The initial domain of the variable W is the constraint domain of the constraint C. Therefore the compatible tuples of C are the values of the variable W. The constraint binding W and V_i is defined by the i-th value of the tuple: The tuple $(v_1, v_2, \ldots, v_n)$ is compatible with value v_i. The added constraints fully replace the original constraint. There is a bijective relation between the solutions of the original CSP and the solutions in the binarized CSP. Moreover, arc consistency is preserved, because the supporting tuple of the constraint domain, which is required by the GAC definition, is represented by the supporting value of the variable W that is required by the binary AC definition.

Our proposed binarization is a generalization of the hidden variable binarization in two aspects. Firstly, we will allow to add more than one variable per constraint. Secondly, the added variables will represent the sets of tuples instead of single tuples.

3.1.2 Preliminaries

We will consider a constraint that will be fixed for the rest of the chapter. We will call the constraint *superconstraint* and the constraints introduced by the binarization will be called *subconstraints*. The superconstraint is n-ary and we will refer to the bound variables as $V_1, V_2, \ldots, V_n$. These variables appear in the CSP definition therefore we will call them the *real variables*. The added variables will be called the *pseudovariables* and we will denote them

V_{n+1}, V_{n+2}, …, V_m, where m is the number of all variables (both real and pseudovariables).

Denoting $dom(V_i)$ the initial domain of the real variable V_i, we will call *universe* the Cartesian product $dom(V_1)\times dom(V_2)\times...\times dom(V_n)$. Any considerable tuple, either compatible or incompatible, is an element of the universe. We will use the term *t-set* (derived from "tuple set") for a subset of the universe. The constraint domain (denoted C), is an example of t-set. Any set of t-sets will be called *t-system*. We will use Greek alphabet letters to denote value tuples, lower case Latin letters for t-sets and upper-case letters for t-systems.

When decomposing the constraint we will constitute *pseudovalues* of the pseudovariables. A pseudovalue can be any t-set, although when developing the theory we will see that only some t-sets are useful. The real values will also be represented by t-sets. For example value (V_2, v_2) is represented by the t-set $dom(V_1)\times\{v_2\}\times dom(V_3)\times...\times dom(V_n)$. Note that this leads to the notation $\tau \in (V_2, v_2)$ (the tuple is an element of the t-set) instead of common $(V_2, v_2) \in \tau$ (the tuple contains v_2).

We will say the value x *supports* the value y iff x and y intersect. Let us consider an example in figure 2. The rectangle "a" represents a pseudovalue with t-set $2..6\times 2..3$. Value $(V, 3)$, as defined above, is represented by t-set $\{3\}\times dom(W)=\{3\}\times 1..10$. These t-sets intersect, therefore the values support each other. The value $(V, 3)$ also supports the rectangle "c" but does not support the rectangle "b".

Real variables and pseudovariables are t-systems because they are sets of real values and pseudovalues that are represented by t-sets. The left part of figure 2 depicts a constraint domain, which is a set of tuples (t-set). On the other hand, the right part depicts a pseudovariable "R" that is a t-system of 3 t-sets (rectangles "a", "b" and "c"). We will need to make statements such as that the pseudovariable "covers" the constraint domain. In order to do that, we introduce an operation of *grounding* to convert the t-system to a t-set. The grounded t-set of the t-system S is the union of t-sets the t-system S consists of, we use the notation $\cup S$ that is common in the set theory. Note that we can use the grounding operation on any t-system, not only on pseudovariables. For example consider a current domain of the real variable containing rectangles "a" and "b". The grounding of the current domain (union of rectangles "a" and "b") represents the set of feasible tuples after pruning the pseudovalue of rectangle "c".

We will also need an operation generalizing the Cartesian product for pseudovariables. Let V_i and V_j be two variables (either real or pseudo). Let S_i be a subset of V_i and S_j subset of V_j. Then we define the product of sets S_i and S_j as $(\cup S_i)\cap(\cup S_j)$. Note that this definition is consistent with the Cartesian product of the real variables: Let us consider figure 2 again. Value v of the variable V is represented by t-set $\{v\}\times dom(W)$. Then the set S_i of values $(V, 3)$, $(V, 4)$ and $(V, 5)$ is the t-system $\{\{3\}\times dom(W), \{4\}\times dom(W), \{5\}\times dom(W)\}$. Therefore $\cup S_i = \{3, 4, 5\}\times dom(W)$. Analogically subset S_j of W consisting of values $3..7$ has grounding $\cup S_j = dom(V)\times 3..7$. Then
$(\cup S_i)\cap(\cup S_j) = (3..5\times dom(W))\cap(dom(V)\times 3..7) = 3..5\times 3..7$

Thus our definition of the product is equivalent with the Cartesian product when the Cartesian product is applicable.

In the rest of this thesis, we will use the term variable for both real variables and pseudovariables and term value for both real values and pseudovalues.

3.1.3 Motivation

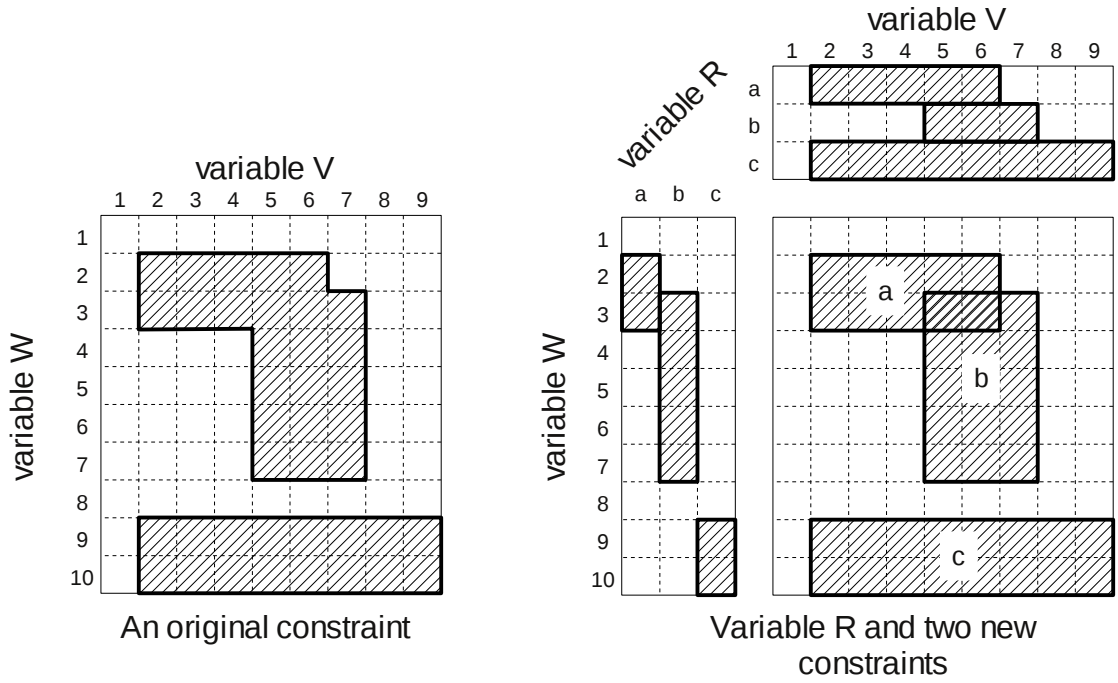


Figure 2: an example of introducing a pseudovariable

As a motivational example, consider a constraint in the figure 2. We will denote *a* a pseudovalue corresponding to the proposition "value of V is in 2..6 and value of W is 2 or 3". The t-set of the pseudovalue "*a*" is *2..6×2..3*. *a* with analogically defined *b* and c will form the domain of pseudovariable R. The reason of introducing the pseudovariable is this: The naturally defined constraints, one binding R and V and another binding R and W (see figure 2), can together replace the original constraint between V and W. This is because the grounding of the pseudovariable R is equal to the constraint domain. Therefore, we can extend each solution *(V:=v, W:=w)* with an appropriate value for variable R. On the other hand, if we prune all values from the domain of R then there is no solution.

Let us focus on the mentioned fact that the grounding of the pseudovariable R is equal to the constraint domain. We will use such facts, in a more general form, in our theory. They will form a condition for soundness of our binarization. We have also mentioned the "naturally defined constraints" in the previous paragraph. Such naturally defined constraints exist only for particular

pseudovariables. The existence of the natural definition of the constraints will be required for AC-completeness of the propagation (on the binarized network of subconstraints).

Although handling two constraints instead of one does not seem better, we will show it is reasonable because the new constraints have a particular structure. We can also use the same principle to transform one *a*-ary constraint consisting of hyper-rectangles into *a* binary constraints.

Another motivational example is a constraint in the form known in SICStus Prolog as “case”. Figure 3 shows the constraint binding variables U, V and W with this semantics: A tuple is compatible if there is an oriented path in the directed acyclic graph such that all restrictions denoted on path's edges and leafs hold. The form of the case constraint requires the graph to consist of layers and all restrictions for a given variable are required to be in one layer. Suppose we have a solution (u, v, w). The solution uses a path and therefore it uses one of the edges in the first layer. Let us denote this edge by pseudovariable I_U. The pseudovariable I_U consists of two pseudovalues: one corresponding to the edge “U in 1..2” and the second representing “U in 4..7”. Analogically we will define pseudovariable I_V consisting of 3 pseudovalues representing the edges in the second layer. Finally I_W is defined for the layer of leafs. This way we can internally implement the case constraint as a system of five binary constraints: U-I_U, V-I_V, W-I_W, I_U-I_V and I_V-I_W.

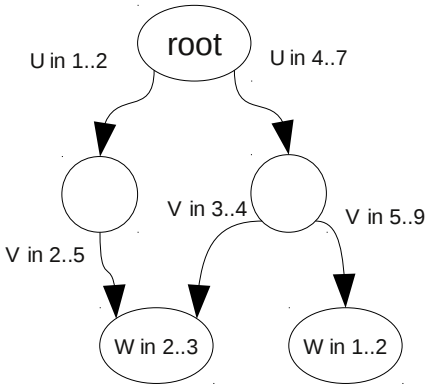


Figure 3: an example of constraint in the case format

3.1.4 The concept

Nothing we mentioned up to this point is in conflict with the fine grained approach. On the other hand, for efficiency reasons we will use the skeleton of the AC-3 algorithm to develop our concept. As we have outlined in examples, the pseudovariables can be pruned just like variables (hence the name). Recall that the elementary constraints binding the pseudovariables are called *subconstraints*. As an analogy of the constraint network we define the *subconstraint network:*

Definition 3.1: Subconstraint network

> The *subconstraint network* is a graph having real variables and pseudovariables as nodes and subconstraints as edges of the graph

The scheme of the subconstraint network of the above examples is shown in figure 4. The circles and the squares represent the variables (both real and pseudo) and the connections between them represent the subconstraints. The difference between circles and squares will be explained later (for now, the real variables are typically are circles and the pseudovariables are squares).

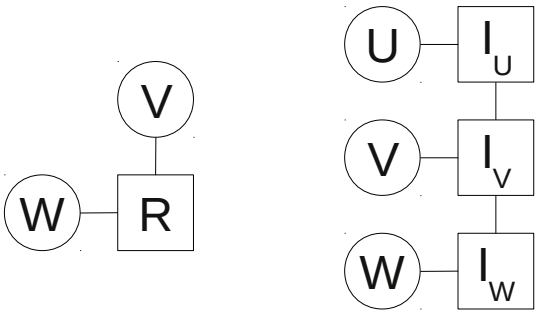


Figure 4: schemes of subconstraint networks. The left scheme represents the rectangles example (figure 2) and the right scheme represents the "case" example (figure 3)

The significant fact is that the whole subconstraint network must be acyclic. We also require that the real variables are leafs and the pseudovariables are inner nodes of the subconstraint network. That ensures that propagating each subconstraint at most once in each of its directions during the superconstraint propagation is sufficient to fulfill the AC-soundness and the AC-completeness. This proposition will be proven in theorems 3.6 and 3.7. This fact was discovered by Dechter and Pearl [13] for the tree-shaped constraint networks. We will use it as an inspiration for the subconstraint networks and we will define the process of the superconstraint propagation accordingly. The process consists of the upward phase and the downward phase. In the upward phase, we propagate arcs from the leafs of the network to the root. The downward phase propagates in the reverse direction.

The propagation definition will expect that indices i of variables V_i define the ordering of the variables increasingly from the leafs to the root. If the ordering is unsuitable, the variables can be renamed in order to fulfill this property. We will show the process of renaming after the formal definition of the desirable ordering:

Definition 3.2: Topological ordering

Let us consider the subconstraint network as a rooted tree having the pseudovariable V_m as the root. We define function $UP: \{1, \ldots, m-1\} \longrightarrow \{1, \ldots, m\}$ such that $UP(j)=i$ iff V_i is the parent node of V_j in the rooted tree ($UP(m)$ is not defined because V_m is the root).

We say the constraint network is *topologically ordered* iff $UP(i)>i$ for each i.

To find the proper indices, we will pick a root and set it's index to m. Then we will set the orientation of all edges of the tree in the direction to the root. We can perform the topological sorting because the tree is an acyclic graph. Then we will set the indices of the variables respectively to the topologically sorted ordering. From now we will suppose that $V_1, \ldots, V_m$ are named in a way forming the topologically ordered constraint network.

Definition 3.3: Proper superconstraint propagation

Let us consider the topologically ordered subconstraint network.

We define the *upward domain* U_i of the variable V_i this way: If V_i is real variable (leaf, $i \leq n$), U_i is the i-th input domain of the superconstraint propagator call. Otherwise U_i is the

domain of V_i after applying the propagation from the upward domains U_j of each child V_j of V_i.

We will start the definition of the *downward domain* (denote D_i) at root: $D_m \stackrel{\text{def}}{=} U_m$. For other node i the downward domain is defined as U_i after applying the propagation from the parent node's downward domain $D_{UP(i)}$.

At the end of the process, downward domains $D_1, \ldots, D_n$ of the real variables are returned as the output domains.

The defined procedure is shown in the following code:

```
procedure Proper_superconstraint_propagation(in out domains : array[1..n] of sets)
  for i in 1..n
    U[i] := domains[i]
  end for
  for i in 1..m-1
    propagate(U[i], U[UP(i)])
  end for
  D[m] := U[m]
  for i in m-1..1   /* m-1..1 represents the decreasing sequence {m-1, m-2, ..., 1} */
    D[i] := U[i]
    propagate(D[UP(i)], D[i])
  end for
  for i in 1..n
    domains[i] := D[i]
  end for
end procedure
```

Rectangularity

In the previous section, we focused on the acyclic shape of the subconstraint network. To define another requirement on the subconstraint network, let us study the above motivational examples. In the example in figure 2, all pseudovalues' t-sets were Cartesian products of subsets of the variable domains. This is not the case in the second example: There are three paths in the case graph, from the left: one representing the t-set (1..2)×(2..5)×(2..3), other standing for (4..7)×(3..4)×(2..3) and the last one corresponding to (4..7)×(5..9)×(1..2). The definition of the “case” form says that the tuple is compatible iff it is represented by one of the paths. Therefore, the constraint domain of this constraint is the union of these three t-sets. Consider the edge labeled “U in 4..7” and its respective pseudovalue. In order to recognize the t-set of this pseudovalue, imagine that all other pseudovalues of the pseudovariable are pruned. Tuples that are feasible in this situation form the t-set of the pseudovalue “U in 4..7”. The t-set is
(4..7)×(3..4)×(2..3)∪(4..7)×(5..9)×(1..2) = (4..7)×((3..4)×(2..3)∪(5..9)×(1..2))
This set is not a Cartesian product of three sets. Thus, pseudovalues can represent sets that are more complex than the Cartesian products of subsets of the variable domains. On the other hand, we need the pseudovalues to be rectangular locally: The pseudovalue must be the product of its projections to all neighboring (pseudo)variables. We will define this property in a form most suitable for future usage in proofs.

Definition 3.4: Rectangularity

We say the t-set x is *rectangular* with respect to t-systems A and B iff
$\forall\, a \in A, b \in B: x \cap a \neq \emptyset \wedge x \cap b \neq \emptyset \Rightarrow a \cap b \subseteq x$
We say a pseudovariable V is *rectangular* iff each pseudovalue v of V is rectangular with respect to any pair of pseudovariables neighboring with V in the subconstraint network.
The subconstraint network is *rectangular* iff all its pseudovariables are rectangular.

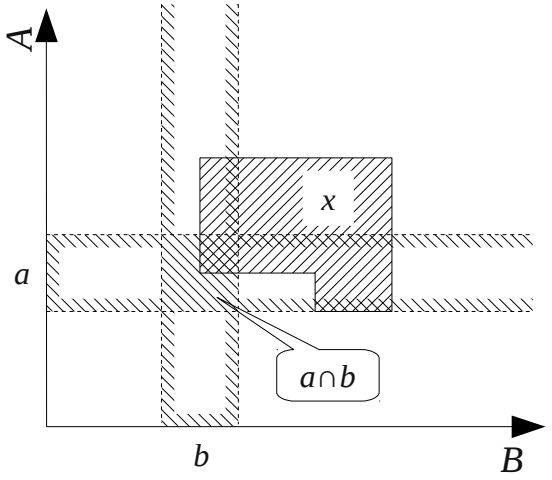


Figure 5: An example of t-set x that is not rectangular with respect to t-systems A and B

Figure 5 shows a situation forbidden by the definition of rectangularity. $a \cap b$ it is not a subset of x, although a and x and also b and x intersect. On the other hand, the rectangles in figure 2 are rectangular.

For a better understanding of this concept, let us recall the original requirement: "The pseudovalue must be the product of its projections to all neighboring variables". When this holds, the geometrical boundary of the pseudovalue is composed of boundaries of values of the neighboring variables. Boundaries of the real values are linear and orthogonal to axes. Combination of such boundaries forms a geometrical rectangle. On the other hand, the pseudovalues are arbitrary t-sets and the resulting shapes might be rather distant from geometrical view of rectangles.

Expressiveness equivalence

We will now define another property we need to preserve when defining the pseudovariables. We need the pseudovariables to express the constraint domain of the constraint we are trying to model. For this to assert we require that tuple τ is in the constraint domain if and only if for each pseudovariable there exists a pseudovalue comprising τ. Formally $\tau \in C \Leftrightarrow \forall V_i \exists v_i \in V_i : \tau \in v_i$ and equivalently $\tau \in C \Leftrightarrow \tau \in \bigcap_i (\bigcup V_i)$.

Definition 3.5: Expressiveness equivalence

We say that the system of pseudovariables $\{V_i\}_i$ is equivalent to the constraint domain C iff

$C = \bigcap_i(\bigcup V_i)$

3.1.5 Properties of the superconstraint propagation

Theorem 3.6: Soundness of the superconstraint propagation

Consider a proper superconstraint propagator of a subconstraint network equivalent to the constraint domain *C*. If all propagators of the subconstraints are sound then the superconstraint propagation is sound.

proof: For contradiction, suppose the superconstraint propagation is not sound. There is a tuple τ in the constraint domain *C* and in the input domains $U_1, ..., U_n$ that have been pruned. The superconstraint propagator generates the sequence of domains $U_1, U_2, \ldots, U_m=D_m, D_{m-1}, ..., D_1$ (see definition 3.3). Consider groundings of these t-systems. We know that τ was feasible before the propagation, therefore $\tau \in \bigcup U_i$ for all $i \in \{1, \ldots, n\}$ (τ is covered by each input domain). On the other hand, τ is not feasible after the propagation, therefore $\tau \notin \bigcup D_i$ for some $i \in \{1, \ldots, n\}$ (some of the output domains exclude τ). There is a domain (either D_i or U_i) that is the first excluding τ in the sequence.

First consider that the first excluding domain is upward (U_i). Let V_j be the source variable of the propagation that excluded τ from U_i (V_j is a child of V_i). U_j appeared in the sequence before U_i, therefore $\tau \in \bigcup U_j$. Thus there is a value $v_j \in U_j$ such that $\tau \in v_j$. Because of the expressive equivalence with C, there is also value v_i of the variable V_i that contains τ. The problem is that the value v_i was pruned from U_i. The value v_j supports value v_i because their t-sets intersect (τ is the common element). Therefore v_i was pruned incorrectly, the propagator used on arc (V_j, V_i) was not sound.

The second alternative is that the first domain in the sequence was downward instead of upward. Analogically the propagator that pruned a value containing τ was not sound. Either way we have a contradiction with the assumption that all propagators of the subconstraints were sound. ***QED***

Theorem 3.7: Completeness of superconstraint propagation

Consider a proper superconstraint propagator of the subconstraint network having all variables rectangular. Let the subconstraint network be equivalent with the constraint domain *C*. If all propagators of the subconstraints are AC-complete then the superconstraint propagation is AC-complete.

proof: For contradiction, suppose that the propagation left an unsupported value in some output domain. Without loss of generality, let the unsupported value be v_1 in the output domain D_1. Because the propagator of the arc from the parent node $D_{UP(1)}$ to D_1 was AC-complete, there is a value $v_{UP(1)}$ in $D_{UP(1)}$ supporting v_1. The value $v_{UP(1)}$ has support $v_{UP(UP(1))}$ in $D_{UP(UP(1))}$ for the same reason and this way we can find values in the downward domains

indirectly supporting v_1 in all variables on the path from V_1 to the root. Continuing in the counter-propagation direction into the upward phase, we can find values $v_i \in U_i$ for the rest of the variables. We will define a tuple τ from values $(v_1, ..., v_n)$ of the real variables. Values $v_1, ..., v_n$ were in the respective input domains U_i. Therefore if τ is in the constraint domain, then τ is a supporting tuple of the value v_1, and v_1 need not have been pruned by an AC-complete propagator. The only fact left to prove is that $\tau \in C$.

We have the system of values $\{v_i\}_i$ such that v_i supports v_j for all the arcs (V_i, V_j) in the subconstraint network. Supporting means that t-sets of v_i and v_j have a common tuple. Using the rectangularity we will show that the common tuple is identical for all the arcs. In fact, we will show that τ is such a common tuple, in other words $\tau \in v_i$ for all $i \in \{1, \ldots, m\}$. This holds for the leafs (real variables) because that is how we defined τ. As an induction step, consider v_i having supports v_j and v_k in two children[2] nodes of V_i. Applying the induction assumption on the children, we get $\tau \in v_j$ and $\tau \in v_k$. By the definition of support: $v_i \cap v_j \neq \emptyset$ and $v_i \cap v_k \neq \emptyset$. The rectangularity of V_i then states that $v_j \cap v_k \subseteq v_i$, therefore $\tau \in v_i$. By applying the induction up to the root, we get $\tau \in v_i$, thus $\tau \in \bigcup V_i$ for each node V_i. Now we have $\tau \in \bigcap_i (\bigcup V_i)$ and because the subconstraint network is equivalent to the constraint domain C, $\tau \in C$. Therefore we conclude that τ is compatible, feasible and therefore it supports the value v_1. Leaving v_1 in the output domain D_1 does not violate AC-completeness of the superconstraint propagator. *QED*

We will now state some trivial facts without proofs.

Fact 3.8: Entailment of superconstraint propagation

If all the subconstraint propagators are entailed then the superconstraint propagator is entailed.
If all the downward domains $D_{n+1}, \ldots, D_m$ of the pseudovariables are singletons then the superconstraint propagator is entailed.

Fact 3.9: Failure of superconstraint propagation

If any of the subconstraint propagators returns failure then the superconstraint propagator also fails.
If any of the downward domains $D_{n+1}, \ldots, D_m$ of the pseudovariables is empty then the superconstraint propagator fails.

3.1.6 Advantages of using pseudovariables

This binarization approach allows us to study AC algorithms from yet another perspective. Many

2 We assume each node has at least two children. There is no sense in creating other subconstraint networks because if we left out the node we would have a network with more efficient propagation (due to less subconstraint propagation calls). The theorem holds for such subconstraint networks too, but require a technically more complex proof.

algorithms have the structure of pseudovariables although in an implicit way. We will try to reveal these structures in the rest of the thesis.

If we found a pseudovariables structure in an algorithm, the algorithm typically prunes the pseudovariables in the fine grained way. Using coarse grained pruning (propagation) on subconstraints may have effect of multiplicative constant on time requirements. We will present the propagators extensively using the bit parallelism in chapter 5.

Some domain representations have their preferable ways of processing. Consider a domain of variable V in our “case” example (figure 3). Obviously, we need to test whether it intersects with the intervals on edges. The queries for individual intervals are in total less effective than one complex query listing all the intersected edges. The returned list is nearly the domain of variable I_V.

We can also provide several propagation algorithms for pruning the binary constraints that have their particular strengths and weaknesses. Depending on the characteristics of the problem, we can choose which of the propagators to use or let the solver to choose the algorithm depending on the input data.

4 Existing domain representations and algorithms

4.1 Set and domain representations

In this chapter we will describe data structures that can be used for storing sets and domains.

4.1.1 Bit array

The simplest way to store domains and sets is a bit array. The bit indexed by a value is set to 1 if the value is an element of the represented set. Note that we can represent only finite domains, we will without loss of generality assume that we represent at most *d* values and they are all from *{0, …, d-1}*. We will call *d* the *extent* of the represented set.

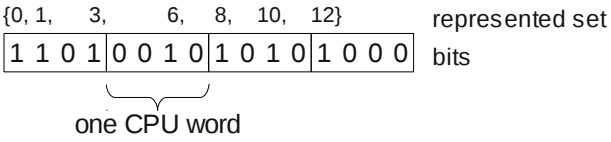


Figure 6: a bit array example

For efficiency, we will store and process bits grouped into CPU words. The algorithms for operations with this data representation are trivial. Algorithms for inserting, removing and testing an element are using bit shift and bitwise operation and run in a constant time. To produce union or intersection of two sets the algorithm uses bitwise operations that require *O(d)* time. The operation of checking non-emptiness of the set searches for non-zero word and require *O(d)* time. As an example, code 4.1 shows the operation of inserting an element (The code is written in C language because it is too low-level to be shown in a pseudo-code).

Using this data structure for maintaining arc consistency was studied by Lecoutre and Vion [17], where the detailed description of the algorithms can be found. The main benefit of using the bit array, called the bit parallelism, is that the CPU processes many values at once. Although the effect on the time requirements is only multiplicative, it is significant.

Code 4.1: Inserting an element to the bit array

```
void insert_element(unsigned int* bit_array_data, int elem) {
  int position =       elem / WORD_SIZE;
  int mask     = 1 << (elem % WORD_SIZE);
  bit_array_data[position] |= mask;
}
```

We will also define an auxiliary data structure called bit-array *chunk*. The chunk consists of a position and a mask and can be viewed as a single CPU word extracted from the bit array. The extracted word represents the mask and the position is the position of the extracted word. The chunk

can represent sets having elements so close that the elements fall into single CPU word of the bit-array representation. For example any singleton set *{elem}* can be represented by a chunk with position *elem div word_size* and the mask is *(elem mod word_size)*-th bit. In fact, the above algorithms for setting, clearing and testing an element create the chunk implicitly, as code 4.1 shows.

4.1.2 List of intervals

Another way of the set representation is to find maximal (with respect to inclusion) intervals and store them sorted in an array or a linked list. We can also represent some infinite domains using symbols *infimum* and *supremum* for negative and positive infinity. Denote *i* the number of intervals the set consists of. The set operations mentioned in chapter about bit arrays have their respective versions and all of them run in time *O(i)*. For adding, removing and testing a single element, the list-of-intervals algorithms are worse compared to the algorithms for the bit array (*O(i)* vs. *O(1)*). For other operations, these two data structures have the same worst-case time complexity (*O(i)* and *O(d)*) because *i* is *O(d)*. To present these algorithms we will introduce a data structure that is equivalent to the list of intervals but provides simpler formulation of the algorithms.

4.1.3 Cumulative points

The *cumulative points* data structure also represents the set as the list of intervals. It can be seen in the algorithm proposed by Beldiceanu and Carlsson in [3]. The cumulative-points structure consists of a number called *bias* (typically 1) and a set of *points*. The point consists of *position* and *increment*; we use the notation position$increment. An interval *min..max* is then represented by points *min$(+1)* and *(max+1)$(−1)*.

For example the set *A={2, 3, 4, 5, 9}*, in the list-of-intervals form *2..5∪9..9*, is represented by points *2$(+1), 6$(−1), 9$(+1), 10$(−1)*. The representation is shown in figure 7, the other depicted set labeled “B” is *3..5*. Each arrow represents a point, upward with the positive increment, downward with the negative one. By merging lists of points representing the sets A and B, we get list *2$(+1), 3$(+1), 6$(−1), 6$(−1), 9$(+1), 10$(−1)*. Points with the same position can be grouped into one point, setting its increment to the sum of increments of the points being grouped. This way, *6$(−1), 6$(−1)* can be replaced by *6$(−2)*.

We say the list of points is *consolidated* iff the increments of the points in the list are alternating *$(+1)* and *$(−1)*. In figure 7, lists “A” and “B” are consolidated, while “merge(A, B)” is not. Consolidated lists can be straightforwardly transformed into the list-of-intervals representation. And the transformation from the list of intervals produces consolidated list of points.

Note that both A∩B and A∪B can be extracted from the merge depicted in figure 7. The data structure field “*bias*” is used to distinguish between the intersection (*bias=2*) and the union (*bias=1*). Then the intersection or the union is defined by sections, where the horizontal line is at

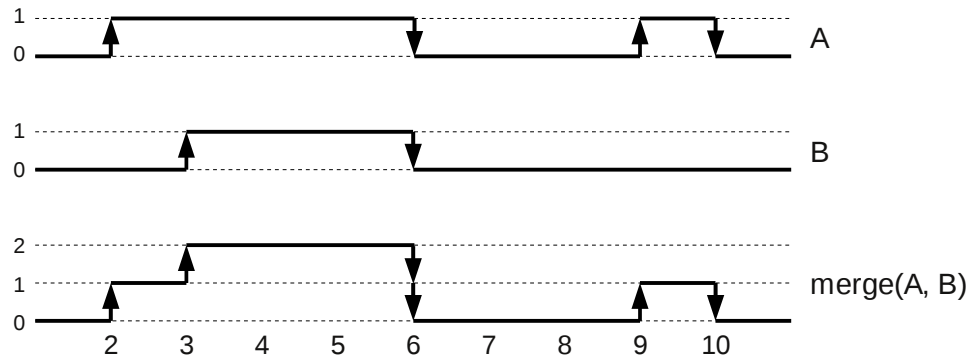


Figure 7: two cumulative sets and their merge

the level of bias or higher.

For a given position *x*, we need to determine the level at which the horizontal line is. If the cumulative points data structure was consolidated, the level would be 0 or 1 depending only on the last point preceding the position of *x*. In general, we need to accumulate the sum of all increments of points prior to *x*. Then the value *x* is element of the *set* iff

$$set.bias \le \sum\{increment \mid (position \$ increment) \in set.points \wedge position \le x\} \qquad (4.2)$$

Unless said otherwise, we store or enumerate the set of points sorted by the position. For technical reasons we introduce a new symbol *sentinel* which is greater than the supremum and put sentinel$0 at the end of each list of points. We will use three elementary operations and combine them to create intersections and unions of the sets in this representation.

The `merge` procedure merges points of two lists so that the result is their properly sorted join. Then we use the `group` procedure to replace many points of the same position by one point with the proper increment. The `consolidate` procedure horizontally cuts the image in figure 7 at the level of bias and returns arrows crossing the cut. More precisely, the procedure processes the points in order of increasing position and maintains the cumulative sum of increments. If the sum is below the bias and the point *position$increment* rises the sum to the bias or higher, then the point *position$(+1)* is sent to output. Analogically point *position$(−1)* is generated when the sum declines below the bias. The result of the `consolidate` procedure is consolidated and can be straightforwardly translated into the list-of-intervals representation.

Code 4.3: Set operations in cumulative points representation

```
procedure merge(list1, list2)
  result := []          /* an empty list */
  do
    if (list1.First() < list2.First()) then
      point := enum1.First()
      list1.DeleteFirst()
    else
      point := enum2.First()
      list2.DeleteFirst()
    end if
    result.append( point )
  while point.position < sentinel
  return result
end procedure

procedure group(inputList)
  result := []
  kept := nil
  for point in inputList
    if kept.position == point.position then
      kept.increment += point.increment
    else
      result.append( kept )
      kept := point
    end if
  end for
  result.append( point(sentinel, 0) )
  result.DeleteFirst()      /* the first element of result is nil */
  return result
end procedure

procedure consolidate(inputList, bias)
  result := []
  level := 0
  for point in inputList
    newLevel = level + point.increment
    if level < bias and newLevel >= bias then
      result.append( point(point.position, +1) )
    else if level >= bias and newLevel < bias then
      result.append( point(point.position, -1) )
    end if
    level := newLevel
  end for
  result.append( point(sentinel, 0) )
  return result
end procedure
```

When intersecting or unioning more than two sets, we merge them all together and apply the consolidation only once. Let us denote *n* the number of sets to be intersected/unioned. To compute the intersection the bias is set to *n*. In the case of union, the bias remains 1.

4.1.4 Sparse sets

When backtrack occurs, the retrieval of old search state is needed. The previously mentioned set data structures needed to be copied when entering a search tree branch and the copy was restored when backtracking. A data structure called *sparse set* introduced by Briggs and Torczon in [7] does

not have such drawback and it is restorable in a constant time.

The data structure consists of

- the size of the represented set
- the dense array containing elements of the set in *dense[0], …, dense[size−1]*. The rest of the array is undefined.
- and the sparse array containing indices into the dense array, maintaining *dense[sparse[e]]=e* for all elements. The value of *sparse[e]* is undefined if *e* is not an element of the set.

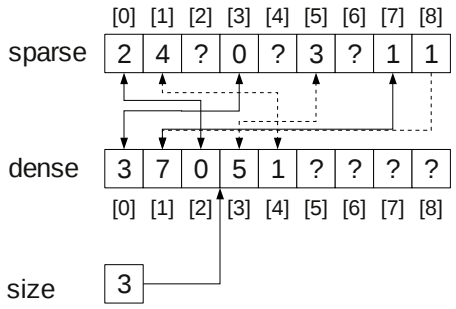


Figure 8: sparse set representing set {0, 3, 7}. Numbers 1, 5 and 8 were elements in backtracked parts of the search tree. The continuous lines represent valid field values. The dashed lines represent field values that are undefined. Fields "?" were never written to.

The following code shows adding a new element into the sparse set

Code 4.4: inserting the element into the sparse set.

```
procedure insert(element)
  size := size + 1
  dense[size-1] := element
  sparse[element] := size-1
end procedure
```

The elements in the dense array are ordered by the time of insertion. Setting the size can vanish elements added after some time moment. This operation allows us to restore the structure when backtrack occurs. Only information needed is the size of the structure in the moment of entering the search tree branch.

To test whether *elem* is an element of the set, the algorithm needs to follow the data structure definition precisely. The reason is that the undefined items actually might be meaningful because they may have been set in backtracked branches. The algorithm first retrieves a possible index to the dense array from *sparse[elem]*, denoting it *index*. Then if *index*≥*size* the data structure was restored upon backtracking and the restoration excluded the element from the set. Number 1 in figure 8 represents such a case. In case *dense[index]*≠*elem*, the elem was inserted in branch that was later backtracked and other element took its place in the current branch of the search tree. For example, number 8 was inserted into the set, later it was backtracked and replaced by number 7. In both cases the *elem* is not an element of the set, otherwise it is.

Code 4.5: testing whether the argument is an element of the sparse set.

```
procedure test(element)
  index := sparse[element]
  if index < size and dense[index] = elem then
    return true
  else
    return false
  end if
end procedure
```

A technical detail is that the data structure must be used to represent the set that is growing during filtering. Therefore we must store the set of pruned values instead of the set of feasible values.

Cheng and Yap prefer in [12] the sparse set over the bit array. They argue that the backtrack cost *O(1)* vs. *O(d)* might be significant for large sets. In the class of problems this thesis focuses on, the complexity of the problem is imposed by constraint arity rather than by the size of the domains. Besides, Cheng and Yap were using algorithms that could not profit from the bit parallelism. In our implementations, the possible benefit of using the sparse sets is balanced by loosing the benefits of bit parallelism. Therefore we use bit arrays in our experiments. Note that the difference in cost of backtrack is irrelevant in complexity estimates since it is majorized by the complexity of the propagation itself.

4.1.5 Choosing appropriate representation for domain

The time complexities of the bit array and the sparse set algorithms depend on the size of domain representation (denoted by d). On the other hand, the complexities of the list of intervals and the cumulative points are determined by the count of intervals the domain consist of (i). In the worst case $i=O(d)$, under some circumstances this is the expected case. Let us consider a variable having statistically independent values with uniform probability p of being feasible. The lengths of the intervals then obey geometrical distribution and so do the lengths of the gaps between the intervals. The expected value of geometrical distribution is constant; we will denote l the expected length of the interval and the following gap together. We can expect $i=d/l=O(d)$ intervals to be present in the domain of size d. In this case, the bit array and the list of intervals have the same asymptotic complexities of algorithms. We should prefer the bit array in such a situation. We can expect a smaller multiplicative constant because the bit array representation is more processor-friendly. The bit array is processing the whole CPU word in one instruction and does not use so many conditional jumps as cumulative points[3].

We will call *discrete* those variables, which we find to be more suited for representation by the bit array then by the list of intervals. We will use the word “discrete” as a philosophical not a mathematical term, including not only the variables that fulfill the strict specification in the above

3 Conditional jumps are known to be ineffective because after-jump instructions must be prepared for execution after it is known whether condition holds or not. It makes the processor instruction queue dormant.

example but also the variables meeting it only partly. The variables, which are better represented by the list of intervals, will be called *continuous*. Those variables have typically strongly dependent neighboring values. In the subconstraint network schemes such as in figure 4 we will use squares for the discrete variables and circles for the continuous variables.

A typical example of the discrete variable is *R* in the rectangles example on page 17. Any permutation can be used to represent rectangles *"a"*, *"b"* and *"c"* by numbers 1, 2 and 3. Therefore, we have no reason to claim that the neighboring values 1 and 2 are more dependent than the distant values 1 and 3.

The list of intervals is based on the same principle as the run-length encoding compression. So let us consider data compression as an analogy. Compression exploits some structures (patterns, symmetries, relations) in the plain data and produces smaller compressed data. The compressed data does not contain such structures anymore, therefore they cannot be compressed more. Analogically, the constraint binding a pseudo and a real variable typically exploits continuity of the real variable, leaving the lack of dependencies in the pseudovariable. Therefore, pseudovariables are typically discrete.

When deciding whether a real variable should be treated as discrete or continuous we might also take solver's interface into consideration. For example, in SICStus prolog, all domains are passed and pruning results are expected in the list-of-intervals form. The same is true for the GECODE library unless the user extends the library by a new variables' implementation. We should handle the real variables as continuous in such cases.

On page 14 we mentioned the theoretical lower bound for the worst-case time complexity of any AC algorithm. The stated complexity depends on *d*. Using the list-of-intervals representation leads to such asymptotic complexity assessments that depend on *i* instead of *d*. Those assessments are still bounded by $O(ed^2)$ but offer stronger results if the worst case $i=O(d)$ is not met.

Note that the fine grained algorithms handle values separately instead of organized in variables. Those algorithms are not aware of values being neighboring; therefore treat all variables as discrete. Because of that, the fine grained algorithms are inherently ineffective in some kinds of problems.

4.2 AC algorithms for discrete ad hoc constraints

We will now describe well known AC algorithms for ad-hoc constraints. First we will focus on the algorithms that expect the variables to be discrete. In the next chapter we will then focus on algorithms reading domains in list-of-intervals form.

For now, we will define an example constraint, which we will use in figures depicting the algorithms' data structures. Our constraint will bind variables A, B and C, each with the initial domain $\{0, 1, 2\}$. Triples allowed by the constraint are those, for which $(A + B + C) \bmod 2 = 0$. Although this is not a constraint definition by extension, we can represent it in a constraint domain

consisting of 14 triples.

4.2.1 Tuple list[4]

The algorithms described in chapter 2 are in fact for ad-hoc constraints. When dealing with constraints defined by mathematical or logical formula, the algorithms broke the constraint definition into a list of tuples as if it was defined in extension. Problem of these algorithms was their dealing with binary constraints only. Bessière and Régin proposed in [5] a new scheme of ad hoc arbitrary-arity algorithms. As the first instantiation of this scheme they introduced an algorithm, on which we will focus now.

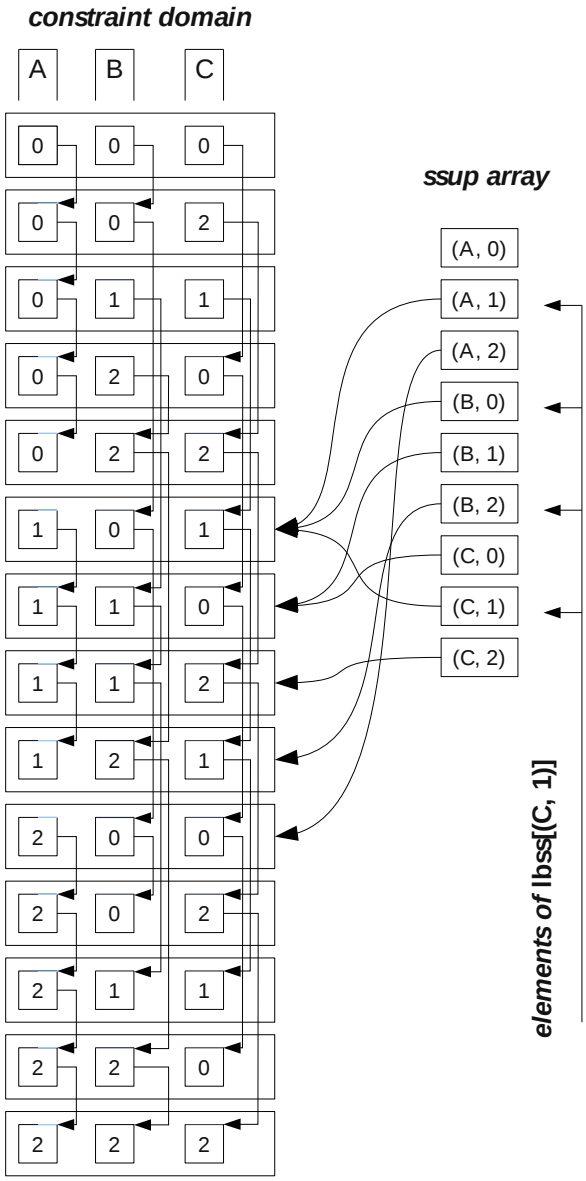


Figure 9: The tuple list data structures after pruning value (A, 0)

The algorithm is based on a generalization of the AC-6 or the AC-7 algorithm for arbitrary-arity constraints. We were trying to describe these AC algorithms in a more general form. However, there is one non-trivial aspect of the generalization: the sparse matrix. We will now explain why such data structure, depicted in figure 1, naturally emerges in AC-4.

Considering the constraint domain as the set of tuples, we were using two equivalence relations on this set. One equivalence represented the columns and the other equivalence forms the rows in the matrix in figure 1. The first equivalence is grouping tuples with the same value of the variable *V* and the second one is defined by projection to the variable *W*. Let us consider creating a system of linked lists such that each list represents one class of the given equivalence relation. We can extend the object's data structure by one pointer to link the objects directly because each object is in exactly one equivalence class. The AC-4 algorithm uses two equivalence relations, therefore each tuple has two pointers (horizontal and vertical). Let us denote *a* the arity of the constraint, on which the “tuple list

4 It is not very common for the author of the paper to give a name to the newly proposed algorithm, especially when the algorithm is not the main concern of the paper. Therefore we use the name given to the algorithm's constraint domain representation by Cheng and Yap in [12].

algorithm” is used. Then we will need *a* equivalence relations linked by *a* pointers in each tuple. For example in figure 9, three systems of linked lists are shown in the columns representing variables “A”, “B” and “C” respectively. When generalizing the sparse matrix for the tuple list algorithm, we need to operate on a class of supports of given value. In such class, tuples have one value fixed, while values of the other *a-1* dimensions vary. The classes represent *a-1* dimensional hyperplanes in the hypercube (That may not be the most intuitive way of generalizing the sparse matrix). Figure 10 shows constraint domain of our example constraint in 3-dimensional space. One system of linked lists, representing one equivalence relation, is shown. The other two systems group tuples into planes oriented in the other dimensions.

As a result, each tuple in the algorithm's constraint-domain representation has *a* pointers (as in figure 9). Let *i*-th value in the tuple be *(V, v)*, then the *i*-th pointer points to the next tuple supporting *(V, v)*. When processing a pending value *(V, v)*, the algorithm uses *ssup* to determine the position of the lost support. To find a new support, the algorithm dereferences the *i*-th pointer and checks, whether the target tuple is feasible (all values in the tuple are in actual domains of their respective variables). When needed, the algorithm is moving along *i*-th tuples' pointers until a feasible tuple is found. If no support is found, the values determined by *lbss* become pending, just like in the AC-6 algorithm. Of course, the multi-directional version of the algorithm first tries to find the new support by the method analogical to the AC-7's.

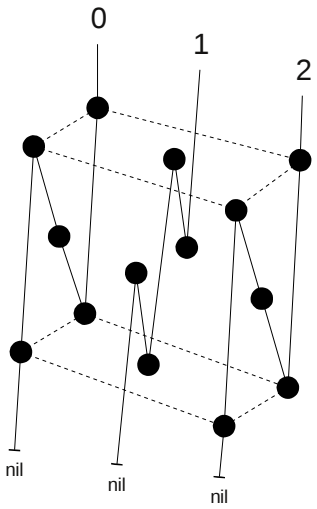


Figure 10: one system of linked lists

For example, let us consider a situation in which the value *(C, 1)* is pruned (because of some other constraint). Then values listed in *lbss[(C, 1)]* become pending, because the smallest supports for these values become invalid. All four values listed in *lbss[(C, 1)]* are enqueued as pending. For example the value *(B, 0)* has the smallest support *csup[(B, 0)] = (1, 0, 1)*. When the value *(B, 0)* will be popped from the queue of pending values, the algorithm will try to find the new support. The tuple *(1, 0, 1)* has three links, we will dereference link in the column of the variable *B*. The link points to the tuple *(2, 0, 0)* which is feasible. If the tuple was not feasible (contained a pruned value), we would continue by using the B-column link of the tuple *(2, 0, 0)* and so on until the feasible support is found.

Denoting *a* the constraint's arity and *t* the number of tuples, the space complexity of the algorithm is *O(at)*. Each pointer is used at most once, therefore the worst-case time complexity is *O(at)* too.

4.2.2 Trie

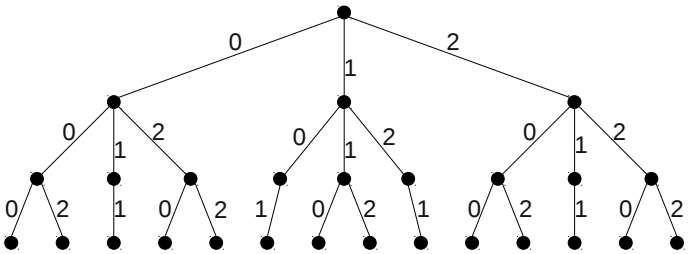


Figure 11: a trie for the example constraint

The drawback of tuple list is that the algorithm must traverse the whole link list when there is no support. To save some tuple viability checks, Gent et al. [16] proposed storing the constraint domain in trie [15], the well known data structure for string algorithms.

To define the trie data structure, we will need some string-like terms. We will say that *m*-tuple $(a_1, a_2, \ldots, a_m)$ is a prefix of *n*-tuple $(b_1, b_2, \ldots, b_n)$ iff $m \leq n$ and $a_i = b_i$ for $1 \leq i \leq m$. The nearest proper prefix of *m*-tuple $(\ldots, a_m)$ is the tuple's prefix of length *m-1*, while a_m will be called the *distinguishing value*. Then a *trie* over the constraint domain is a tree with nodes representing all prefixes of all tuples in the constraint domain. There is an edge between the parent node *P* and the child node *C* iff *P* is the nearest proper prefix of *C*. Each edge is labeled by the respective distinguishing value and each node is determined by concatenation of labels along the path from the root to the node.

Prefixes of the same length form layers in the trie. The constraint domain tuples are stored in the layer of leafs. Distinguishing values in the *l*-th layer relate to the *l*-th variable in the constraint's scope. A tuple represented by a leaf is feasible iff the distinguishing values along the path from the root to the leaf are feasible (in the current domains of the respective variables).

Let us consider a child of the root and it's distinguishing value *v*. A subtree determined by the child contains tuples having *v* as the first element. Search for supports of *v* is done by traversing the subtree. As in the AC-6, the algorithm maintains reference to leaf representing the current smallest support. When the support is lost, the algorithm search for the next support by depth-first-search-like traversing: First visit the node's children; when returning from children, step to the node's sibling; if the node is right-most of the siblings, move to the node's parent. These steps are iterated until some leaf that represents a feasible tuple is reached. Compared to the tuple list, an advantage of using the trie is that the support-search algorithm can omit whole subtrees rooted in the nodes having the distinguishing value pruned.

This way the algorithm searches for supports of the variable related to the first level of the trie. Trying to apply this algorithm to generate supports of other variables discards advantages of using the trie. Therefore authors of the algorithm recommend to use *a* tries for an *a*-ary constraint, having for each variable one trie that has it's first level differentiated by the variable.

Each trie has *t* leafs and *a* layers. Therefore the trie has at most *O(at)* nodes, all tries consume space $O(a^2t)$. One trie can be built in time *O(at)*. Building tries is the limiting factor, since each edge of a trie is used at most two times. Therefore the worst-case time complexity of the algorithm is $O(a^2t)$.

4.2.3 Multi-valued decision diagram

In the string algorithms theory, a generalization of the trie data structure called *directed acyclic word graph (DAWG)* [6] was proposed in order to reduce the memory consumption. The trie algorithm is traversing the trie, therefore a more compact data structure can save not only the memory but also a significant amount of time. In the constraint satisfaction, a data structure analogical to DAWG is called *multi-valued decision diagram (MDD)* and was introduced by Cheng and Yap in [12].

In the trie, each child had one parent (except for the root). The MDD allows a node to have more neighbors in the preceding layer. Therefore MDD forms a directed acyclic graph instead of a tree. The trie can contain couple of subtrees that are pair-wise isomorphic. In the MDD all those subtrees can collapse into one copy. Figure 12 shows the resulting MDD after one such operation performed on the trie from figure 11. We expect all possible collapses to be done when creating the MDD, as depicted in figure 13. Note that using this rule, all leafs collapse into one leaf. On the other hand, the MDD keeps some properties of the trie: The MDD also consists of layers related to the variables in the constraint's scope. Again, a tuple is represented by the MDD iff there exists a path from the root to the leaf that has edges labeled by the tuple's respective values. The tuple is feasible, is all edges of the path have feasible labels (Edge has a feasible label iff the value on the label is not pruned).

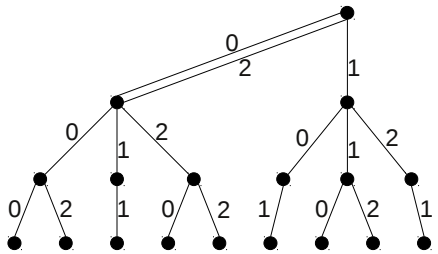


Figure 12: MDD got from the trie after one operation of collapsing isomorphic subtries

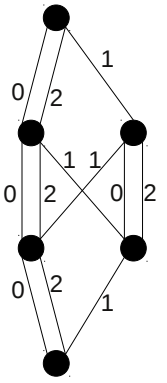


Figure 13: MDD for the example constraint

Although the trie and the MDD data structures are similar, their respective pruning algorithms are significantly different. The trie algorithm was fine grained, keeping the smallest support and dealing with a loss of the support. On the other hand, the MDD algorithm called *MDDC* is coarse grained, in fact it is a propagator: The algorithm gets the input domains of all constraint's variables and computes all the output domains.

To describe the algorithm, let us consider the nodes of the MDD to be pseudovalues defined in this way: A node represents the set of all tuples using the node in their path from the root to the leaf.

Declaring the node to be a pseudovalue gives meaning to terms *feasible node* and *pruned node*. A node is feasible when these conditions hold:

1. The node is reachable (by an oriented path) from the root only by edges that has feasible labels, and
2. The leaf is reachable from the node also by edges having feasible labels

The oriented path from the first rule and the oriented path from the second rule together form a path from the root to the leaf. Labels along this path determine a tuple that is support of the node. If some of the conditions does not hold, there is no support, ergo the node not feasible.

Consider an edge *e* that has feasible nodes on both ends. Again, there is a path from the root to the leaf using the edge. The labels along the path define a feasible tuple. The tuple supports each value contained in the tuple, among others the value of the edge's *e* label.

The algorithm for searching the next support in the MDD is called MDDC. It traverses the MDD from the root in the depth-first-search way. The algorithm uses only edges having feasible labels. Therefore the condition 1 is implicit for all visited nodes. The algorithm determines by results of recursive calls, whether the condition 2 holds for a current node. Each of the recursive calls returns success if the leaf was reached and failure if the leaf is not reachable. When none of the recursive calls has reached the leaf, the node is pruned and failure is returned. Otherwise the node is feasible and the call returns success. In such case, not only the current node is feasible, the parent node of the current node will be marked as feasible too. Therefore an edge between the current and the parent node is feasible and the value of the edge's label is supported. The algorithm collects all these labels and returns them as the new current domains after inspecting the whole MDD.

When it is determined whether a node is feasible or pruned, this information is saved. Next time the algorithm reaches an already visited node, the saved value is returned instead of searching the subtree again. Once a node is pruned, it cannot become feasible until backtrack occurs. Therefore the algorithm can store the set of pruned nodes in its internal state. Cheng and Yap recommend to use the sparse set data structure to store the set of pruned nodes. On the other hand, a set of feasible nodes can shrink. Therefore the algorithm must empty the set of feasible nodes at the beginning each propagation.

To estimate the complexities of the MDD we will focus on the global constraint called *regular*, introduced by Pesant in [20]. The regular constraint is described by two parameters: an arity *a* and a deterministic finite state automaton (DFA) *F*. Figure 14 shows the DFA for the example constraint. The constraint allows tuples that have length *a* and are accepted by the DFA *F*. Denote *f* the count of transitions in the DFA *F*. The regular constraint description can be disposed of the length parameter, implanting it into the DFA. Instead of the original DFA, we will use a layered DFA that has

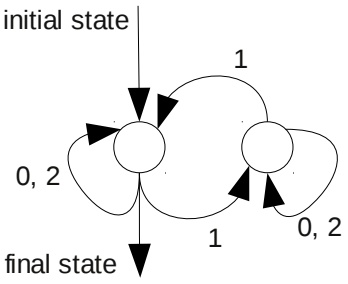


Figure 14: a DFA defining the example constraint domain

O(af) transitions[5]. The layered DFA can be used as MDD (providing all final states of the DFA were collapsed into one).

The algorithm uses each edge at most twice during a single propagation. Denoting *f* the minimal number of transitions of DFA defining the constraint domain, the space complexity of the MDD and the worst-case time complexity of single propagation is *O(af)*. There are *O(ad)* values to be pruned, where *d* is the size of the largest initial domain. Therefore the single-branch worst-case time complexity is $O(a^2df)$.

4.3 AC algorithms for continuous ad hoc constraints

The previous algorithms accepted domains in the bit-array form. We have explained that the list-of-intervals form is more suitable when neighboring values of variables are statistically dependent. We need to define new algorithms for continuous variables because the algorithms from the previous chapter would cease the advantages of using the list of intervals.

4.3.1 The case in SICStus

One of the ways to exploit the list-of-intervals representation of the domains is to generalize the MDD. The original MDD has values in the labels of edges. Labeling the edges by intervals instead of values result in the format of the SICStus case constraint. In fact, the paper [12] of Cheng and Yap on MDD contains the algorithm analogical to the algorithm given by Mats Carlsson at presentation [9]. The difference is in the definition of an edge having a feasible label: In the “case” form, the label is an interval and it is feasible iff the interval intersects the current domain of the respective variable.

4.3.2 Set of rectangles

Particular constraints can have specific properties that a filtering algorithm can exploit. There are many algorithms being strong on some classes of constraints. One of the typical approaches is to identify rectangles that cover the constraint domain. It is one of the steps done in figure 2. Some algorithms might require rectangles to be non-overlapping. We will not restrict the overlapping of the rectangles for now. Another variation of algorithms can accept also discontinuous rectangles defined as Cartesian products of general sets instead of intervals. There are also algorithms that require continuity of rectangles only with respect to some dimensions of the rectangle. It is natural to accept continuous rectangles only when working with the list-of-intervals.

First we will describe the simplest algorithm, the first of two algorithms introduced by Barták

5 There is DFA with *a+1* states that accepts language of all words of length *a*. We need to intersect this language with the language defined by the DFA *F*. It can be done by creating DFA with states defined as Cartesian product of state sets of automatons. The resulting DFA (the layered DFA) has *(a+1)*-times more states than the DFA *F* and *(a+1)f* transitions.

and Mecl in [1]. We will consider a rectangle to be a pseudovalue, thus giving meaning to terms “supported rectangle”, “feasible rectangle”, etc.. Consider the rectangle and the interval induced as the projection of the rectangle into the dimension of one of the variables. The rectangle is supported in the variable iff the variable's current domain intersects with the interval. On the other hand, the interval is a set of values of the variable that are supported by the rectangle.

The algorithm process each rectangle separately. To check, whether a rectangle is feasible, the algorithm ensures that the rectangle is supported in each of the variables (by checking the intersection of the current domain and the projected interval). If the rectangle is feasible, the algorithm adds each projection into the new domain of the respective variable. The algorithm processes all rectangles forming the constraint domain this way. At the end, the algorithm ensures the domain contraction by intersecting each new domain with the corresponding old domain.

Denote *i* the number of intervals the domain consist of. Symbol *r* will refer to the number of rectangles the constraint domain is decomposed to. The decision, whether rectangle projection intersects the domain, can be made in time *O(log i)* using binary search. All rectangles can be processed in time *O(i+r.log i)* including creation of array for the binary search. To create the new domain, the algorithm unions the projections of feasible rectangles. This can also be done in time *O(i+r.log i)*. Then the worst-case time complexity of the propagation is *O(a(i+r.log i))*, where *a* is arity of the constraint.

4.3.3 Sweep pruning

The other algorithm Barták and Mecl proposed in [1] is more complex. The algorithm is based on a concept of sweep from the computational geometry. The algorithm is designed for binary constraints. The rectangles are considered as placed in a plane with axes *x* and *y*. The algorithm keeps a line called the *sweep line* that is orthogonal to axis *x* and is being moved with ascending *x*. For the current position of the sweep line the algorithm maintains a set of *active* rectangles. The rectangle is active iff it intersects with the current sweep line and is supported in the domain of *Y*. Beside of the set of active rectangles the algorithm also keeps an information, whether the current position of the sweep is inside of the domain of the variable *X*.

The state of the sweep line can change at the positions of *x* where

- some rectangle starts or ends (with respect to the projection of the rectangle to the dimension *X*)
- an interval of the domain of variable *X* starts or ends

Such positions of *x* are called *events* and are processed in the order of increasing *x*. Events of starting or ending a rectangle also carry a information identifying the rectangle. Each of the 4 kinds of events has it's processing procedure. Firstly when processing the start/end of a rectangle, the rectangle is added/removed from the set of active rectangles. When processing the start/end of a domain interval, the inside flag is set/cleared.

The operations described so far have only maintained the sweep line state. Besides that, the

algorithm also uses event processing to build new domains. The new domain of variable Y is determined by the union of Y-projections of all feasible rectangles. The rectangle is feasible if it is active and the inside flag is set. To follow this rule the event processing checks one condition when the other condition is starting to hold. Namely:

- when the rectangle starts and the rectangle is supported in the current Y-domain, the rectangle becomes active. The algorithm therefore checks the inside flag and considers the rectangle feasible if the flag is set.
- when the domain interval starts, the inside flag is being set. Thus all active rectangles become feasible.

When the conditions meet either way, the algorithm adds Y-projection of the feasible rectangle into the new Y-domain.

The new domain of the variable X is being build using a similar principle. Value v is in the new X-domain iff v is in the old X-domain and there is an active rectangle in the sweep line at position v. Again, there are two conditions that must coincide. The event processing checks one condition when the other starts to hold:

- when processing the domain-interval start event, the algorithm checks, whether there is at least one active rectangle
- when processing the rectangle start event, the algorithm can transit from state in which there were no active rectangles to state in which there is an active rectangle. Then the algorithm checks, whether the inside flag is set

One way or the other, when the conditions coincide, a new interval of the output X-domain is started.

The algorithm must also recognize the end of the interval: When one of the conditions ceases, the event processing checks the other condition. If the other condition does not hold, the X-domain interval is already closed. On the other hand, if one condition holds while the other ceases, the interval of the new X-domain should be ended. Again, there are two conditions and therefore two ways this can happen in:

- when processing the rectangle end, the rectangle becomes inactive. If the rectangle was the only active, the algorithm must check, whether the inside flag is set
- when processing the domain interval end event, the algorithm checks, whether there are any active rectangles

If one of these situations occur, the interval of the output X-domain is ended.

We should mention a technical detail that is important for the algorithm to work properly. There might be two events on the same position such that one sets one of the watched conditions and the other ceases the other condition. For example one event starts an active rectangle while the other event ends a domain interval. If these events occur on the same position, the conditions meet for a moment and the rectangle is feasible. It is important to first process the rectangle start and then the end of the domain interval. The other ordering would fail to recognize the rectangle feasible. As a

general rule to solve all problems of this kind, the algorithm always processes the start events before the end events of the same position.

We will now estimate the worst-case time complexity, using i and r as before. There are r events of rectangle start, each of which requires $O(\log i)$ time to determine whether the Y-projection intersects the Y-domain. Again, processing of all these events requires $O(i+r.\log i)$ time. The rectangle end event can be processed in a constant time. The domain interval start event occurs i times. Except for processing a feasible rectangles only constant work is done. Each rectangle is processed as feasible only once during the propagation, assuming some additional effort of the algorithm. Each feasible rectangle generates Y-projection interval. The algorithm needs $O(r.\log r)$ time to union all these intervals. The end of domain interval events require time $O(i)$ because processing of a single event is constant. Putting it all together, the propagation's worst-case time complexity is $O(i+r.\log r+r.\log i)$.

4.3.4 Box constraints collections

To generalize decomposing the constraint domain into rectangles, Cheng et al. [10] expanded a repertoire of covering objects by introducing *triangles*. A triangle is a hyper-rectangle intersected by half-space defined by linear inequality of the form

$$\sum p_i x_i \leqslant q \tag{4.6}$$

The common name for both triangles and hyper-rectangles is a *box*.

Let us assume that all p_i are positive. We can separate x_1, leading to inequality:

$$x_1 \leqslant \frac{q - \sum_{i \neq 1} p_i x_i}{p_1} \tag{4.7}$$

Let $min_i..max_i$ be the projection of the box into i-th dimension and let D_i be the current domain of the i-th variable. Then

$$x_i \in D_i \cap (min_i .. max_i) \tag{4.8}$$

therefore

$$x_i \geqslant min(D_i \cap (min_i .. max_i)) \tag{4.9}$$

By replacing x_i in (4.7) by the right side of inequality (4.9), the inequality is preserved and we get

$$x_1 \leqslant \frac{q - \sum_{i \neq 1} p_i min(D_i \cap (min_i .. max_i))}{p_1}$$, leading to the interval notation

$$x_1 \in (min_1 .. \frac{q - \sum_{i \neq 1} p_i min(D_i \cap (min_i .. max_i))}{p_1}) \tag{4.10}$$

When p_i is negative, some inequalities revert, causing usage of the maximum instead of the minimum.

A box that has the intersection $D_i \cap (min_i..max_i)$ empty is not supported in the domain of the i-th

variable. All such boxes are ignored by the algorithm. The algorithm computes the intervals like in (4.10) for all other triangular boxes. For hyper-rectangle boxes the needed interval is just $x_1 \in (min_1..max_1)$. Union of all the resulting intervals is the new domain of X_1. The algorithm computes new domains of all other variables in the same way.

The original version of the algorithm instantiate constants p_i and group expressions for all triangles into one big expression. The expression is then written in an indexicals formalism supported by CSP solvers.

A question we have not answered yet is how to build the collection of boxes for a given constraint. Because finding the optimal collection is expensive, the authors propose greedy algorithm giving not necessarily optimal collection. The algorithm works with three kinds of tuples:

- *positive tuples* must be covered by boxes
- *neutral tuples* may or may not be covered. For example already covered tuples are neutral. Some other optimizations in the Cheng's paper produce neutral tuples.
- *negative tuples* must not be covered by boxes.

Initially all tuples in the constraint domain are positive and tuples outside of the constraint domain are negative.

Searching for the box starts by picking a random positive tuple (the tuple is considered as hyper-rectangle of size 1). The algorithm tries to extend the hyper-rectangle of the box in one of the dimensions. New tuples might be of all kinds, the algorithm only needs to enforce that positive and negative tuples will be separable by a hyper-plane in (4.6). To test the separability and to find parameters p_i and q, the algorithms keeps all restrictions on the parameters in a system of linear equations. If the tuple $(d_1, d_2, \ldots, d_n)$ being added is positive, equation $\sum p_i d_i \leqslant q$ is added to the system. If the tuple is negative, equation $\sum p_i d_i > q$ is added to the system. No equation is added to the system for a neutral tuple. When extending the box, the algorithm adds equations for all new tuples. If the system of equations has a solution, the algorithm commits to the extension. Otherwise the algorithm tries to extend the box in the other direction or another dimension. When there is no box extension possible, the algorithm computes parameters p_i and q and adds the box into the collection. All positive tuples in the box become neutral. The algorithm searches for another box or terminates if there are no positive tuples left.

While the algorithm described in chapter 4.3.2 has worst-case time complexity of one propagation $O(a(i+r.\log\ i))$, this algorithm has the complexity $O(a(i+b.\log\ i))$. Boxes are more expressive than rectangles therefore b might be much less than r.

5 Elementary propagators

In this chapter we will focus on some propagators for binary constraints that we can use on subconstraints. Propagators are unidirectional and deal with the source and the target variable. The source variable is used for searching for supports only, while the target domain is being pruned.

5.1 AC-3.1-like propagators

5.1.1 AC-3 propagator

The first propagator we will focus on is the AC-3 Revise procedure described in chapter 2.3.1: (Recall that we use term “extent” for the size of the bit-array representation as opposed to term “size” that refers to the number of elements the represented set has)

Code 5.1: AC-3 propagator

```
 1  structure AC3.Constraint
 2    supports : array[0..targetDom.extent-1] of BitArray[0..sourceDom.extent-1]
 3  end structure

 4  structure AC3.InternalState
 5    /* empty */
 6  end structure

 7  procedure AC3(constraint, state, sourceDom : BitArray, targetDom : BitArray)
 8    for i in targetDom
 9      if {} = sourceDom ∩ constraint.supports[i] then
10        targetDom := targetDom \ {i}
11      end if
12    end for
13  end procedure
```

In the code above, we have defined data structures `AC3.Constraint` and `AC3.InternalState`. The “constraint” data structure is shared by all calls of the propagator. On the other hand, the “internal state” data structure passes data between the consecutive calls within a single search-tree branch. Therefore the internal state must be cloned in order to backtrack. In order to understand the propagators, it is necessary to distinguish between data stored in the internal state and the “constraint” data structure. Therefore we will declare these data structures in all pseudo-code segments describing the propagators.

We will denote d_S the size of the representation of the source domain, d_T the size of the representation of the target domain and c_T the number of elements in the target domain. The time complexity of a single propagator call is $O(d_T+c_T d_S)$ as we have shown in chapter 2.3.1.

5.1.2 AC-3.1 propagator

To change propagator from code 5.1 to the AC-3.1 algorithm, we will modify the test on line 9 of the code 5.1. A `IntersectUsingHint` procedure will be used to determine whether the source domain and the set of supports intersect. The procedure gets a hint as an argument in addition to the arguments of two sets being intersected. The hint is an index of CPU word in the bit-array representation where the procedure found the largest support the last time the procedure was called. Because the domains are only shrinking, no common element (support) can be found past the hint. Therefore the procedure scans the bit-arrays from the position of the hint until a support is found. The position of the newly found support is set as a new hint. If no support is found, the hint is set to −1 and the sets does not intersect.

In fact the hint is analogical to the smallest support that the AC-3.1 algorithm saves. There are two technical differences: The first is that while the classical AC-3.1 stored the smallest support, the hint points to the largest support. The reason is that −1 is easier to test than the past-the-end word index of the bit array. The second difference is that the hint points to the bit-array chunk instead of the individual element. The chunk position would have been computed from the individual element's position anyway.

The AC-3.1 algorithm stores the smallest supports for each value in the propagator's internal state. The hint version will analogically store the hints. The initial value of hints is the index of the last CPU-word in the bit-array representation of the sets (namely `(extent−1)/WORD_SIZE`). To illustrate how CPU-friendly the procedure is, the code is written in C language:

Code 5.2: IntersectUsingHint procedure

```
void IntersectUsingHint(unsigned int* bit_array_data_1,
                        unsigned int* bit_array_data_2,
                        int* hint)
{
  while (*hint>=0 && 0==(bit_array_data_1[*hint] & bit_array_data_2[*hint])) {
    (*hint)--;
  }
}
```

The resulting code for the AC-3.1 algorithm is then

Code 5.3: AC-3.1 propagator

```
1  structure AC31.Constraint
2    supports : array[0..targetDom.extent-1] of BitArray[0..sourceDom.extent-1]
3  end structure

4  structure AC31.InternalState
5    hints : array[0..targetDom.extent-1] of integer
6  end structure
```

```
 7  procedure AC31(constraint, state, sourceDom : BitArray, targetDom : BitArray)
 8    for i in targetDom
 9      IntersectUsingHint(sourceDom, constraint.supports[i], state.hints[i])
10      if -1 = state.hints[i] then
11        targetDom := targetDom \ {i}
12      end if
13    end for
14  end procedure
```

Consider all calls of this propagator in a single branch of the search tree. The initial hint value is $O(d_S)$, the internal state consists of d_T hints, therefore the hint can be decremented at most $O(d_S d_T)$ times. Except for decrementation, one iteration of the "for" loop at line 8 requires a constant time. Propagator is called at most d_S times (when the source domain shrinks). The "for" loop has at most d_T iterations. Therefore the single-branch time complexity of the algorithm is $O(d_S d_T)$.

5.2 Building propagators

We will now present new propagator algorithms that are asymptotically worse than the AC-3.1 but practically behave better in some situations as the experiments showed.

5.2.1 Simple builder

The previous algorithms were checking whether the value has a support and pruning those values that had no support. The following algorithms will build the domain by collecting the compatible values of all values in the source domain. We will need a different organization of the constraint domain data, so we define the array *compatible* of bit arrays as:

$$i \in compatible[j] \overset{def.}{\Leftrightarrow} j \in supports[i]$$

As a consequence, the bits representing the constraint domain are organized into CPU words of bit arrays differently. An example for four-bit processor is shown in figure 15. Code 5.4 shows the simplest algorithm based on this idea.

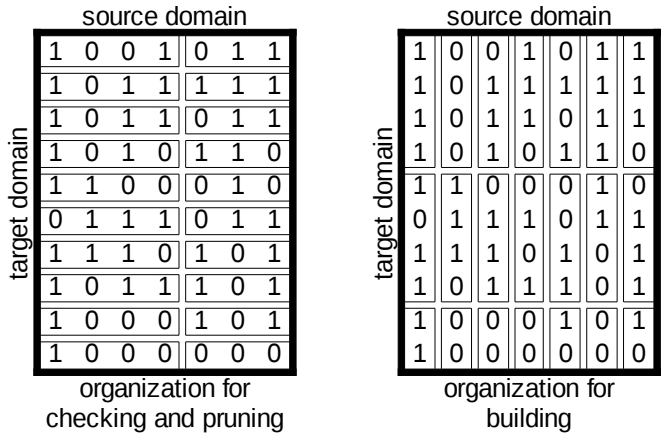


Figure 15: constraint domain data organization

Code 5.4: Simple builder

```
structure SimpleBuilder.Constraint
  compatible : array[0..sourceDom.extent-1] of BitArray[0..targetDom.extent-1]
end structure

structure SimpleBuilder.InternalState
  /* empty */
end structure

procedure SimpleBuilder(constraint, state, sourceDom : BitArray, targetDom : BitArray)
  accumulator := {}
  for i in sourceDom
    accumulator := accumulator ∪ constraint.compatible[i]
  end for
  targetDom := targetDom ∩ accumulator  /* to be contracting */
end procedure
```

This procedure has time complexity $O(d_S+c_Sd_T)$, which is comparable to the time complexity of the AC-3 propagator. In order to create a more efficient algorithm we will introduce a more complex data structure.

5.2.2 Tree builder

The *tree builder* algorithm will build binary trees consisting of nodes that will carry partly accumulated sets of compatible values. Each node *i* has its *check set* and a *result set* such that $resultSet(node)=\cup_{j\in checkSet(node)} compatible[j]$

The goal is to create or to have available such a node that its check set is equal to the source domain of the current propagation call. Then we can use the result set of this node to prune the target domain, in the same way the accumulator was used in the code 5.4.

The initialization of the tree is started by creating the nodes for singleton check sets. Their result sets are trivially the members of the array *compatible*. The nodes for singleton check sets will be the leafs of the trees in our data structure. The inner nodes of the trees will be created by a `Join` procedure. Given node indices *i* and *j*, the procedure will create a new node *k* with $checkSet(k)=checkSet(i)\cup checkSet(j)$ and $resultSet(k)=resultSet(i)\cup resultSet(j)$. Using the `Join` procedure we will build the balanced binary tree over the leafs. When the perfect balance of the tree cannot be achieved, any tree with the minimal (logarithmic) possible height can be created. The check set of the root of the initial tree is the initial domain of the source variable.

The propagator will store the index of the root of the most recently generated tree in its internal state. When called again, the propagator will create a new tree. The previous tree will be preserved intact to reuse in case of backtracking.

The new tree is created in such way that the check set of its root is the current domain of the source variable. Recall that then the result set of the new root can be used for pruning the target domain. The new tree will share as many nodes with the previous tree as possible. For example consider a situation in which the source domain is one element smaller compared to the previous

propagator call. Then one leaf will be left out. The nodes on the path from this leaf to the root will have to be created for the new tree while other nodes will be shared with the previous tree (as shown in figure 17).

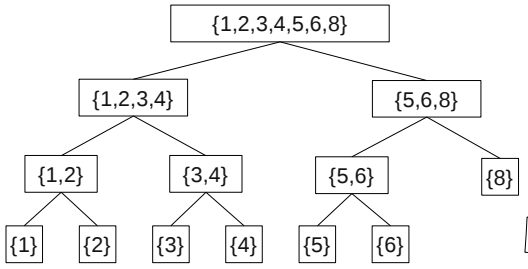


Figure 16: An example tree: the initial source domain was 1..8 and value 7 is pruned

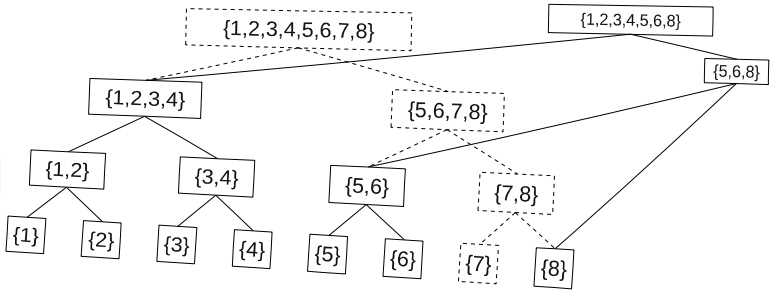


Figure 17: Tree from figure 16 sharing nodes with an initial tree

To build the new tree we will use the procedure `Restrict`. The arguments of the `Restrict` procedure are the current source domain and the node, the restriction of which should be returned. The `Restrict` returns `nil` if it reaches bottom of the tree. Otherwise it returns the node that has the check set equal to the intersection of the argument node's check set and the current source domain. As a special case, `Restrict` on the root of the previous tree will return the root of the desired new tree. The `Restrict` procedure will do its job this way:

If the argument node's check set is subset of the source domain, no restriction is needed and the procedure will return the argument node (for example node "*{1, 2, 3, 4}*" in figure 17). Otherwise the procedure returns the join of results of the recursive calls on the node's children. The recursive call may return `nil`, in such a case the procedure returns the result of the other child (Restrict on node "*{7, 8}*" returned node "*{8}*"). The recursion is stopped at leafs by returning `nil`. The complete procedure is shown in the following code:

Code 5.5: Restriction procedure

```
 1  procedure Restrict(sourceDom : BitArray, node, in out last)
 2    if node.checkSet ⊆ sourceDom then return node
 3    if node.isLeaf then
 4      return nil
 5    else
 6      restrLeft  := Restrict(sourceDom, node.left, last)
 7      restrRight := Restrict(sourceDom, node.right, last)
 8      if restrLeft = nil and restrRight = nil then return nil
 9      else if restrLeft  = nil then return restr.Right
10      else if restrRight = nil then return restr.Left
11      else return Join(restr.Left, restr.Right, last) /* Join creates a new node */
12    end if
13  end procedure
```

For efficiency reasons we perform the inclusion check from line 2 only if the whole check set

falls into one bit-array chunk. Such check is performed in a constant time. Nodes with so compact check sets are in the lower part of the tree. In the upper part, the check set is not stored because it cannot be represented by a single bit-array chunk. We will recognize the inclusion relation from line 2 by the results of the recursive calls on node's children: The inclusion relation holds iff *restrLeft = node.left* and *restrRight = node.right*. This way, the actual subset checks are delegated to the highest level of the lower part of the tree (border level). This algorithm alteration has only the multiplicative effect on the time complexity because the number of nodes in the border level is proportional to the size of the bit-array representation in the root. On the other hand, it allows us to store only constant memory bit-array chunk as node's check set.

All nodes are stored in a dynamic array. The array is enlarging on demand and does not need to be stored in a continuous region of the memory. The propagator remembers the index of the last valid element of the array, which is shifted when the `Join` procedure creates a new node. The last-valid index is stored in the propagator's internal state. Thus when backtrack occurs, all nodes created in the failed branch of the search tree will be invalidated just by restoring the last-valid index from the appropriate internal state. Creating new nodes then overwrites the memory of nodes created in the failed branches of the search tree.

Code 5.6: Tree builder propagator

```
structure TreeBuilder.constraint
  nodes : array[0..] of Node
end structure

structure TreeBuilder.InternalState
  root : integer
  last : integer
end structure

procedure TreeBuilder(constraint, state, sourceDom : BitArray, targetDom : BitArray)
  constraint.nodes.size := state.last   /* overwriting data of backtracked branches */
  newRoot := Restrict(sourceDom, state.root, state.last)
  targetDom := targetDom ∩ constraint.nodes[newRoot].resultSet
  state.last := constraint.nodes.size
  state.root := newRoot
end procedure
```

Top-most `Restrict` typically calls `Join`, which returns an index of a newly created node. Therefore in such a case `root = last`. This is not always the case, therefore we need separated `root` and `last` fields in the internal state.

Also note that in contrast with all the previous propagators, the representation of the constraint domain cannot be shared between the constraints having the same constraint domain.

Tree builder analysis

We will estimate the number of nodes that can be generated in a single branch of the search tree. Each call of the propagator removes the leafs representing the elements of the set difference of the previous and the current source domain. The nodes on the paths from the removed leafs to the root

have to be restricted. If one call of the propagator removed several leafs, the common nodes of the paths from the leafs to the root would be restricted only once. On the other hand, if the propagator was called to remove each leaf separately, it would have to create intermediate restrictions for such nodes. Therefore, as the worst case, we will assume that the source domains for two consecutive calls of the propagator differ only in one element.

We will now examine the effects of removing a leaf. When the leaf in depth d is removed, its parent is left out from the tree and the leaf's sibling takes the place of the parent. The restriction of $d-1$ nodes on the path from the sibling's new position to the root requires creation of $d-1$ new nodes. Other nodes can be shared with the previous tree. We will focus on three characteristics:

- n is the number of nodes of the current tree and all the preceding trees
- l is the number of leafs of the current tree
- s is the sum of depths of all leafs of current tree

As we have shown, when removing the leaf in depth d, $d-1$ new nodes are created. On the other hand, s is decremented by $d+1$: the leaf in depth d is left out and its sibling is moved one level up. The third parameter, l, is decremented by 1. Then the expression $n+s-2l$ is invariant, because $(d-1)-(d+1)-2(-1) = 0$. More formally

$$n_0+s_0-2l_0 = n_1+s_1-2l_1 \tag{5.7}$$

for n, s, l describing two moments indexed by 0 and 1. We will denote the state after initialization by index 0: There are $l_0 = d_S$ leafs, $n_0 = 2d_S-1$ nodes and $s_0 \leq d_S\lceil log_2\, d_S\rceil$ (where d_S is the initial size of the source domain). The final state will by denoted by index 1. There is only one leaf left and the propagator is entailed, $l_1=1$, $s_1=0$ and n_1 is the maximum number of nodes we are trying to estimate. Exploiting the invariant (5.7) we get

$n_1 = n_0+s_0-2l_0-s_1+2l_1 = (2d_S-1)+s_0-2d_S-0+2 = s_0+1 \leq d_S\lceil log_2\, d_S\rceil+1$.

Except for the leafs, all of the $O(d_S log\, d_S)$ nodes were created by the `Join` procedure, which computes the union of d_T-sized bit arrays (to create the result set). Thus all `Join` calls require $O(d_T d_S log\, d_S)$ time in total. To examine the `Restrict` procedure we will consider line 2 of code 5.5 and the rest of the procedure separately. The line 2 requires at most $O(d_S)$ time when called on a root of some tree and $O(d_S).2^{-h}$ for a node in depth h. Sum of the geometric progression, $O(d_S)$, is the time spent on line 2 needed to create a single tree. Therefore, the total time spent on line 2 is $O(d_S^2)$. Other parts of the Restrict procedure, measuring the recursive calls separately, run in $O(1)$ time. The `Restrict` is called $O(d_S\, log\, d_S)$ times. The propagator procedure itself is constant and it is called at most d_S times. Putting it all together, the single-branch worst-case time complexity of the tree builder is $O(d_S^2+d_T d_S\, log\, d_S)$. Although this time complexity is much higher compared to AC-3.1's $O(d_S d_T)$, according to our experiments the builder tree outperforms AC-3.1 when d_S is small.

5.3 Interval list propagators

So far both the source and the target domain were represented as bit arrays. In this chapter, we will introduce algorithms dealing with the source domain in the list-of-intervals representation. We

will gather the constraint domain elements into segments $(min_S..max_S)$-$value_T$, maximal (with respect to inclusion) such that $\forall x \in (min_S..max_S): \langle x, value_T \rangle \in constraint\ domain$

Goal of the propagator is to find all intervals $min_S..max_S$ that intersect with some intervals $min_D..max_D$ in the list of intervals of the current source domain. Values $value_T$ of the intersected segments $(min_S..max_S)$-$value_T$ are supported values in the target domain.

5.3.1 Collecting intersected intervals

We will use the term *gap* for the maximal interval of non-elements of a given set. Having set in the list-of-intervals representation, we can identify the gaps from the limits of the consecutive intervals as $(max_D[i]+1)..(min_D[i+1]-1)$; intervals $-\infty..(min_D[0]-1)$ and $(max_D[last]+1)..+\infty$ are also gaps. An interval $min_S..max_S$ is disjoint with the set iff the whole interval falls into one gap.

This propagator works according to the builder paradigm, collecting all supported values $value_T$ into an accumulator. When finished, the accumulator is intersected with the input target domain resulting in the output target domain.

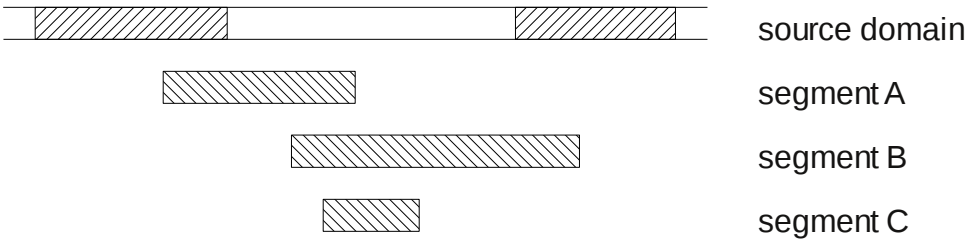


Figure 18: Three cases in determining whether (the interval of) the segment intersects with the source domain

The intersected-intervals propagator will use a list of segments sorted by min_S increasingly. The propagator procedure scans the list in the alternating phases representing the intervals and the gaps of the source domain. In the interval phase the procedure advances to the last segment with min_S not greater than the end of the domain interval. All processed segments (such as segment A in figure 18) are intersected with the current source domain and their respective $value_P$ are put into the accumulator. In the gap phase the propagator advances to last segment with min_S not greater than gap's end. Those of processed segments, which have max_S past the gap's end (like segment B), are intersected. The $value_T$ values of the segments are put into the accumulator. On the other hand other segments (for example segment C) are not intersected.

Code 5.8: Intersected intervals collector

```
structure Segment
  min : integer
  max : integer
  value : integer
end structure
```

```
structure IIC.Constraint
  segments : array[1..] of Segment
end structure

structure IIC.InternalState
  /* empty */
end structure

procedure IIC.Constraint.Initialize(out constraint, in compatible)
    /* compatible represents constraint domain as in code 5.4 */
    /* this code assumes that queries of compatible[s][t] when s is out of range
         are legal and result in 0 */
  for t in 0..targetDom.extent-1
    for s in 0..sourceDom.extent-1
      if compatible[s-1][t] = 0 and compatible[s][t] = 1 then
        min := s
      end if
      if compatible[s][t] = 1 and compatible[s+1][t] = 0 then
        max := s
        constraint.segments.append(new Segment(min, max, t))
      end if
    end for
  end for
end procedure

procedure ProcessInterval(constraint, in out index, in out accumulator, end : integer)
  lastSeg := constraint.segments.size-1
  while index <= lastSeg and constraint.segments[index].min <= end do
    accumulator := accumulator ∪ {constraint.segments[index].value}
    index := index+1
  end while
end procedure

procedure ProcessGap(constraint, in out index, in out accumulator, end : integer)
  lastSeg := constraint.segments.size-1
  while index <= lastSeg and constraint.segments[index].min <= end do
    if constraint.segments[index].max >= end then
      accumulator := accumulator ∪ {constraint.segments[index].value}
    end if
    index := index+1
  end while
end procedure

procedure IIC(constraint, state, sourceDom : ListOfIntervals, targetDom : BitArray)
  accumulator := {}
  index := 0
  for (a..b) in sourceDom.intervals
    ProcessGap(constraint, index, accumulator, a-1)
    ProcessInterval(constraint, index, accumulator, b)
  end for
  ProcessGap(constraint, index, accumulator, +infinity)
  targetDom := targetDom ∩ accumulator
end procedure
```

We will denote i_S the number of intervals of the source domain and g the number of segments; g is $O(d_S d_T)$. The propagator needs only one pass over the sorted list of segments, therefore its time complexity is $O(i_S+g+d_T)$. Summand d_T is induced by intersecting the accumulator and the target domain at the end of the propagation procedure.

5.3.2 Mono-intervalic constraints

We will now introduce a special kind of constraint:

Definition 5.9: Mono-intervalic constraint

We say that n-ary constraint with the constraint domain C is mono-intervalic with respect to the i-th variable of its scope iff for each $v_1, v_2, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$ is the set

$$\{x \mid \langle v_1, v_2, \ldots, v_{i-1}, x, v_{i+1}, \ldots, v_n \rangle \in C\} \quad (5.10)$$

either an interval or an empty set.

A binary constraint is mono-intervalic (with respect to the source variable) if there is at most one $(min_S..max_S)$-$value_T$ segment for every $value_T$ (If there is no segment, the the set (5.10) is empty, if there is exactly one segment, then the set is $(min_S..max_S)$)

When considering the constraints used in CSPs, the mono-intervalic constraints are hardly typical. On the other hand, the pseudovariables are artificial and the design of the pseudovariables determines characteristics of subconstraints. Thus propagators dealing with mono-intervalic constraints might be useful in some designs. For example consider the motivational example on page 17. Both constraints added in the example depicted in the figure 2 are mono-intervalic. As another example, the SICStus's "case" constraint allows edges to be labeled only by intervals, therefore the subconstraint mapping the values to edge indices is mono-intervalic.

5.3.3 Open-Close propagator

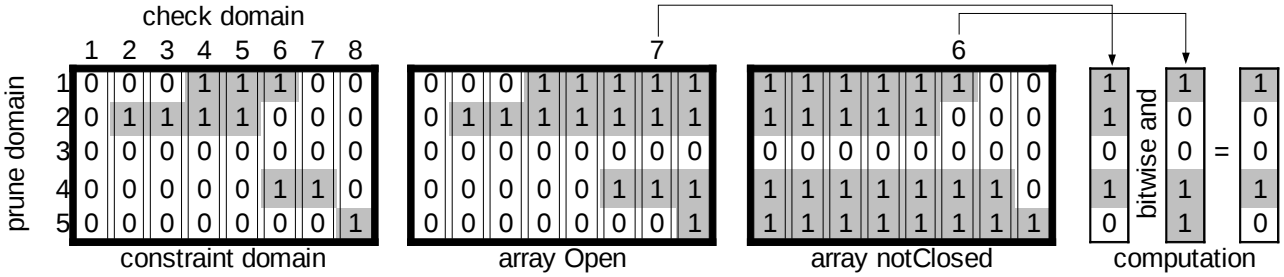


Figure 19: Open-Close data organization and the computation of segments intersected with the interval 6..7

The fact that the constraint domain is mono-intervalic can be exploited in the propagator. We will introduce the propagator that we will call "Open-Close".

We will say that an interval *min..max opens* at point *min* and *closes* at point *max+1*. The interval is *open* at points *min* and greater and it is *closed* at points greater than *max*. The interval *min..max* intersects with the interval *a..b* if $a \leq max$ and $min \leq b$. In the above terms if the interval *min..max* is not closed at point *a* ($a \leq max$) and it is open at point *b* ($min \leq b$). The Open-Close propagator computes the arrays of bit arrays `open` and `notClosed` such that `open[i]` represents the set of values $value_T$ of all segments that are open at point *i*. Analogically `notClosed[j]` represents all segments not closed at point *j* (as shown in figure 19). Then `open[b] ∩ notClosed[a]` is the set

of segments intersected with interval *a..b*. We will compute such intersection for each interval of the source domain. In the accumulator, we store union of all the computed sets. The union is the set of supported values in the target domain.

Code 5.11: Open-Close propagator

```
structure OpenClose.Constraint
  open      : array[0..sourceDom.extent-1] of BitArray[0..targetDom.extent-1]
  notClosed : array[0..sourceDom.extent-1] of BitArray[0..targetDom.extent-1]
end structure

structure OpenClose.InternalState
  /* empty */
end structure

procedure OpenClose.Constraint.Initialize(out constraint, in compatible)
    /* compatible represents constraint domain as in code 5.4 */
  constraint.open := compatible
  for i in {1, 2, ..., sourceDom.extent-1}
    constraint.open[i] := constraint.open[i] ∪ constraint.open[i-1]
  end for

  constraint.notClosed := compatible
  for i in {sourceDom.extent-1, sourceDom.extent-2, ..., 1}
    constraint.notClosed[i-1] := constraint.notClosed[i-1] ∪ constraint.notClosed[i]
  end for
end procedure

procedure OpenClose(constraint, state, sourceDom : ListOfIntervals, targetDom :
BitArray)
  accumulator := {}
  for (a..b) in sourceDom.intervals()
    accumulator := accumulator ∪ (constraint.open[b] ∩ constraint.notClosed[a])
  end for
  targetDom := targetDom ∩ accumulator
end procedure
```

The propagator procedure requires $O(i_S d_T)$ time, which is asymptotically much more than the time complexity of the Intersected intervals collector (*IIC*) from the chapter 5.3.1. Considering g is $O(d_T)$ for the constraints that are mono-intervalic with respect to the target domain, the time complexity of the IIC is $O(i_S+d_T)$. On the other hand, the Open-Close propagator uses bit parallelism and operations that are more processor-friendly than those of the IIC. For small domains, such as those in the application this thesis focuses on, the Open-Close may outperform the IIC.

5.4 Propagators creating the interval lists

In the previous chapter we have described the propagators, which pruned the bit-array target domains based on the list-of-intervals source domains. This chapter analyses the reverse case: the source domain is represented as a bit array and the target domain is in the list-of-intervals form.

5.4.1 Interval list generator

To prune the list-of-intervals domain, we will use the segments as defined in chapter 5.3 with this difference: Because we swapped the representations (the bit array and the list of intervals) of the source and the target domain, we will operate on segments $(min_T..max_T)$-$value_S$ instead of $(min_S..max_S)$-$value_T$.

As the most inner operation, the propagator enumerates the supported segments. The result is a list of such intervals $min_T..max_T$ that their respective $value_S$ are in the source domain, in the order of increasing min_T. Those intervals may overlap, the overlapping intervals are consecutive in this ordering. Thus when enumerating the segments, the propagator can unite the overlapping intervals into one interval. The resulting list of the united intervals conforms the requirements of the list-of-intervals representation. This list is intersected with the input target domain in order to create the output target domain.

Code 5.12: Interval list generator

```
structure ItlListGen.Constraint
  segments : array of Segment  /* array is sorted in order of increasing min */
end structure

structure ItlListGen.InternalState
  /* empty */
end structure

procedure ItlListGen(constraint, state, sourceDom : ListOfIntervals, targetDom :
BitArray)
  list := []
  lastMin := nil
  lastMax := nil
  for segment in constraint.segments
    if segment.value ∈ sourceDom then
      if lastMax < segment.min-1 then
        /* intervals does not overlap, sending the last interval to the final list */
        list.append( lastMin..lastMax )
        lastMin := segment.min
        lastMax := segment.max
      else
        /* unite lastMin..lastMax with segment.min..segment.max */
        lastMax := max(lastMax, segment.max)
      end if
    end if
  end for
  list.append( lastMin..lastMax )
  list.exclude(0) /* deleting first element of list, which is nil..nil */
  targetDom := targetDom ∩ list
end procedure
```

Time complexity of the propagator is $O(g)$, which is $O(d_S)$ for mono-intervalic constraints.

As an alternative, the Tree Builder propagator can also be used to generate target domain in the list-of-intervals form. Only modification needed is to represent the result sets as lists of intervals. We don't discuss this alternative in more detail, because it performed badly in our experiments.

6 Composing algorithms

In this chapter, we will put the ideas of the previous chapters together. We will describe the resulting superconstraint propagation algorithms and estimate their complexities. First we will focus on the constraints described as the set of hyper-rectangles.

6.1 Hyper-rectangle

We expect the domains of real variables to be represented as the lists of intervals. We will denote *i* the length of the longest list of intervals and *r* will stand for the number of rectangles in the constraint definition. The algorithm assigns numbers *1, …, r* to the hyper-rectangles and these numbers will be the pseudovalues of the only pseudovariable. The constraint network is star-shaped as in figure 20.

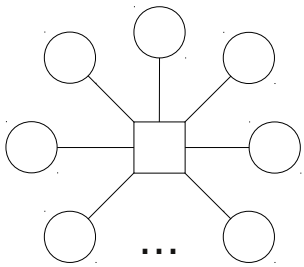


Figure 20: a star shaped subconstraint network

The propagation algorithm works in two phases. First the algorithm propagates the changes of the real variables to the pseudovariable. This can be done in time *O(i+r)* using the Intersected intervals collector from chapter 5.3.1. After this phase, the domain of the pseudovariable contains only feasible pseudovalues (rectangles). All rectangles, which were not supported in every real variable, were pruned. In the second phase, the algorithm uses the Interval list generator from chapter 5.4.1 to create new domains from the projections of feasible rectangles in time *O(r)*. For *a*-ary constraint, the worst-case time complexity of the propagator is *O(a(i+r))*.

hyper-rectangles with numbers	segments	sorted segments
5..7×1..4×2..6 1	*(5..7)-1 (1..4)-1 (2..6)-1*	*(2..8)-5 (0..2)-3 (0..0)-4*
4..4×2..3×7..9 2	*(4..4)-2 (2..3)-2 (7..9)-2*	*(3..6)-6 (1..1)-5 (2..6)-1*
5..9×0..2×6..9 3	*(5..9)-3 (0..2)-3 (6..9)-3*	*(4..4)-2 (1..4)-1 (3..3)-6*
4..6×5..6×0..0 4	*(4..6)-4 (5..6)-4 (0..0)-4*	*(4..6)-4 (2..3)-2 (6..9)-3*
2..8×1..1×8..9 5	*(2..8)-5 (1..1)-5 (8..9)-5*	*(5..7)-1 (2..5)-6 (7..9)-2*
3..6×2..5×3..3 6	*(3..6)-6 (2..5)-6 (3..3)-6*	*(5..9)-3 (5..6)-4 (8..9)-5*

Figure 21: data organization of hyper-rectangles

The algorithm uses the same principle as the algorithm by Barták and Mecl described in chapter 4.3.2: identify the feasible rectangles and build the new domains from their projections. The difference is that the algorithm breaks the list of hyper-rectangles into *a* lists in such way that each list contains the hyper-rectangles' projections into one dimension (middle section of figure 21).

Each of the lists is sorted by the starts of projected intervals (right section of figure 21). The “Intersected intervals collector” propagator uses the sorted list to detect all rectangles supported by the domain in a single pass (in a linear time). In the second phase, “Interval list generator” is used. Again, the fact that the lists of segments are sorted, allows us to create the output domains using a single pass in a linear time. This way the algorithm ceases the logarithmic factor the original Barták's algorithm has.

The difference of these two algorithms is in a granularity: Preforming individual queries on intersections of the domains and the projections can be seen as fine grained while one query generating all intersections is coarse grained.

6.2 Revision of the MDD

When describing the MDDC algorithm in chapter 4.2.3 we have regarded the algorithm coarse grained. The algorithm acts as a propagator that has domains of the variables as the input and produces the new domains. On the other hand, we will show that it works internally as a fine grained algorithm. Therefore we will later distinguish internal and external granularity of algorithms.

To refer the internal granularity of the MDD we will now show the possible internal structures. When describing the MDDC algorithm, we have mentioned the rules for a node to be feasible:

(A1) The node is reachable (by an oriented path) from the root only by the edges that has feasible labels

(A2) The leaf is reachable from the node also by the edges having feasible labels

In fact, these rules can be formed in a more local way. The rule A1 can be replaced by:

(B1) The node is connected with a feasible node in the previous layer using the edge with a feasible label.

The rule (A2) can be replaced analogically. These definitions are equivalent because iterative using of rule (B1) creates the path required by rule (A1). The rule (B1) is in fact a ternary constraint binding the index of the node in the previous layer, the index of the node in the current layer and the label of the edge. The constraint can be defined for example by a list of triples. The constraint can be defined for each layer of the MDD. All those ternary constraints then form the subconstraint network depicted in figure 22 that models the MDD. Variables V_i are real variables, while variables N_i store the node indices (variables N_0 representing nodes in the layer of the root and N_n representing nodes in the layer of the leaf have trivially singleton domains, therefore the top-most and the bottom-most level can be pruned by binary constraints). We can prune each of three variables bound by the constraint. Pruning the indices of nodes in the current layer applies the rule (B1) and pruning the indices of nodes in the previous layer corresponds to rule (B2). The third direction, pruning the labels of edges creates the domain of the real variable of the layer.

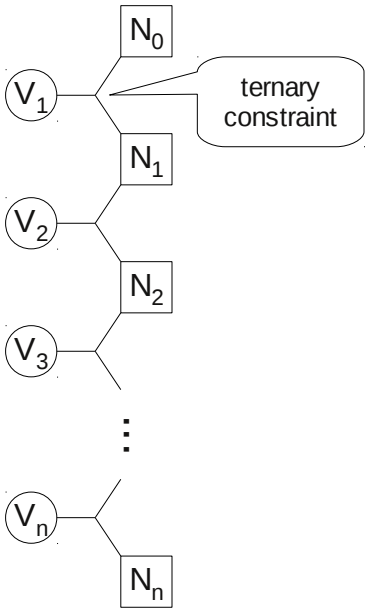


Figure 22: Model of the MDD using ternary constraints (via nodes)

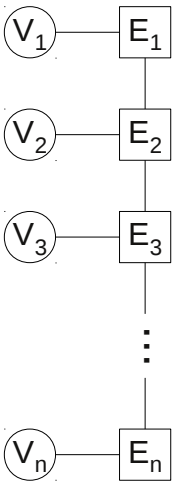


Figure 23: Model of the MDD using binary constraints (via edges)

Note that the ternary subconstraints can be used as well as binary although we have proven required theorems only for the binary constraints. On the other hand, there is an alternative modeling of MDD that uses binary constraints. As opposed to focusing on feasibility of nodes, the next binarization of MDD defines feasibility of edges. The edge is feasible if:

(C1) There is a neighboring[6] feasible edge in the previous layer

(C2) Label of the edge is feasible, meaning it is in the current domain of the variable respective to the edge's layer.

(C3) There is a neighboring feasible edge in the next layer

Rule (C1) and rule (C3) applied to a pair of consecutive layers represent two directions of the same binary constraint. The subconstraint network modeling the MDD is shown in figure 23.

From this perspective, the MDDC algorithm is internally fine grained because it treats the nodes individually. This view of MDD also gives us an idea, how to implement the filtering algorithm for the MDD. We might either choose to implement it in the way of some fine grained algorithm (for example AC-6) or as coarse grained using the appropriate propagators from chapter 5. The coarse grained approach is specially useful when implementing the SICStus case constraint – MDD with the edges labeled by intervals instead of the values. We can eliminate the logarithmic factor by using the same interval processing as we used for hyper-rectangles in the previous chapter.

To estimate the complexity of the filtering algorithm for MDD, we will denote *a* the arity, *d* the size of the largest initial domain and *f* the number of transitions of the smallest deterministic finite automaton defining the constraint domain. The domains of the real variables have size *O(d)* and the domains of the pseudovariables have size *O(f)*. The propagations between the real variables and the

6 Edges in consecutive layers are neighboring iff they share a node.

corresponding pseudovariables require single branch time $O(df)$ when using the Intersected intervals collector and the Interval list generator. The propagations between the pseudovariables require time $O(f^2)$ in a single branch of the search tree. In total for a layers, the single-branch worst-case time complexity of the proposed algorithm for MDD is $O(adf+af^2)$. In comparison, the original MDDC algorithm has the complexity of the same type $O(a^2df)$. The MDDC is worse of factor a because when one domain changes, the algorithm must traverse the whole MDD. On the other hand, the pseudovariables' algorithm can propagate locally by stopping at level of the MDD, where the domain of the pseudovariable does not change.

More precisely, the subconstraint propagator signals, whether the target domain was changed. For example the Builder Tree can detect this in the following way: the target domain was pruned if the index of the old tree's root is not equal to the new root index. When the target domain was pruned, the target pseudovariable is marked as dirty. The real variables are marked dirty if the domain has changed since the last superconstraint propagator call. Then the dirty flag is used to determine, whether it is necessary to call subconstraint propagator on given arc of the subconstraint network. If the source variable of the arc is not dirty then the target domain did not loose supports in the source domain. Of course, we assume that when the super constraint propagator starts, all domains of pseudovariables are initialized to their state after the previous superconstraint propagator call.

6.3 How to decompose constraints into subconstraint networks?

We have not discussed yet how to create the subconstraint network for a given constraint. We expect that this decision is made by the person that formalizes the problem into the CSP. The modeler has the same responsibility when choosing variable ordering in the SICStus's “case” constraint, which can also have great impact on the constraint propagation effectiveness[7]. First we will describe the algorithms creating the subconstraint network for a given constraint domain. The algorithms are too ineffective to be used in practice, except for cases when one constraint is reused many times in CSPs. On the other hand, the algorithms show the principles that the human modeler needs to understand to create efficient subconstraint network designs. Note that the modeler has typically more complex objective. While the algorithm is creating the network for the given constraint domain, the modeler typically designs the network for a whole class of constraint domains.

First we will define a criterion measuring the efficiency of the subconstraint network. In the chapter 5 we have shown algorithms for propagating discrete domains, continuous domains to discrete domains and discrete domains to continuous domains. These are all kinds of propagators we need when implementing superconstraint propagation over the subconstraint network. For each of these three kinds of propagations we have presented algorithms with single-branch worst-case

7 This issue is targeted for example in [11].

time complexity $O(d_S d_T)$. Therefore we define a score of an edge of the subconstraint network as a product of the initial domain sizes of variables represented by the nodes at both ends of the edge. The score of the whole network is the sum of scores of all the edges in the network. Then the score is related to the time complexity and therefore is a good measure of the subconstraint network efficiency.

6.3.1 A hill climbing algorithm

The presented algorithms will try to find a subconstraint network with the smallest score. One possible algorithm can find such a subconstraint network by computing the score of all possible networks. We will now present another algorithm that is more efficient but does not guarantee finding the optimal solution.

The algorithm starts with a star-shaped subconstraint network: The network has one pseudovariable and the pseudovariable is bound by binary subconstraint with each of the real variables (first network in figure 24). The algorithm performs the following operation: A pseudovariable having at least three children is selected. Then the algorithm chooses a subset of children of the selected node and connect the chosen children to a newly created parent node. The parent node will be a child of the selected node, so the chosen children become grandchildren of the selected node. We can form any subconstraint network by iterating this operation.

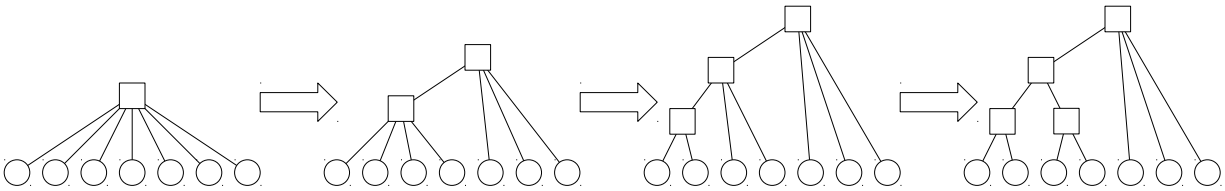


Figure 24: example process of the hill climbing algorithm

In order to determine, which node to select and which children to choose, the algorithm computes the score of all networks that can be results of all possible choices. The algorithm then commits to the best choice. The algorithm continues by examining the possible operations on the new network. The algorithm stops when there is no choice leading to a network with the score better than the current score.

6.3.2 A greedy algorithm

The previous algorithm needed to examine $O(2^a)$ choices in one iteration. The problem of the algorithm is that the computation of the score is time-expensive. For the given subconstraint network the algorithm must constitute the pseudovalues in order to count them. Consider a parent node having n children. In the worst case, the parent node values reflect n-tuples of the children's values. If possible without violating the rectangularity, the parent value can represent a group of such tuples. Such grouping is analogical to identifying isomorphic subtries when building the MDD.

Two conditions must hold to group the values:

1. The set of grouped values must be rectangular with respect to the children
2. Value's t-set projected to variables that are not in the subtree of the current node must be equal for each value that is being grouped.

The second rule ensures that the group will be rectangular with respect to the parent node too. This way we can constitute values of all pseudovariables up to the root. Also we can estimate scores for subtrees of the network by running this computation on a subtree only.

The following algorithm will work with a set of trees over the variables. Initially there are *n* trees each consisting of a single node representing one real variable (as depicted in figure 25). The algorithm will use two operations to join the trees, the operations are used until there is only one tree. The final tree will be the resulting subconstraint network. The operations are:

1. A new node is created and the roots of two trees become children of the new node.
2. The root of the tree becomes a child of the other existing root. We want real variables to be leafs in the subconstraint network, therefore the potential parent must represent a pseudovariable.

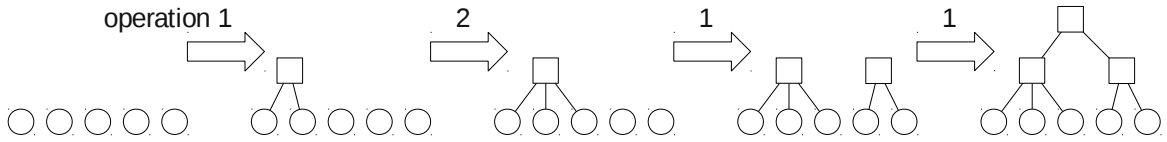


Figure 25: example process of the greedy algorithm

The algorithm must choose two trees to join. There are $O(a^2)$ choices, much less in comparison with the hill climbing algorithm. Moreover the greedy algorithm computes only partial score when deciding which choice to commit. Therefore the greedy algorithm is more efficient than the hill climbing algorithm. On the other hand, the resulting network may be more distant from the optimum when using the greedy algorithm.

As said at the beginning of this chapter, these algorithms will serve us as an inspiration to formulate the guidelines for creating subconstraint networks by human modelers. To describe the principles on examples we will introduce the problem class that was an inspiration for the theory in this thesis.

6.4 Planning problems

6.4.1 Introduction to planning

The problem class origin is in an artificial intelligence branch called planning. Goal of the planning is to find a sequence of actions (*plan*) transforming a world into a desired state. The world is defined by a set of *state variables*. An *initial state* of the world is described by values given to the state variables. A set of *actions* is defined to alter the world. Each action may have conditions and

effects defined for some of the state variables. An effect of the action sets a given value to a given state variable. If the action has no effect defined for a given state variable, the state variable keeps the value it had before the action application. The condition requires a given state variable to have a given value. The action can appear in the plan only if all conditions hold at the moment of the appearance. The planning problem definition also includes a goal, which is a set of conditions that must hold after applying the whole sequence of actions.

We can use constraint satisfaction to find a plan of a given length N[8]. Actions will be represented by variables $A_1, \ldots, A_N$ and state variables by $S_{0,1}, \ldots, S_{N,1}, S_{0,2}, \ldots, S_{N,2}, \ldots, S_{0,M}, \ldots, S_{N,M}$, where M is the number of state variables the world is described by. Action A_i checks its conditions in states $S_{i-1,1}, \ldots, S_{i-1,M}$ and effects of the action appear in states $S_{i,0}, \ldots, S_{i,M}$. We can formulate the key aspect of the planning, the state transition, as a $(2M+1)$-ary constraint on variables $S_{i-1,1}, \ldots, S_{i-1,M}, A_i, S_{i,1}, \ldots, S_{i,M}$. Alternatively we can use a set of ternary constraints on variables $S_{i-1,j}, A_i, S_{i,j}$ for $j \in \{1, ..., M\}$. We can describe the ternary constraints by logical formulas describing the relations of the preceding state, the action and the following state as stated in the previous paragraph. On the other hand, as shown by Barták and Toropila in [2], such formulas are so complex that constraint definition in extension leads to much effective propagation. Therefore we will generate the set of compatible triples for $(S_{i-1,j}, A_i, S_{i,j})$ from the planning problem. For action a having condition c and effect e we will put (c, a, e) into the constraint domain. If an effect is not defined, the corresponding triple is (c, a, c). If neither condition nor effect are defined for the action, triples (k, a, k) for each possible state value k are added. And finally if only effect is defined, we include triples (k, a, e) into the constraint domain.

The constraint domain C for the $(2M+1)$-ary state transition constraint is impracticably large to enumerate. The description of the constraint by a subconstraint network may have reasonable size even for large constraint domains. In our case, the size of the subconstraint network description will be comparable to the size of all ternary state transition constraints together. The question is the shape of the subconstraint network. The structure of the problem class suggests us that the subconstraint network should comprise of M similar parts. Let us put this suggestion aside and focus on more general criteria that might be helpful for modeling other problems.

6.4.2 Shaping the subconstraint network

Relatedness

Let us start by selecting two variables, which we will make siblings in the network. Consider selecting an old state and a new state of the same variable $S_{\_,j}$ as siblings. Their parent will (in the worst case) represent pairs of values of the siblings. An action having neither the condition nor the

8 The CSP formalism does not allow adaptable number of variables and constraints. When no plan is found for the given length, we formulate a new CSP for incremented length. Unlike the CSP formalism, the Gecode library (also SICStus, etc.) allows to create variables and constraints during the search.

effect on the variable $S_{\_,j}$ generates pairs *(k, k)* for each state value *k*. Some other pairs *(c, e)* are generated by actions having a condition *c* and an effect *e*. Typically, not all the possible pairs are present.

On the other hand, if we chose non-related variables (for example two old state variables or an old state and a new state of different planning state variables), typically all pairs would appear in the domain of the parent. For example let the children variables be new state variables $S_{i,1}$ and $S_{i,2}$ and we are trying to find a pair *(k, l)*. Typically there is an action that has neither the condition or the effect defined for both the state variables. Such action *a* allows triple *(k, a, k)* for variables $(S_{i-1,1}, A_i, S_{i,1})$ and triple *(l, a, l)* for variables $(S_{i-1,2}, A_i, S_{i,2})$. In the constraint domain, there is a tuple *(k, l, …, a, k, l, …)* for variables $(S_{i-1,1}, \ldots, S_{i-1,M}, A_i, S_{i,1}, \ldots, S_{i,M})$ respectively. The tuple has pair *(k, l)* on positions of variables $S_{i,1}$ and $S_{i,2}$.

If such non-related variables were chosen to be siblings, the parent domain would represent all the pairs. On the other hand, the parent domain would be smaller, when choosing the related pair of the old and the new state of the same state variable as siblings. Therefore the greedy algorithm would choose the related variables. As a general rule, the modeler should place more related variables nearer in the subconstraint network.

Independence

There is another view on the problem: While the pairs of the old and the new variable of the same state are bound by rules, the non-related variables are bound only indirectly via the action variable. If some variables were not bound at all, we say they are *independent*. Of course, there is no reason to bound independent variables by a constraint. When the dependency is defined as a quantity property, we might have a constraint in which some groups of variables would be more dependent and some groups would be less dependent. Then we should place the more dependent variables nearer in the subconstraint network and that way we separate the less dependent variables to distant parts of the networks.

To define the (in)dependency, let us have a constraint and Γ and Δ disjoint subsets of the constraint's scope. Let C/Γ be the constraint domain projected to the variables in Γ only; we define C/Δ and $C/(\Gamma\cup\Delta)$ analogically. If Γ and Δ are independent groups of variables, any partial tuple from C/Γ would be compatible with a partial tuple from C/Δ. There would be combination of those two partial tuples in $C/(\Gamma\cup\Delta)$. When comparing the sizes of those sets, equation
$|C/\Gamma|\,|C/\Delta| = |C/(\Gamma\cup\Delta)|$ holds for independent Γ and Δ. When Γ and Δ are more dependent, less combinations of the partial tuples are compatible and the set $|C/(\Gamma\cup\Delta)|$ is smaller. We can measure dependency by ratio

$$0 \leqslant \frac{|C/(\Gamma\cup\Delta)|}{|C/\Gamma||C/\Delta|} \leqslant 1$$

In fact, the children of the selected node in each iteration of the hill climbing algorithm are in the

role of *(Γ∪Δ)*. The algorithm searches for the best factoring of *(Γ∪Δ)* into *Γ* and *Δ*, using a criterion similar to the dependency.

Aside of the mathematical description, the dependency is intuitive. The human modeler should have sense of which variables are less and which are more dependent and model the subconstraint network accordingly.

6.4.3 The resulting subconstraint network for planning

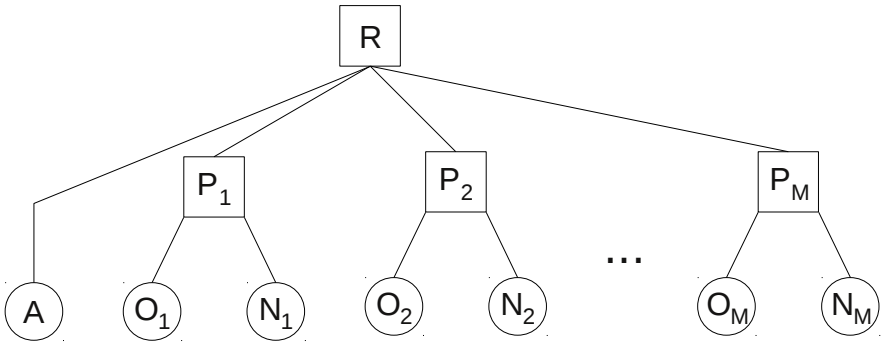


Figure 26: The subconstraint network for (2M+1)-ary state-transition constraint in planning

Applying the above principles, we create a subconstraint networks for the *(2M+1)*-ary state transition constraint. The scheme of the network is shown in figure 26. The node “A” represents the action variable, “O_i” and “N_i” nodes represent the old state and the new state of the i-th planning variable. The pseudovariable “P_i” is created to group “O_i” and “N_i” because they are strongly related. Other pairs of variables are rather independent, therefore no other pseudovariable is introduced and the variables are connected only via the root (denoted “R”).

7 Experimental evaluation

In this chapter, we will present the results of experiments. The experiments are divided into the following two chapters. The first group of experiments will focus on evaluating the concept of the pseudovariables itself. Then we will present the experiments comparing the effects of using different subconstraint propagators described in chapter 5.

All experiments ran on a 64-bit Intel Core 2 Duo 1.6GHz CPU with 2MB L2 cache and 2GB operating memory. The computer runs the Linux 2.6 OS kernel and the user-space[9] time of the process is measured. The experimental implementation, called *genesis*, is implemented in C++ using the Gecode[10] constraint library. The genesis is compiled using the GCC 4.4.3 with "−O2" optimization option.

The inspiration for our research was solving the planning problems. Therefore we measure efficiency of solving the problems in the experiments. We compare implementations of the state transition that is the essential aspect of the planning. Besides of that, the genesis solver also implements some auxiliary planning techniques. The first technique is relevance of action at the last (goal) layer. The last action of the plan must have effect that is needed to accomplish the goal. The actions that does not have relevant effects are excluded from the last layer by an unary constraint. The second technique is used to eliminate some symmetrical solutions. We say that actions *Act1* and *Act2* are independent if sequences *(…, Act1, Act2, …)* are exchangeable with *(…, Act2, Act1, …)* in any plan. The actions are independent if they have no conflicted conditions and effects and effect of one action is not needed by condition of the other action. To narrow the search space, we introduce constraints allowing only one of the orderings of the independent actions (actually we implement unidirectional version relying on allovance instead of independence). The action variables and the state variables are labeled together using the "dom/deg" labeling strategy.

7.1 The main experiments

7.1.1 Compared approaches

In this chapter we will focus on solving the planning problem in different ways. One approach uses the *2M+1* dimensional complex state-transition constraint modeled by the subconstraint network in figure 26. Another approach uses the set of the 3-dimensional single-state transition constraints, each of which binds the action, the old state and the new state of one planning state variable. We model the ternary constraint by two different propagators: In the first case, we use the

9 The time the process spent on the CPU, except for OS kernel calls (reading files, memory allocation, etc.). The time is reported by the OS kernel module that is responsible for allocating the CPU to the process.

10 http://www.gecode.org/

hyper-rectangle superconstraint from chapter 6.1. The second model of the ternary constraint uses the Gecode's standard extensional constraint DFA, implementation of which is similar to the MDD algorithm described in the chapter 4.2.3.

The first two approaches uses the concept of the subconstraint network, therefore various subconstraint propagators (from chapter 5) can be used. We will evaluate the individual subconstraint propagators in the next chapter. In this chapter, we evaluate the “global” approaches. Therefore we use the best combination of the subconstraint propagators: We choose “Open-Close” to propagate the real variables to the pseudovariables and the “Interval List Generator” for the other direction of propagation. Moreover, the *(2M+1)*-ary constraint also contains subconstraints binding pseudovariables together. We have two options for this kind of subconstraints: the AC-3.1 propagator and the Tree Builder. We use the Tree Builder for the upward propagation ($P_i \rightarrow R$ in figure 26) and the AC-3.1 for the downward propagation ($R \rightarrow P_i$).

Unfortunately, single propagations run too short to measure. To compare the above approaches we will measure the runtime of the complete search for a solution. Besides the propagator calls, there are other computations made by the solver. We run the experiments with the same parameters of the search. The evaluated propagators are all AC-sound and AC-complete. Therefore all the propagators are equivalent and the same values are pruned at the same phases of all the experiments. In other words, the search trees are identical. Therefore all computations except for the propagations are done identically and all differences in measured times are caused by propagations.

There is another reason for measuring the time required by the complete search: We are not comparing the propagation algorithms only, but also the structures of the CSP (*(2M+1)*-ary versus ternary constraints). Evaluating the propagation of the *(2M+1)*-ary constraint in comparison with the set of ternary constraints is possible at the level of a single propagation. On the other hand, this fact justifies making the comparison at level of the complete search.

The experiments consist of running the solver variants on the set of 86 planning problems. The problems were selected from the International Planning Competition. The set of planning domains is the same as in the evaluation of findings in [2] (which will be compared with the genesis later). The planning domains are: Gripper, Logistics, Mystery (from IPC 1), Blocks, Elevator (IPC 2), Depots, DriverLog, Zenotravel (IPC 3), Airport, PSR (IPC 4), Pipesworld, Rovers and TPP (IPC 5). Each problem is solved 10 times and the average times are then compared. Problems that are not solved within 1000 seconds are ignored.

7.1.2 Implementation of the ternary constraint: The hyper-rectangle versus the Gecode's original propagator

In the previous chapter we have presented three approaches that we will compare. In fact, the hyper-rectangle approach is a midway milestone in an evolution from the Gecode's original propagator approach to the complex state-transition constraint. Therefore we will divide this

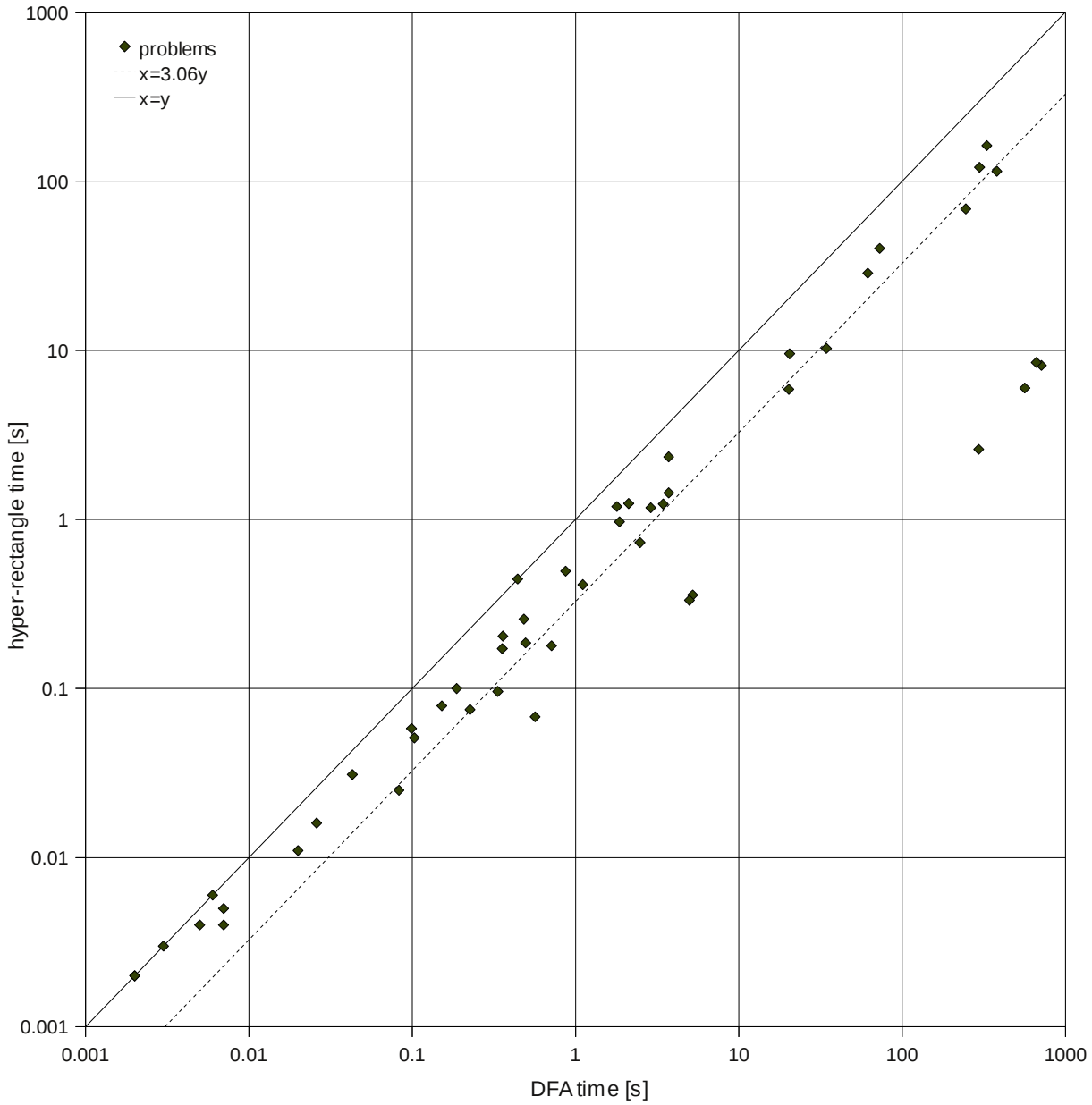


Figure 27: Comparison of the GECODE's DFA constraint and the hyper-rectangle constraint

evolution into two steps and evaluate the impact of the steps individually.

Figure 27 presents the improvements induced by the first step. Only 50 problems were solved using both the compared approaches within the 1000-second time limit. The chart represents 10-iteration averages of runtimes on the problems. The dashed line $x = 3.06y$ represents the geometric-mean ratio of the compared times.

To interpret the results we will describe the differences in the compared approaches. The main difference is in the internal granularity of the propagators: The GECODE's DFA propagator works in a way similar to the MDDC propagator and therefore the propagator is internally fine grained. On

the other hand, the hyper-rectangle propagator is internally coarse grained as shown in chapter 6.1. For example, the hyper-rectangle propagator uses the Open-Close algorithm exploiting the bit parallelism.

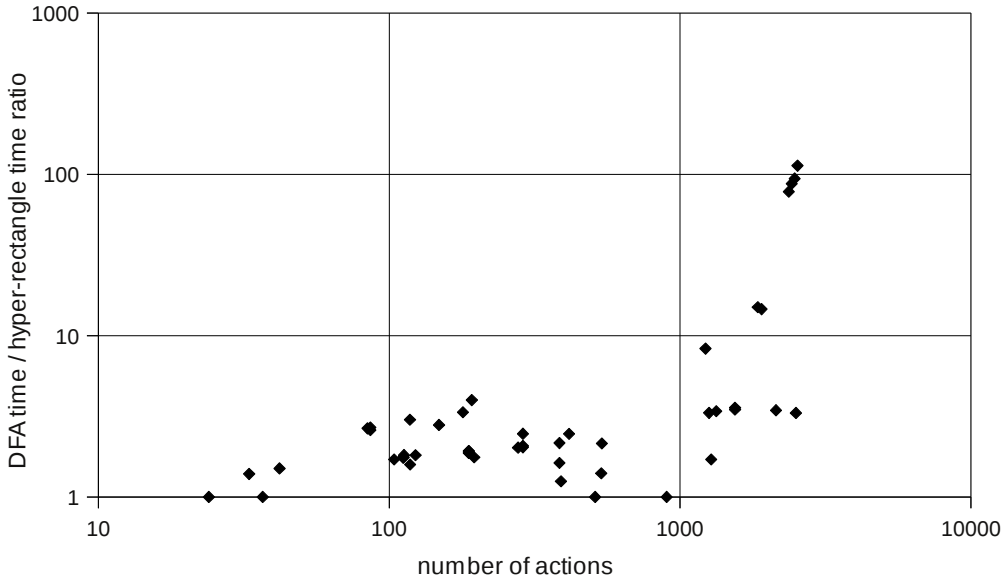


Figure 28: dependency of the time improvement on the problem complexity

Figure 28 shows the dependency of the time ratio on the problem complexity. The complexity is measured by the number of actions the planning problem has. The number of actions is used because it is the size of the largest initial domain. We can see that the hyper-rectangle propagator improves the total solver's time more for the more complex planning problems. This result is unexpected and we will focus more deeply on the differences of the compared approaches in order to interpret the results.

The (Gecode's original) DFA propagator uses the sparse matrix data structure to store the set of pruned nodes in the MDD. Cheng and Yap recommend in [12] usage of the sparse matrix structure instead of the bit array because the sparse matrix can be restored in a constant time upon backtracking. The bit array that is used by the hyper-rectangle propagator needs a linear time for restoration. Despite of that, the results show that the benefits of the bit parallelism overbalance this disadvantage. Note that the recommendation in [12] may be valid for the MDDC algorithm because it is internally fine grained and thus no bit parallelism is conceivable.

We should also note that we have used the hyper-rectangle algorithm with the Open-Close propagator. We have discussed the Open-Close complexity in chapter 5.3.3. We have stated that the algorithm is asymptotically worse than the Intersected Intervals Collector (chapter 5.3.1) and therefore it should be less efficient for greater domains. This is the second reason why the trend in figure 28 is surprising. We don't expect the time ratio to rise for even more complex planning problems.

7.1.3 Comparing the single-state transition and the complex state transition constraint

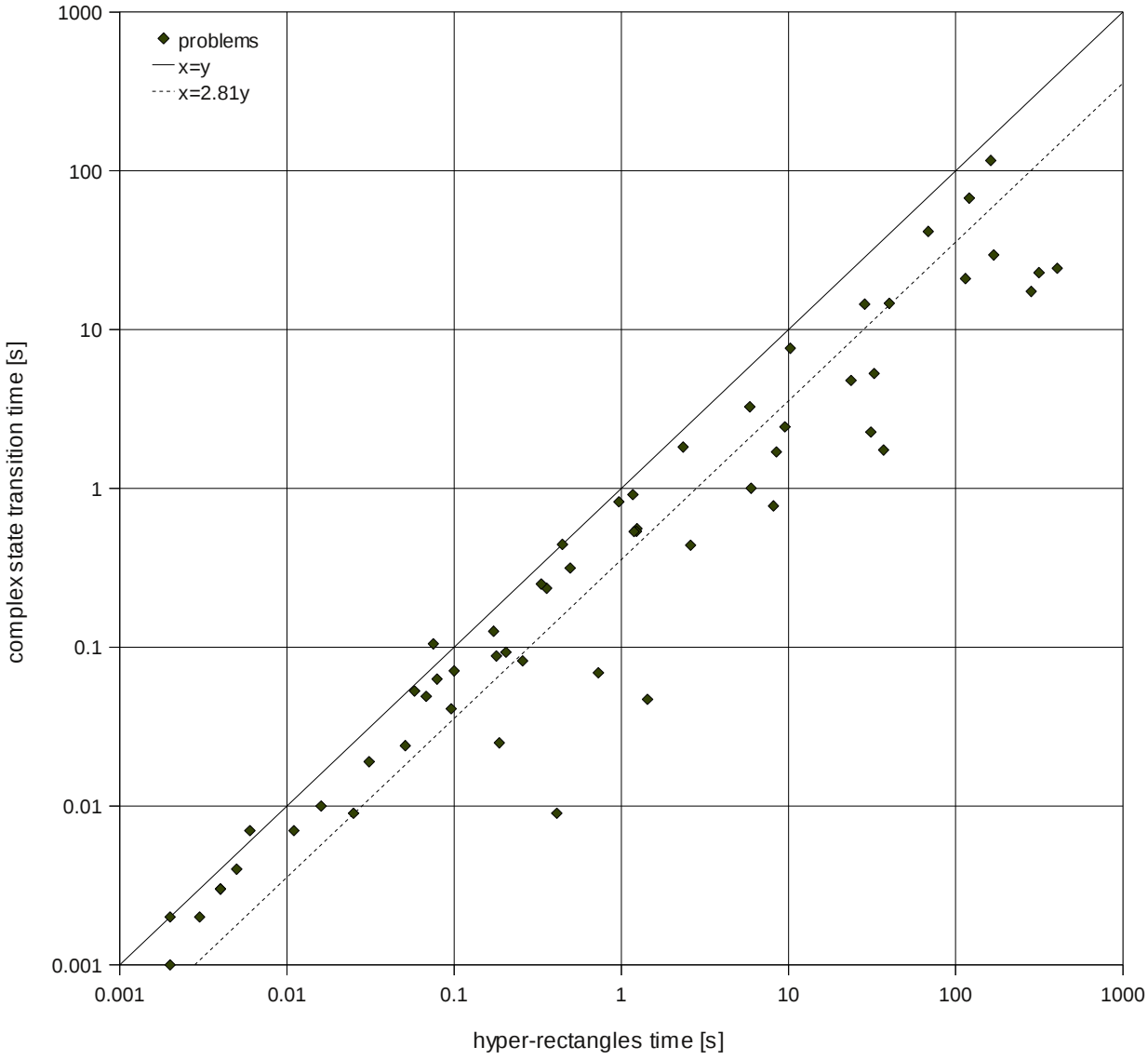


Figure 29: Comparison of the single state and the complex state transition constraint approach

After evaluating the first step of the evolution, we will now focus on the second step. The second step groups the set of ternary single-state transition constraints into one *(2M+1)*-ary complex state-transition constraint (depicted in figure 26). As the previous chart, the chart on figure 29 shows the relation of 10-iteration averages of the runtimes. The chart displays 56 problems that were solved within the time limit by the both compared solvers. Again, the dashed line *x=2.81y* represents a geometric mean of the time ratios.

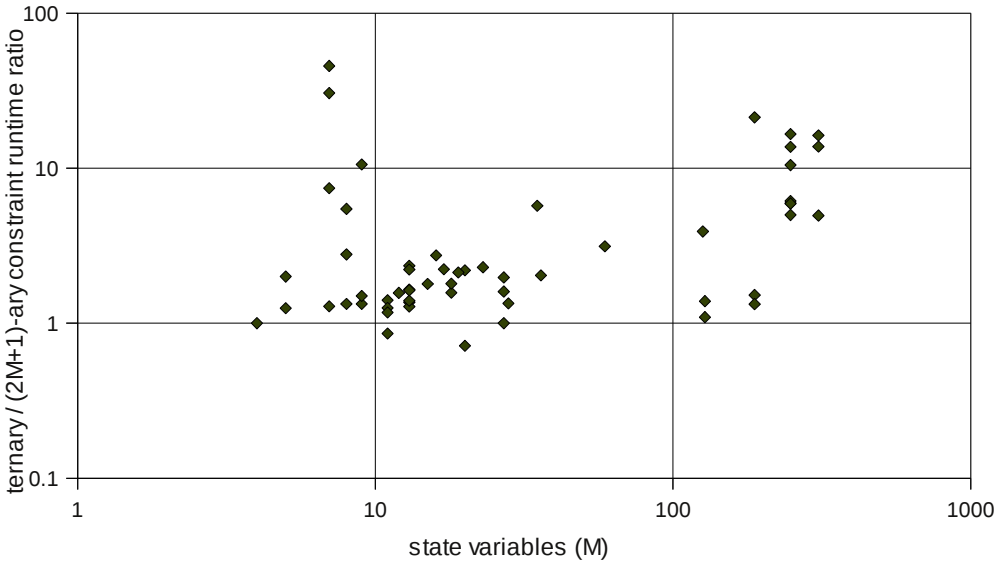


Figure 30: dependency of the time improvement on the problem complexity

We will now analyze the relation of the improvement time and the problem complexity. We define the problem complexity as the number of the state variables the planning problem has. Denoting this number *M*, we are comparing usage of *M* ternary constraints versus usage of single *(2M+1)*-ary constraint. Therefore using *M* to define the problem complexity is a good choice for analyzing the improvement time ratio.

The hypothesis is that the effect of replacing *M* ternary constraints by single high-arity constraint will be more significant for larger *M*. Results of the experiment are shown in figure 30. Unfortunately, although some trend can be seen in the chart, no statistically significant conclusion can be made from the results of the experiment.

There are various reasons why the hypothesis was not supported by the experiment. One of the possible reasons is the distribution of the problems. The problems represent 14 families of planning problems. Each of the families has a particular characteristics and the selection of the problems may not be representative. Another possible cause is measuring the total search time instead of the runtime of a single propagator call. And also the hypothesis may not have been supported by the experiments simply because the hypothesis is not valid.

7.1.4 Comparison with an external solver

The motivation for this thesis came from the paper of Barták and Toropila [2] studying formulation of planning problems into a CSP. For comparison, we also run the solver called SeP that Barták and Toropila developed to evaluate their findings in the paper. The paper and this thesis studies different problems: The paper was focusing on formulating the planning problem into the CSP. The authors have developed and implemented many techniques that concerns more the planning than the constraint satisfaction. The SeP solver was implemented in the SICStus Prolog.

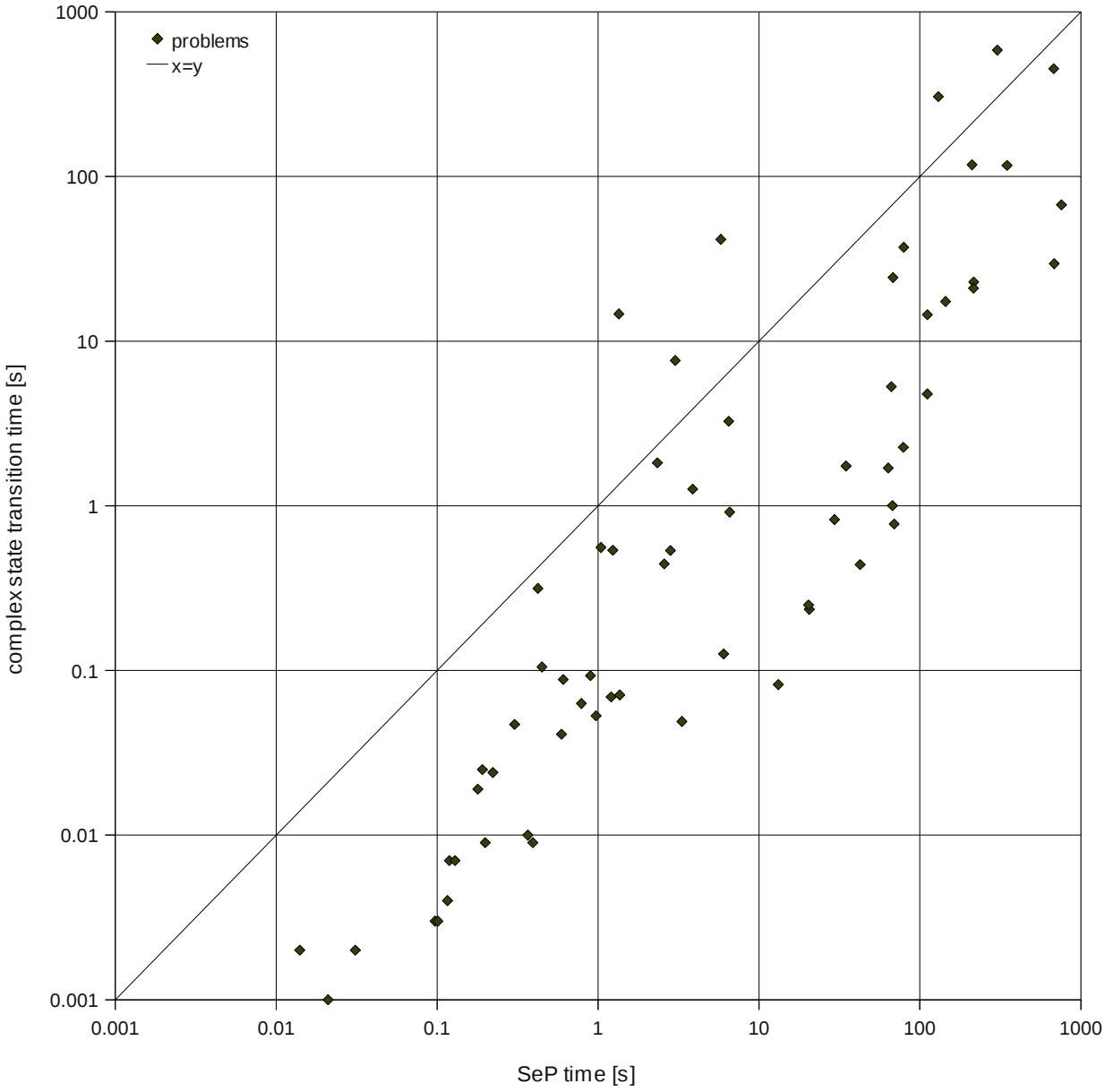


Figure 31: comparison of runtimes of SeP and the best variant of genesis

On the other hand, this thesis focuses on the implementation of the AC filtering algorithms and implements only a part of the planning-originated techniques (breaking symmetries using action allovance, action relevance at the goal layer). Moreover, our experimental solver is implemented in the C++. Therefore the SeP and the genesis are so distant that their comparison cannot conclude into any scientific results. We use this comparison only to decide, whether this thesis brought an actual improvement for solving the planning problems.

The SeP solved 67 problems within the 1000 second time limit while the genesis solved 65

problems. Runtimes (10-iteration average) of 63 problems that were solved by both solvers are shown in figure 31. The genesis outperforms the SeP solver on about 87% of the problems. On the other hand, we can see that the dominance of genesis is clear for simple problems but less significant for harder problems. We presume that the genesis solver efficiently implements the propagation. On the other hand, the SeP profits of implementation of the planning-related techniques when solving the harder problems.

7.2 Evaluation of the elementary propagators

In chapter 5 we have described several binary propagators. The propagators differ in forms of the source and the target domain the propagators work with. There are three kinds of propagators: bit-array → bit-array, list-of-intervals → bit-array and bit-array → list-of-intervals. Algorithms of the same kind are exchangeable. We have presented two propagators of kind "bit-array → bit-array": the AC-3.1 propagator and the Tree builder. In the category "list-of-intervals → bit-array" we have proposed the Intersected Intervals Collector propagator and the Open-Close propagator. We will now evaluate these propagators.

First we will focus on the AC-3.1 propagator and the Tree builder. These propagators expect both the source and the target domain to be in the bit-array form. Therefore they are suitable for the subconstraints between the "P_i" and the "R" node in figure 26. There are two directions of the propagation: the upward $P_i \rightarrow R$ and the downward $R \rightarrow P_i$. While the initial domain of the "R" may contain thousands of values, the "P_i" has typically the number of values is the order of tens. Recall our notation denoting d_S the size of the source domain and d_T the size of the target domain. The Tree builder time complexity, which in $O(d_S^2 + d_T d_S \log d_S)$ in a single branch, is much more sensitive on the source domain size than the target domain size. Therefore using the Tree builder in the downward direction, where d_S would be 1000, is inefficient. On the other hand, in the upward direction is d_S relatively small and d_T is large. In this case, the Tree builder may be competitive with the AC-3.1 propagator that has the time complexity $O(d_S d_T)$.

We have tested all four variants of the propagator usage. The most efficient is the one implementing the downward propagation using the AC-3.1 and the upward propagation by the Tree builder. We will call this variant "AB". The "AA" variant uses the AC-3.1 propagator for both the upward and the downward direction and the "BB" variant uses the Tree builder propagator for both directions. The last variant ("BA") uses the Tree builder for the downward propagation and the AC-3.1 propagator for the upward direction. Chart 3 in figure 32 shows the average of runtimes of these variants normalized to the best "AB" variant. Chart 1 shows the "AA" to "AB" runtime ratio depending on the domain size of the "R" pseudovariable. Analogically chart 2 shows the runtimes of the "BB" variant.

The second choice is whether to use the Intersected intervals collector (IIC) or the Open-Close propagator. These propagators expect the source domain in the list-of-intervals form and produce

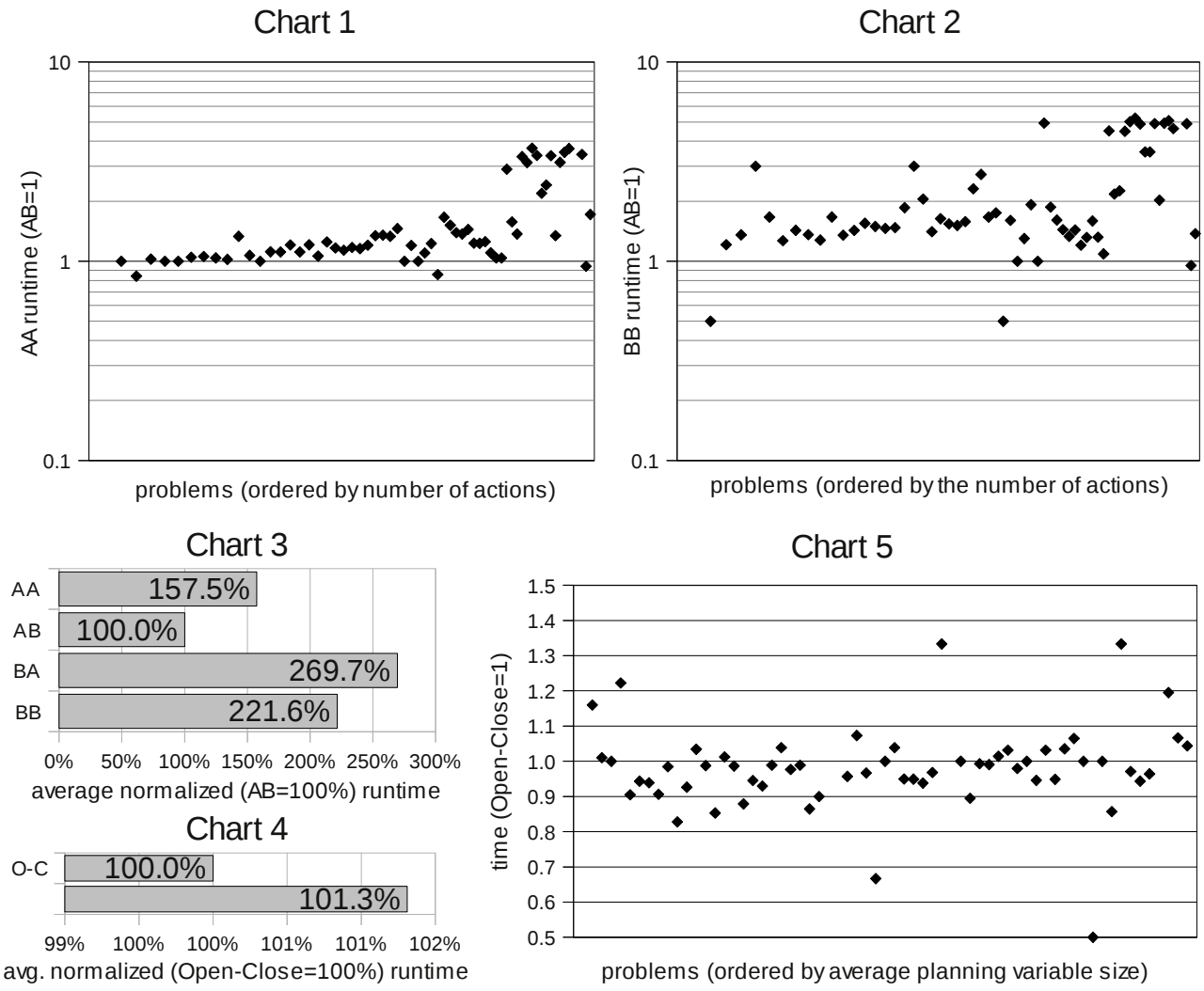


Figure 32: Results of the experiments comparing the propagators

the target domain in the bit-array form. Therefore we are using one of these propagators on arcs "O_i"→"P_i" and "N_i"→"P_i" (of the subconstraint network depicted in figure 26). The arc "A"→"R" is also of this kind, but in fact the arc represents an identity constraint, therefore it is handled in a different way.

Chart 5 shows the runtime ratio of the IIC and the Open-Close propagator. The x-axis of the chart represents problems sorted by their average "O_i" and "N_i" domain size increasingly. Both the IIC and the Open-Close performed well in the experiments and it is undecidable, which of the propagators is more efficient.

8 Conclusion

In this thesis we have studied an implementation of filtering algorithms for the constraints defined in extension. In order to represent the constraint domain, we have proposed a new concept of binarization. The constraint domain is decomposed into values of the newly created variables. The concept is so general, that many of the existing algorithms can be described in terms of the concept. Moreover, the concept can be used also for constraints that cannot be effectively expressed by the current techniques such as MDD. For example, the *(2M+1)*-ary complex state-transition constraint cannot be efficiently handled by the existing filtering algorithms. The set of ternary single-state transition constraints must be used instead. The experiments showed that such replacement results in a loss of efficiency.

Another contribution of the binarization approach is expressed by the internal granularity. We can describe an internal structure of the existing algorithms in the perspective of our concept. The values of the revealed structure are typically pruned individually, in a fine-grained way. Using the coarse-grained propagation for the internal-structure constraints is more efficient, as the experiments show. Moreover, when handling the domains in the list-of-intervals form, the coarse-grained approach also leads to an asymptotic improvement (of a logarithmic factor).

The requirements on the binarization were shown in the developed theory. We have presented guidelines for creating the internal structure by a human modeler. Also the algorithms for an automated modeling were presented. Unfortunately, the algorithms are more costly than the propagation itself, therefore they are useful in special situations only.

We have presented several elementary propagators that can be used on the internal constraints. As the experiments show, each of the propagators is useful for particular constraints. For example, the Tree Builder is more efficient than the AC-3.1 propagator when the source domain is relatively small. This is typical in the upward phase of the propagation of the complex constraint. On the other hand, the higher nodes of the constraint's internal structure have greater domains, therefore the AC-3.1 propagator is more suitable in the downward phase. The suitable propagators can be chosen not only when modeling the constraint structure, but also the solver can choose the suitable propagator at runtime.

At last, our findings contribute to efficiency of solving the planning problems, which was the motivation for our research.

References

[1] Barták R., Mecl R.: Implementing Propagators for Tabular Constraints, CSCLP 2003 (2004) pp. 44-65.

[2] Barták R., Toropila D.: Reformulating Constraint Models for Classical Planning, Proceedings of the 21st International Florida AI Research Society Conference (FLAIRS 2008) (2008) pp. 525-530.

[3] Beldiceanu N., Carlsson M.: Sweep as Generic Pruning Technique Applied to the Non-overlapping Rectangles Constraint, CP 2001 (2001) pp. 377-391.

[4] Bessiere C., Freuder E., Regin J.: Using inference to reduce arc consistency computation., IJCAI'95 (1995) pp. 592-598.

[5] Bessière Ch., Régin J.: Arc Consistency for General Constraint Networks: preliminary results, International Joint Conference on Artificial Intelligence (1997) pp. 398-404.

[6] Blumer A., Blumer J., Haussler D., Ehrenfeucht A., Chen M. T., Seiferas J.: The smallest automaton recognizing the subwords of a text, Theoretical Computer Science 40 (1985) pp. 31-55.

[7] Briggs P., Torczon L.: An Efficient Representation of Sparse Sets, ACM Letters on Programming Languages and Systems 2(1–4) (1993) pp. 59-69.

[8] C. Besière, M. Cordier: Arc-Consistency and Arc-Consistency Again, Artificial Intelligence 65 (1994) pp. 179-190.

[9] Carlsson M.: Filtering for the case Constraint. Talk given at Advanced School on Global Constraints (2006).

[10] Cheng Ch., Lee J., Stuckey P.: Box Constraint Collections for Adhoc Constraints, CP 2003 (2003) pp. 214-228.

[11] Cheng K., Yap R.: Applying ad-hoc global constraints with the case constraint to Still-life, Constraint 11(2-3) (2006) pp. 91-114.

[12] Cheng K., Yap R.: Maintaining Generalized Arc Consistency on Adhoc r-Ary Constraints, CP 2008 (2008) pp. 509-523.

[13] Dechter R., Pearl J.: Network-based heuristics for constraint satisfaction problems, Artificial Intelligence 34 (1988) pp. 1-38.

[14] Dechter R., Pearl J.: Tree clustering for constraint networks, Artificial Intelligence 38 (1989) pp. 353-366.

[15] Fredkin E.: Trie Memory, Communications of the ACM, v.3 n.9 (1960) pp. 490-499.

[16] Gent I., Jefferson C., Miguel I., Nightingale P.: Data structures for generalized arc consistency for extensional constraints, National Conference on Artificial Intelligence (2007) pp. 191-197.

[17] Lecoutre Ch., Vion J.: Enforcing Arc Consistency Using Bitwise Operators, Constraint Programming Letters 2 (2008) pp. 21-35.

[18] Mackworth A.: Consistency in Networks of Relations, Artificial Intelligence 8 (1977) pp. 99-118.

[19] Mohr R., Henderson T.: Arc and Path Consistency Revised, Artificial Intelligence 28 (1986) pp. 225-233.

[20] Pesant G.: A Regular Language Membership Constraint for Sequences of Variables, International Conference on Principles and Practice of Constraint Programming (2004) pp. 482-495.

[21] Rossi F., Petrie C., Dhar V.: On the equivalence of constraint satisfaction problems, Technical Report ACT-AI-222-89, MCC, Austin, Texas (1989).

[22] Y. Zhang, R. Yap: Making AC-3 an Optimal Algorithm, In Proceedings of IJCAI-01 (2001) pp. 316-321.

Appendix A: Notation in complexity estimates

Symbols used in complexity estimates are defined in different parts of the thesis. We will recapitulate their meaning here

Measures of the problem

e	Number of constraints in the problem (edges in the constraint network)

Measures of constraints

a	Constraint's arity
t	Size of the constraint domain. Number of tuples
g	Number of segments the constraint domain can be broken in. See chapter 5.3.1 for definition
r	Number of (hyper)rectangles. Size of the smallest set of rectangles that covers the constraint domain
b	Number of boxes in box collection constraint. See chapter 4.3.4 for definition

Measures of domains and sets

d	Size of the initial domain.
c	Size of the current domain (set). Count of elements
i	Number of intervals (maximal with respect to inclusion) the current domain consist of. Size of list-of-intervals representation

Indices for variables of unidirectional binary propagation

$d_S c_S i_S$	Domain of the source variable, the domain that is left intact by the propagator
$d_T c_T i_T$	Domain of the target variable, the domain that is being pruned by the propagator