Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Information Sciences 177 (2007) 3639–3678
INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL
www.elsevier.com/locate/ins

Partition search for non-binary constraint satisfaction

Julian R. Ullmann *

Department of Computer Science, King's College London, Strand, London WC2R 2LS, United Kingdom

Received 30 August 2005; received in revised form 8 March 2007; accepted 31 March 2007

Abstract

Previous algorithms for unrestricted constraint satisfaction use reduction search, which inferentially removes values from domains in order to prune the backtrack search tree. This paper introduces partition search, which uses an efficient join mechanism instead of removing values from domains. Analytical prediction of quantitative performance of partition search appears to be intractable and evaluation therefore has to be by experimental comparison with reduction search algorithms that represent the state of the art. Instead of working only with available reduction search algorithms, this paper introduces enhancements such as semijoin reduction preprocessing using Bloom filtering.
© 2007 Elsevier Inc. All rights reserved.

Keywords: Constraint satisfaction; Non-binary constraints; Hash join; Multijoin; Decision diagram; BDD; Consistency; Forward checking; Semijoin reduction; Dual search; Partition search; Random problems

1. Introduction

This paper uses database techniques to solve constraint satisfaction problems that are specified as follows. We are given a set $V = \{V_1, \ldots, V_i, \ldots, V_N\}$ of N discrete variables such that the domain of possible values of V_i is a given set D_i. We are also given a set $\Sigma = \{S_1, \ldots, S_j, \ldots, S_q\}$ of q constraint *scopes* that are subsets of V. Constraint scope S_j comprises n_j variables; $S_j = \{V_{j_1}, \ldots, V_{j_{n_j}}\}$. Moreover, for each constraint scope S_j we are given a constraint relation R_j. The constraint relation R_j is a set of n_j-tuples, each comprising one value for each of $\{V_{j_1}, \ldots, V_{j_{n_j}}\}$. The *arity* of R_j is n_j. Constraint R_j is said to be *tight* if its cardinality is small in comparison with its greatest possible cardinality.

A solution is an N-tuple $z = \langle v_1, \ldots, v_i, \ldots, v_N \rangle$ that comprises one value for each variable in the given set V of N variables and belongs to the set

$$Z = \{z | (\forall_{1 \leqslant j \leqslant q} j)(z[S_j] \in R_j) \wedge (\forall_{1 \leqslant i \leqslant N} i)(z[V_i] \in D_i)\}$$

where square brackets denote restriction [74]; so $z[S_j]$ is the subset of z comprising values of variables that belong to S_j. Enumerating Z is a constraint satisfaction problem. Finding a single member of Z, or reporting

* Tel.: +44 2082978746.
E-mail address: jrullmann@acm.org

0020-0255/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2007.03.030

that Z is empty, is a slightly simpler constraint satisfaction problem. If $n_j > 2$ for any j the problem is said to be *non-binary*.

These problems are well known to be NP-hard. The present paper does not apply restrictions that ensure tractability (e.g. [20]) and instead we accept that exponential time complexity will limit the range of practical problems that can be solved reasonably quickly.

An example of a practical application is as follows. Successive characters in a line of text can be regarded as variables that can have values ‘*A*’, ‘*B*’, ... An optical character recognition system is required to assign one such value to each successive character. However, when the print or writing is sufficiently poor, a recognition system may not be able to make an unequivocal decision and may instead decide, for example, “this character is either ‘*Q*’ or ‘*O*’ or ‘*G*’ ”. In this case the output of the recognition system is a sequence of sets such as {‘*Q*’, ‘*O*’, ‘*G*’}. Using relational projections of a dictionary as constraint relations, we can use constraint satisfaction technology to find combinations that may possibly be legitimate words [77].

Available algorithms for unrestricted non-binary constraint satisfaction employ reduction search [15,24,67]. *Reduction search* algorithms combine domain reduction and/or constraint relation reduction with backtrack search. *Domain reduction* means removing from any domain D_i any value that cannot possibly be in $\pi_{V_i}(Z)$. *Constraint relation reduction* means removing from any constraint relation R_j any n_j-tuple that cannot possibly be in $\pi_{S_j}(Z)$.

The main business of this paper is the introduction of an alternative approach using join-processing techniques. A basic property of the natural join operation is [74]:

$$R_r \bowtie R_s = \{t | t \text{ is a tuple on } S_r \cup S_s \text{ and } t[S_r] \in R_r \text{ and } t[S_s] \in R_s\}$$

whence the set Z of solutions can be written [38,83]

$$Z = R_1 \bowtie R_2 \bowtie \cdots \bowtie R_q$$

and Z can be enumerated by:

$Z := R_1$; **for** $j := 2$ **to** q **do** $Z := Z \bowtie R_j$ **end for**;

Partition search is a depth-first implementation of this simple routine. Partition search may be of interest because

- Within a limited range of problems it is very much faster than reduction search.
- It differs radically from reduction search in that it does not remove anything from domains or constraint relations.
- It is simple.

Because it does not prune the backtrack search tree by removing values or tuples, partition search may appear to be very foreign from the viewpoint of the constraint satisfaction literature. But partition search is not at all surprising from the viewpoint of database implementation.

Section 5 of this paper reports experimental comparison between partition and reduction search. In Section 2, domain reduction search is briefly introduced from first principles, to make this paper reasonably self-contained and to provide a clear starting point for developments. Section 2 introduces simple tabular reduction which is an algorithm that makes reduction search more competitive than when the benchmark non-binary reduction search algorithm GAC2001 [14] is used.

The substantial literature on reduction search almost universally makes central use of the concept of *arc consistency* [24,52] and developments thereof, e.g. [23,63]. This may be helpful with binary constraints but it is certainly not necessary in the present context. Section 2 achieves unusual simplicity by omitting arc consistency. In Section 3, instead of dual arc consistency [9,67], we equivalently [86] work with primal semijoin reduction, as in [10,32,50,61].

Within the range of problems where partition search performs most strongly, the performance of reduction search can be enhanced by constraint relation reduction preprocessing. This preprocessing can greatly improve the competitiveness of reduction search against partition search. For this reason, Section 3 briefly introduces

constraint relation reduction from first principles and describes its efficient implementation by application of Bloom filtering [5,61].

Section 4 introduces partition search, its implementation using hash joins [33,58], and also a different implementation using decision diagrams [84]. Furthermore, Section 4 explores the relationship between partition search and dual search. Dual search [9,27,67] instantiates scopes, whereas partition search instantiates subscopes which in some cases consist of only one variable. In this sense, dual search does, and partition search may, instantiate tuples. A radical difference between dual and partition search is that partition search is not a reduction search process. Moreover, partition search does not work with pairwise dual constraints. Partition search has more in common with depth-first multijoins in [68] than with dual search.

2. Domain reduction

2.1. Removal of values from domains

We define Σ_i to be the set of scopes in Σ that include the variable V_i; thus $\Sigma_i = \{S_j \in \Sigma | V_i \in S_j\}$. Assuming that every variable belongs to at least one scope, the definition of Z can be rewritten:

$$\begin{aligned} Z &= \{z|(\forall_{1\leqslant j\leqslant q} j)(z[S_j] \in R_j \wedge (\forall_{1\leqslant k\leqslant n_j} k)(z[V_{j_k}] \in D_{j_k}))\} \\ &= \{z|(\forall_{1\leqslant j\leqslant q} j)(z[S_j] \in R_j \wedge z[S_j] \in (D_{j_1} \times D_{j_2} \times \cdots \times D_{j_{n_j}}))\} \\ &= \{z|(\forall_{1\leqslant j\leqslant q} j)(z[S_j] \in (R_j \cap (D_{j_1} \times D_{j_2} \times \cdots \times D_{j_{n_j}})))\} \end{aligned}$$

From Theorem 1, in the Appendix, the set Z remains unchanged if domain D_i is replaced by

$$\cap_{S_j \in \Sigma_i}(\pi_{V_i}(R_j \cap (D_{j_1} \times D_{j_2} \times \cdots \times D_{j_{n_j}})))$$

Because D_i is one of the domains in this Cartesian product, replacement cannot increase the number of values in domain D_i. However, replacement may reduce the number of values in D_i and may thereby enable deletion of values from further domains. A detailed example of propagation of deletion is shown in [75, Fig. 3]. If, as a result of domain reduction, that is, deletion of values from domains, any domain becomes empty, then Z is empty.

2.2. A reduction search algorithm

Instantiation of a variable V_i can be regarded as the removal of all except one of the values that is currently in D_i. A reduction search algorithm [15,40,41,57,76,77,81] instantiates a variable, applies domain reduction, then perhaps instantiates another variable, again applies domain reduction, and so on. Eventually a set W of variables have been instantiated to w such that if $V_k \in W$ then V_k has been instantiated to a value $w[V_k]$ in domain D_k. At the time when W has been instantiated to w the set Z' of N-tuples that satisfy the constraints is:

$$Z' = \{z|(\forall_{1\leqslant j\leqslant q} j)(z[S_j] \in R_j) \wedge (\forall_{1\leqslant i\leqslant N} i)(z[V_i] \in D_i) \wedge z[W] = w\} = \{z \in Z | z[W] = w\} = \sigma_{W=w}(Z)$$

where σ denotes relational selection [74].

After instantiation of W to w, a reduction search algorithm applies domain reduction, as in Section 2.1, but working with the definition of $\sigma_{W=w}(Z)$ instead of the definition of Z. It is worthwhile to apply domain reduction at this stage because more domains are now single-valued, and this fact may enable deletion of values that could not previously be deleted. If domain reduction empties any domain then $\sigma_{W=w}(Z)$ is empty, which means that w is not part of any solution N-tuple z in Z. In this case there is no point in exploring instantiation of any further variable while $W = w$; the search tree is pruned by omitting all such instantiations.

If an application of domain reduction does not empty any domain then the reduction search algorithm will instantiate a further variable V_i to a value v that is currently in domain D_i. Domain reduction operations may have reduced the cardinality of D_i, thus pruning the search tree by reducing the number of different values of V_i that the search algorithm should try whilst $W = w$.

Before instantiating V_i, a search algorithm retains copies of the domains of all uninstantiated variables, typically by pushing these onto a stack. Contents of domains are restored, typically by popping copies of domains off the stack, when V_i is re-instantiated to a different value, and also when the search backtracks to re-instantiate a variable in W. Similarly, if tuples are removed from constraint relations, these must be restored when V_i is re-instantiated and when the search backtracks.

It is well known that the speed of reduction search may depend strongly on the sequence in which variables are instantiated. A*static variable ordering* is an instantiation sequence that does not change during the search. The only heuristic used for this in the present paper is that variables are processed in sequence of decreasing degree, as in [41,76]. The *degree* of a variable is the number of scopes to which it belongs.

For *dynamic variable ordering*, which changes the instantiation sequence during the search, we use the *minimum remaining values* heuristic [7,36]. This selects for instantiation a variable V_k such that the cardinality of D_k is minimal. If more than one uninstantiated domain has the smallest cardinality, we choose the first of these in the sequence determined by static variable ordering. This implements the *dom* + *deg* heuristic of Frost and Dechter [30]. Another popular heuristic is *dom/deg* which selects V_k such that $|D_k|$ divided by the degree of V_k is minimal [13].

Fig. 1 outlines a reduction search algorithm in which procedure *choose* returns i such that V_i is the next variable to be instantiated. Procedure *choose* returns *allDomainsAreSingleValued* = **true** if and only if all domains are single valued. Domain reduction, and possibly also constraint relation reduction, is done by procedure *reduce* which returns *consistent* = **false** if any domain is empty. If procedure *reduce* removes tuples from constraint relations, these are eventually restored by calling procedure *restore* instead of by popping off a stack.

2.3. Restricted reduction search

During each invocation of procedure *reduce*, propagation of deletion from domains may be allowed to continue until convergence, which is the situation where no further value can be deleted without changing $\sigma_{W=w}(Z)$. However, the cost of evaluating conditions for deletion has to be taken into account. Available theory does not tell us which deletions will prove to be cost effective. Nor does it say that reduction search will terminate tolerably quickly; prohibitively exponential time will be required in the worst case because too many domains have too many values at the time of instantiation.

```
1  Z:= empty; staticResequence; preprocess; initialiseStacks;
2  choose(i, allDomainsAreSingleValued);
3  repeat
4    if any untried value remains in domain D_i then
5      v:= an untried value selected from D_i
6      push(i); push(D); D_i := {v};
7      reduce(i, D, consistent);
8      if consistent then
9        choose(i, allDomainsAreSingleValued);
10       if allDomainsAreSingleValued then
11         include z in Z;
12         restore(i); pop(i); pop(D);
13       else mark all values in domain D_i to indicate that they are untried
14       end if
15     else restore(i); pop(i); pop(D);
16     end if
17   else
18     if not stackEmpty() then pop(i); pop(D); restore(i) else i := 0 end if;
19   end if
20 until i = 0;
```

Fig. 1. Reduction Search Algorithm that enumerates Z.

In our terminology, *C-reduction search* is domain reduction search such that procedure *reduce* removes values from domains until convergence; C-reduction is an abbreviation for convergence reduction. Ullmann [77] found experimentally that in some cases Z could be enumerated faster by *S-reduction search* in which domain reduction is confined to a single iteration (as in preclusion in [36]) instead of continuing until convergence. S-reduction is an abbreviation for single-iteration reduction. When S-reduction is used, Fig. 1 search algorithm is modified so that it enters an N-tuple z into Z only if z satisfies an explicit check for satisfaction of all constraints.

Immediately after instantiation of V_i, procedure *reduce* in Fig. 1 can be restricted to process only a subset of the scopes that belong to Σ_i. For binary constraints, McGregor made S-reduction search faster by processing only those scopes in Σ_i that included an uninstantiated variable [57, Fig. 5]. This technique was investigated by Haralick and Elliott [41] who gave it the name *forward checking*.

Following Bessière et al. [15], forward checking can be generalized by using a parameter κ to limit the scopes that are processed by procedure *reduce*. Specifically, a constraint in Σ_i is processed only if its scope includes at least one and not more than κ uninstantiated variables. If κ is reduced then fewer constraints are processed so the search tree is not pruned so heavily and more nodes of the search tree are visited, but procedure *reduce* works faster. In our terminology, Sκ-reduction search uses S-reduction with a specified value of κ. Forward checking in [81] is an example of S$n-1$-reduction search with $\kappa = n - 1$, where n is the arity of all constraints. Cn-reduction search is C-reduction without restriction on the scopes processed by procedure *reduce*. Following [66], Cn-reduction search is sometimes known as MAC, or, for non-binary constraints, GMAC.

2.4. Tabular implementation of constraint relations

2.4.1. Simple tabular reduction search

When constraint relations are available as explicit tables of n_j-tuples, the intersection $R_j \cap (D_{j_1} \times D_{j_2} \times \cdots \times D_{j_{n_j}})$ can be implemented by using the *tabular* strategy

for each tuple $t = \langle t_1, \ldots, t_k, \ldots, t_{n_j} \rangle$ in R_j **do**
  for $k := 1$ **to** n_j **do** seek t_k in D_{j_k} **end for**
end for

which is used in a version of procedure *reduce* that is shown in Fig. 2. Following [52], this version works with a queue of scopes. The queue is implemented as a set so that duplicates are removed automatically. Domains are also implemented as sets to enable rapid implementation of *seek* t_k *in* D_{j_k}. To relate the global numbering to the local (within a scope) numbering of variables we use a function G such that $G_j(k) = i$, where V_{j_k} is the same variable as V_i.

If a tuple t in R_j is not in $(D_{j_1} \times \cdots \times D_{j_{n_j}})$ then this tuple cannot be found again in $(D_{j_1} \times \cdots \times D_{j_{n_j}})$ until reduction search has restored values to domains. Until this time, any such tuple t can be removed temporarily from R_j so procedure *reduce* in Fig. 2 will process fewer tuples [78]. For this purpose, the table R_j is stored as a linked list of tuples. We provide a two-dimensional array, called *eliminated*, of linked lists of tuples such that *eliminated*$[i, j]$ will contain tuples removed from *list* R_j during the execution of procedure *reduce* immediately after instantiation of V_i. Initially every list in *eliminated*$[i, j]$ is empty. In this paper, square brackets are overloaded to signify either restriction or array subscripts, depending on the context.

Tuples that have been temporarily eliminated from constraint relations must be restored when a variable is re-instantiated and when the search backtracks. This restoration is done by the call of *restore(i)* in Fig. 1. The effect is that for each scope S_j the list *eliminated*$[i, j]$ is appended to the list R_j and then *eliminated*$[i, j]$ is re-initialized to be empty. The append operation is implemented without traversing either of the operand lists and without moving any tuple in memory. It is therefore quick. This simple method of temporarily eliminating and subsequently restoring tuples is called *incremental restoration* because it is redolent of *incremental dumping* in database technology.

```
procedure reduce(in i: integer; in out D: domainType; out consistent: boolean);
begin
1  queue:= Σ_i;
2  repeat
3    remove j from queue;
4    for k := 1 to n_j do C_k := empty end for;
5    for each tuple t = ⟨t_1, ..., t_k, ..., t_{n_j}⟩ ∈ R_j do
6      if (∀k, 1≤k≤n_j)(t_k ∈ D_{G_j(k)}) then
7        for k := 1 to n_j do include t_k in C_k end for
8      else
9        remove t from list R_j; insert t into list eliminated[i, j]
10     end if
11   end for
12   for k := 1 to n_j do
13     if C_k is empty then consistent:= false; return end if;
14     if C_k ≠ D_{G_j(k)} then
15       D_{G_j(k)} := C_k;
16       queue:= queue ∪Σ_{G_j(k)};   remove j from queue
17     end if
18   end for
19 until queue is empty;
20 consistent:= true;
end reduce;
```

Fig. 2. C*n*-reduction implementation of procedure *reduce*.

```
procedure reduce(in i: integer; in out D: domainType; out consistent: boolean);
begin
1  for each j in Σ_i do
2    u := number of uninstantiated variables in the scope S_j;
3    if 1 < u ≤ κ then
4      for k := 1 to n_j do C_k := empty end for;
5      for each tuple t = ⟨t_1, ..., t_k, ..., t_{n_j}⟩ ∈ R_j do
6        if (∀k, 1≤k≤n_j)(t_k ∈ D_{G_j(k)}) then
7          for k := 1 to n_j do include t_k in C_k end for
8        else
9          remove t from list R_j; insert t into list eliminated[i, j]
10       end if
11     end for
12     for k := 1 to n_j do
13       if C_k is empty then consistent:= FALSE; return end if;
14       D_{G_j(k)} := C_k;
15     end for
16   end if
17 end for
18 consistent:= true;
end reduce;
```

Fig. 3. S*κ*-reduction implementation of procedure *reduce*, which does not require a queue.

We use the term *simple tabular reduction search* to mean reduction search as in Fig. 1 with procedure *reduce* implemented as in Fig. 2 or, for Sκ-reduction, as in Fig. 3. Simple tabular reduction search is characterized by the use of sets $C_1, \ldots, C_{n_j}$, as in [77,78], and also by the use of incremental restoration.

With simple tabular reduction the main overall memory requirement is for storing constraint *n*-tuples and is therefore O(nqK), where q is the total number of scopes, K is the maximum cardinality of a constraint relation, and, for simplicity, all scopes are assumed to have arity n. The memory requirement for the array *eliminated* of pointers is O(Nq) which is relatively unimportant when $nK \gg N$.

```
1 for each variable V_h in S_j do
2    for each value v in D_{G_j(h)} do
3       seek t = ⟨t_1, ..., t_k, ..., t_{n_j}⟩ in R_j
4       such that (v = t_h) ∧ (∀k_{1≤k≤n_j})(t_k ∈ D_{G_j(k)})
5    end for
6 end for
```

Fig. 4. Outline of reduction strategy in GAC2001.

2.4.2. Related algorithms

Algorithm GAC2001 [14], which is formulated explicitly as Fig. 2 in [54], is particularly important because it is optimal when constraints are binary [14,87]. The strategy used in GAC2001 is outlined in Fig. 4. A tuple that satisfies the condition in Line 4 is said to *support* the value v in $D_{G_j(h)}$. If no support is found in R_j for value v, then this value is deleted from $D_{G_j(h)}$. An array *currentSupport* records, for each j, h, v, which tuple (if any) in R_j supports value v in $D_{G_j(h)}$. Using this array, the search at Line 3 continues from where it stopped last time Line 3 was executed for given values of j, h, v, if there has meanwhile been no backtrack nor re-instantiation.

Within a single iteration of the outer **repeat** loop in Fig. 2, each tuple in R_j is checked just once at Line 6. In simple tabular reduction there is not a separate search through tuples in R_j for each h and v, as at Line 3 in Fig. 4. GAC2001 may check whether $(\forall k_{1\leqslant k\leqslant n_j})(t_k \in D_{G_j(k)})$ for the same tuple t many times during a single execution of the nested loop that starts at Line 1 in Fig. 4. GAC2001 does not remove from R_j a tuple that does not satisfy this condition.

As in [75, p. 593] and [78, p. 153], simple tabular reduction removes from R_j any tuple that does not satisfy $(\forall k_{1\leqslant k\leqslant n_j})(t_k \in D_{G_j(k)})$. Subsequently, every tuple visited in the loop starting at Line 5 in Fig. 2 supports one value in each domain. This loop may do unnecessary work in that it may continue to process tuples of R_j after achieving $(\forall k_{1\leqslant k\leqslant n_j})(C_k = D_k)$. This could be prevented by checking for satisfaction of this condition; but the cost of this check would have to be taken into account. GAC2001 does not necessarily process every tuple in R_j, unless there is a value in some D_k that currently has no support in R_j.

Hidden variable Algorithm HAC [67, Fig. 3] uses the strategy outlined in Fig. 4 and, like simple tabular reduction, removes from R_j any tuple that does not satisfy $(\forall k_{1\leqslant k\leqslant n_j})(t_k \in D_{G_j(k)})$. This removal is accomplished at the earliest opportunity so that, at Line 4 in Fig. 4, there is no need to check, for each k, that $t_k \in D_{G_j(k)}$. Tuple t would not currently be in R_j if this condition were not satisfied.

Simple tabular reduction removes tuple t from R_j by unlinking it from a linked list (and then linking it into a list of eliminated tuples so that it can be restored to R_j in due course). Instead of physically removing t from R_j, HAC resets a Boolean variable associated with t to signify that t is currently not in R_j. At Line 3 in Fig. 4, HAC may spend time inspecting Boolean variables associated with tuples that are not currently in R_j. Reasons for working this way are:

- HAC uses an array *currentSupport* in the manner of GAC2001, which requires the tuples of R_j to be maintained in a fixed sorted order. This sorted order can be preserved if Boolean variables are used as in HAC. The link/unlink method of simple tabular reduction destroys this sorted order.
- The Boolean variable associated with a tuple is an element of a Boolean array. The subscript of this element can be regarded as a value of a hidden variable, as explained in [8]. This is of interest from the viewpoint of conversion of non-binary to binary constraint satisfaction problems [9,54,71].

2.5. Distributive implementation of constraint relations

The tabular strategy is not always the best. When many domains in scope S_j are single-valued, as they are when $\kappa < 3$, the distributive strategy [77]

for each tuple t in $D_{j_1} \times \cdots \times D_{j_{n_j}}$ **do** seek t in R_j **end for**

```
for each tuple t = ⟨t_1, ..., t_k, ..., t_{n_j}⟩ in (D_{G_j(1)} × ··· × D_{G_j(n_j)}) do
  if R_j[t_1, ..., t_{n_j}] = 1 then
    for k:= 1 to n_j do include t_k in C_k end for
  end if
end for
```

Fig. 5. Fragment using distributive representation of constraint relations.

may be faster than processing successive tuples in a tabular representation of R_j. The distributive strategy requires a distributive representation of constraint relation R_j that allows rapid discovery whether or not a given tuple $t = \langle t_{j_1}, \ldots, t_{j_{n_j}} \rangle$ is in R_j. A simple example of a distributive representation is an n_j-dimensional array of bits, initialized so that $R_j[t_{j_1}, \ldots, t_{j_{n_j}}] = 1$ iff $t \in R_j$. The distributive strategy can also be used when R_j is intensively defined by a predicate $P_j(t)$ that is true iff $t \in R_j$; in this case *seek t in* R_j is replaced by *evaluate* $P_j(t)$.

When R_j is represented by an n_j-dimensional array of bits, Lines 5–11 in Figs. 2 and 3 are replaced by Fig. 5. Distributive implementation [77] does not eliminate tuples from constraint relations, because speed would not be gained by this. Calls of procedure *restore* in Fig. 2 are therefore not required.

3. Constraint relation reduction

3.1. Semijoin reduction

With tabular implementation of domain reduction, the search may be speeded up by removal of tuples from constraint relations. Simple tabular reduction will certainly not remove from a constraint relation R_h any tuple $t = \langle t_1, \ldots, t_k, \ldots, t_n \rangle$ such that $t_k \in D_{G_h(k)}$ for all $1 \leqslant k \leqslant n_j$. We now introduce just one method that may remove such a tuple from R_h. This method is based on the fact (Theorem 2 in the Appendix) that the set Z is not changed as a result of the deletion, from any constraint relation R_h, of any tuple t such that $t \notin \pi_{S_h}(R_h \bowtie R_j)$ for some R_j such that $S_h \cap S_j \neq \emptyset$. In other words, if $S_h \cap S_j \neq \emptyset$, we can delete from R_h any tuple t that does not concatenate with at least one tuple in R_j in the natural join $R_h \bowtie R_j$. This deletion can be accomplished by the routine in Fig. 6, which implements *semijoin reduction* [12,53] of constraint relations. The semijoin operation $\ltimes$ is defined [12,53] by

$$R_h \ltimes R_j = \pi_{S_h}(R_h \bowtie R_j)$$

so $R_h \ltimes R_j$ is the subset of R_h that concatenates with at least one tuple of R_j in the natural join $R_h \bowtie R_j$.

Fig. 6 routine may process some (h, j) pairs unnecessarily. To see why, let us denote $S_h \cap S_j$ by S^{hj} and suppose that S^{hj} is non-empty and is a subset of $S_h \cap S_k$ and also of $S_k \cap S_j$. After the semijoin reduction operation

$$R_k := \{t \in R_k | t[S_j \cap S_k] \in \pi_{S_j \cap S_k}(R_j)\}$$

a tuple t survives in R_k only if $t[S^{hj}] \in \pi_{S^{hj}}(R_j)$, because $S^{hj} \subseteq S_j \cap S_k$. Similarly, after the subsequent operation

$$R_h := \{u \in R_h | u[S_h \cap S_k] \in \pi_{S_h \cap S_k}(R_k)\}$$

a tuple u survives in R_h only if $u[S^{hj}] \in \pi_{S^{hj}}(R_k)$. The previous operation has ensured that $\pi_{S^{hj}}(R_k) \subseteq \pi_{S^{hj}}(R_j)$, so a tuple u survives in R_h only if $u[S^{hj}] \in \pi_{S^{hj}}(R_j)$. Therefore it is unnecessary to delete tuples from R_h by using

```
repeat
  for each S_h ∈ Σ do
    for each S_j ∈ Σ such that S_h ∩ S_j is not empty do R_h := R_h ⋉ R_j end for
  end for
until no further tuples are deleted from any constraint relation
```

Fig. 6. A simple semijoin reduction routine for constraint relation reduction.

$R_h := \{u \in R_h | u[S^{hj}] \in \pi_{S^{hj}}(R_j)\}$ directly because these tuples will be deleted from R_h if we first reduce R_k and then R_h. In this case we say that the pair (S_h, S_j) is *redundant* [25,44].

3.2. Acceleration of semijoin reduction by Bloom filtering

Fig. 7 shows Zhang and Mackworth's semijoin reduction *procedure ZM* which uses a queue [52] and does not process redundant pairs [86]. Line 9 of procedure ZM could be implemented by searching linearly through the tuples in $\pi_{S_h \cap S_j}(R_j)$ seeking one that matches $t[S_h \cap S_j]$. This linear search would be repeated for each tuple t in R_h. To avoid this nested loop we could use a multidimensional array, b, of bits. Let $b[u[S_h \cap S_j]]$ be the bit whose subscripts are successive values in the tuple $u[S_h \cap S_j]$. Using this notation, in which outer square brackets denote array subscripts and inner square brackets denote restriction, Lines 8–12 of procedure ZM can be replaced by the fragment shown in Fig. 8, which scans through R_h and R_j just once. Assuming for simplicity that all scopes have arity n, there are $n - 1$ values in the largest overlap between any two scopes and therefore array b requires a maximum of δ^{n-1} bits, where δ is the maximum cardinality of any domain.

Babb [5] avoids excessive memory requirements by replacing the single array b by an array, B, of m arrays, each comprising δ^{ζ} bits, where ζ is a control parameter. Let $\langle u[S_h \cap S_j]\rangle_1^{\zeta}$ be the tuple comprising the first ζ values in $u[S_h \cap S_j]$. Let $\langle u[S_h \cap S_j]\rangle_2^{\zeta}$ be the tuple comprising the next ζ values in $u[S_h \cap S_j]$, and so on. Finally, let $\langle u[S_h \cap S_j]\rangle_m^{\zeta}$ be the tuple comprising the last ζ (or fewer) values in $u[S_h \cap S_j]$. The routine that replaces Lines 8–12 of procedure ZM is now as shown in Fig. 9, in which the first subscript of array B selects one of the m arrays of bits. Because two scopes may overlap in at most $n - 1$ variables the largest possible value of m is $(n - 1)/\zeta$ rounded up to the nearest integer.

The contents of array B constitute a Bloom filter. In accordance with Bloom filter theory [59,62] there is a risk that a tuple which should be deleted from R_h will actually not be deleted, but this risk can be made small by appropriate choice of ζ. All our experiments with procedure ZM have used Babb's Bloom filter, which greatly improves the cost-effectiveness of semijoin reduction by avoiding the nested loop in Fig. 7.

```
1  queue:= {1,...,q};
2  while queue is not empty do
3    remove a value h from queue;
4    changed:= false;
5    for each scope S_j such that
6          S_h ∩ S_j is not empty and
7          the pair (S_h, S_j) is not redundant do
8      for each tuple t in R_h do
9        if t[S_h ∩ S_j] ∉ π_{S_h∩S_j}(R_j) then
10         delete t from R_h; changed:= true
11       end if
12     end for;
13     if changed then
14       include in queue all k such that
15       S_k ∩ S_h is not empty and the pair (S_k, S_h) is not redundant
16     end if
17   end for
18 end while;
```

Fig. 7. Semijoin reduction procedure ZM.

```
set all bits of b to zero;
for each tuple u in R_j do b[u[S_h ∩ S_j]] := 1 end for;
for each tuple t in R_h do
  if b[t[S_h ∩ S_j]] = 0 then delete t from R_h; changed:= true end if
end for;
```

Fig. 8. Using an array of bits to avoid a nested loop.

set all bits of $B[1,\ldots], B[2,\ldots], \cdots, B[g,\ldots], \cdots, B[m,\ldots]$ to zero;
for each tuple u in R_j **do**
  for $g := 1$ **to** m **do** $B[g, \langle u[S_h \cap S_j]\rangle_g^{\zeta}] := 1$ **end for**;
end for
for each tuple t in R_h **do**
  if$(\exists_{1\le g\le m} g)(B[g, \langle t[S_h \cap S_j]\rangle_g^{\zeta}] = 0)$ **then** delete t from R_h; changed:= **true**
  end if
end for;

Fig. 9. Implementation of Lines 8–12 of procedure ZM using Babb's Bloom filter.

3.3. Related work

Algorithm PW-AC [67] avoids the nested loop in Fig. 7 by adapting the idea of piecewise functionality [82]. This idea is applicable because if x is a tuple on $S_h \cap S_j$ then a tuple in $\sigma_{(S_h\cap S_j)=x}(R_h)$ concatenates in $R_h \bowtie R_j$ only with tuples in $\sigma_{(S_h\cap S_j)=x}(R_j)$. For each overlapping pair S_h, S_j and for each possible tuple x on $S_h \cap S_j$, Algorithm PW-AC maintains a counter that always contains the cardinality $|\sigma_{(S_h\cap S_j)=x}(R_h)|$. If this cardinality is zero then every tuple in $\sigma_{(S_h\cap S_j)=x}(R_j)$ can be deleted from R_j. If, as a result of this deletion, for some tuple y the cardinality $|\sigma_{(S_j\cap S_k)=y}(R_j)|$ becomes zero then every tuple in $\sigma_{(S_j\cap S_k)=y}(R_k)$ can be deleted from R_k. Thus deletion is propagated as described in [67].

Section 5.4.1 reports experiments with PW-AC in which all tuples in $\sigma_{(S_j\cap S_k)=y}(R_k)$ have been found simply by linear search through all tuples in R_k, because this allows us to work without limits on $|S_j \cap S_k|$ and on δ. In a nested loop implementation of a semijoin, the search through all tuples in R_k would be repeated for each tuple in R_j.

For the pair S_j, S_k, the amount of memory required for Bloom filter array B is not fixed. Ideally, B should be made so small that further reduction in its size would appreciably reduce the speed of the constraint relation reduction process, because of increased probability of failure to delete some tuples [61]. If the number of bits in array B were greater than $\delta^{|S_j\cap S_k|}$ we could instead use array b as in Fig. 8 which requires this number of bits. This number, which can be regarded as the maximum size of array B, is of bits packed into words. For the pair S_j, S_k, algorithm PW-AC requires this same number $\delta^{|S_j\cap S_k|}$ of counters. Moreover, in Fig. 9 the same array B is re-used for every semijoin. In PW-AC, separate counters are required for all non-redundant overlapping pairs of scopes. When algorithm PW-AC is interleaved with backtrack search, if the counters are not re-initialised at each invocation of PW-AC then a save/restore mechanism must be provided for the counters. Because Bloom filter array B is re-initialised for every semijoin, it does not require save/restore during backtrack. We can safely conclude that the memory requirement for PW-AC is very much greater than that for the Bloom filter implementation of semijoin reduction procedure ZM.

4. Partition search

4.1. Partition resequencing

To evaluate the set Z of solutions, the routine

$Y := R_1$; **for** $j := 2$ **to** q **do** $Y := Y \bowtie R_j$ **end for**; $Z := Y$;

works breadth-first; the result of a join is obtained completely before another join is commenced. For constraint satisfaction it is more appropriate instead to proceed depth-first, as in [68]. One reason is that the computation may be required to stop as soon as any solution N-tuple is found. Another reason is that in a constraint satisfaction problem, many variables may belong to more than one scope. This means that many tuples in an intermediate composite relation Y may not be part of any N-tuple in Z. By working depth-first we avoid provision of memory for tuples that are not part of any N-tuple in Z.

Let Σ_{old} be the set of scopes that is given initially as part of the problem formulation; select and remove from Σ_{old} any S_i;
$S_{union} := S_i$; $k := 1$;
renumber S_i to be $S_1^{renumbered}$;
repeat
  select and remove from Σ_{old} an S_j such that S_j maximally
    overlaps S_{union} so that $|S_{union} \cup S_j|$ is minimal;
  $S_{union} := S_{union} \cup S_j$; $k := k + 1$;
  renumber the selected S_j to be $S_k^{renumbered}$;
until S_{union} contains every variable in $V = \{V_1, \ldots, V_N\}$;
renumber scopes that remain in Σ_{old} in any sequence

Fig. 10. Simplified outline of a maximal overlap resequencing routine.

The present paper assumes that all of the constraint relations $R_1, \ldots, R_q$ are held in main memory. In database systems where $R_1, \ldots, R_q$ are all in secondary memory instead of main memory, it is worthwhile to spend more time determining a (hopefully) optimal execution sequence for multiple joins. Dynamic programming is widely employed for this purpose, unless the number of joins is so large that a simple greedy algorithm is more practical [33,46]. Greedy and combinatorial algorithms commonly use estimates of numbers of tuples in results of proposed joins, assuming statistical independence of values of variables [6,55].

In constraint satisfaction, where many variables may belong to more than one scope, the independence assumption is particularly unsatisfactory [22]. Avoidance or mitigation of the independence assumption has a computational cost [43]; obviously the cost of join sequence optimization must not be so high that the sum of optimization time plus search time is unduly large when working with all constraint relations in main memory. In this paper we use a simple greedy algorithm that does not attempt to estimate cardinalities of result relations.

In the left-deep routine shown at the beginning of this section, the number of tuples in an intermediate composite relation Y can be expected to grow rapidly as j increases. To mitigate this growth, we join constraint relations in a sequence $R_1^{\text{renumbered}}, R_2^{\text{renumbered}}, \ldots$ chosen using a heuristic that is intended to make the scope of Y be as small as possible at each step. As in [64, p. 151], the idea is that by applying as many constraints to as few variables as possible, the number of tuples in each successive Y will be restricted. Constraint relations are renumbered to save time during the subsequent depth-first search.

Fig. 10 shows an introductory outline of a greedy routine that chooses a join execution sequence. This routine is similar to the maximum cardinality algorithm [73] which breaks ties arbitrarily. For partition search it is worthwhile to use heuristic scores in choosing between scopes that have the same $|S_{\text{union}} \cup S_j|$. For each scope $S_h = \{V_{h_1}, \ldots, V_{h_k}, \ldots, V_{h_n}\}$ we compute a score $\alpha_h = (\sum_{k=1}^{n} \rho_{h_k})/|R_h|$, where ρ_i is the number of scopes to which variable V_i belongs. If $|S_{\text{union}} \cup S_j| = |S_{\text{union}} \cup S_h|$ then if $\alpha_j > \alpha_h$ the resequencing algorithm chooses S_j in preference to S_h. Partition search uses the *partition resequencing* algorithm that is outlined in Fig. 11. This algorithm renumbers variables as well as scopes; search efficiency is improved by processing the renumbered variables in the straightforward sequence $V_1, V_2, \ldots, V_N$, thus avoiding indirection that would otherwise be required. We work henceforward with renumbered variables and scopes without the *superscript*$^{\text{renumbered}}$.

Let Σ_d be the set of scopes that have been selected at the time of termination of the **repeat** loop in Fig. 11, just before commencement of the final **while** loop. Let q_d be the number of scopes in Σ_d. Each scope S_j in Σ_d can be regarded as the union of two disjoint (i.e. non-overlapping) subsets S_j^a and S_j^b. S_j^b is the subset of S_j that was not in S_{union} immediately before S_j was selected by the partition resequencing algorithm. S_j^a is the subset of S_j that was, at this time, included in S_{union}. It is easy to see that, as a result of partition resequencing, Σ_d has all of the following properties:

Partition properties

1. For all $S_j \in \Sigma_d$ the variables in S_j^b are contiguous. This means that if $g < k$, $V_g \in S_j^b$ and $V_k \in S_j^b$ then $g < h < k$ implies $V_h \in S_j^b$.

2. The subsets S_j^b are disjoint. Thus if S_h and S_k are any two scopes in Σ_d such that $h \neq k$, then $S_h^b \cap S_k^b$ is empty.
3. $S_1^b \cup S_2^b \cup \ldots \cup S_{q_d}^b = V = \{V_1, \ldots, V_N\}$. Thus the set of subsets $\{S_1^b, S_2^b, \ldots, S_{q_d}^b\}$ is a partition on the set V of variables. (Hence the name *partition resequencing*).
4. In this partition on V the successive subsets are in the sequence $S_1^b, S_2^b, \ldots, S_{q_d}^b$. In other words, if $V_h \in S_j^b$ and $V_i \in S_k^b$ then $j < k$ implies $h < i$.
5. S_1^a is empty and $S_1^b = S_1$.
6. For all j such that $2 \leqslant j \leqslant q_d$, $S_j^a \subset S_1 \cup S_2 \cup \ldots \cup S_{j-1}$.

Table 1 shows an example of partition resequencing with $N = 10$, $n = 5$ and $q = 6$, where q is the number of scopes. A scope is represented by a row of 10 characters; a hyphen signifies that the variable does not belong to the scope. In this example $\Sigma_d = \{S_1, S_2, S_3, S_4\}$ and the partition is $\{S_1^b = \{V_0, V_1, V_2, V_3, V_4\},\ S_2^b = \{V_5, V_6\},\ S_3^b = \{V_7\},\ S_4^b = \{V_8, V_9\}\}$.

In Section 2.4 we introduced a function $G_j(k)$ such that a variable V_{j_k} within scope S_j is the same thing as the variable V_i such that $i = G_j(k)$. Before resequencing, the local numbering $V_{j_1}, \ldots, V_{j_k}, \ldots, V_{j_m}, \ldots, V_{j_n}$ of variables within a scope S_j is such that $k < m$ implies $G_j(k) < G_j(m)$. This sequence is usually destroyed by resequencing, which changes the global numbering of variables. For partition search it is essential that this sequence be restored. We achieve this by permuting the sequence of values in tuples in constraint relations. For all constraints this permutation, which is one of the overheads of partition search, ensures that S_j is again an ordered set $\{V_{j_1}, \ldots, V_{j_k}, \ldots, V_{j_m}, \ldots, V_{j_n}\}$ such that $k < m$ implies $G_j(k) < G_j(m)$.

4.2. Basic algorithm for partition search

Partition search is a depth first search algorithm that instantiates $S_1, S_2^b, \ldots, S_j^b, \ldots, S_{q_d}^b$, always in that straightforward sequence. Instantiation of S_j^b means instantiation of all variables therein; more than one variable may be instantiated at the same time. The instantiation sequence for variables is such that when V_i has been instantiated there is no $h < i$ such that V_h has not been instantiated. When all of the variables in

```
Let Σ_old be the set Σ of scopes that is given initially as part of the problem
formulation; select and remove from Σ_old a scope S_j for which α_j is maximal;
renumber the selected scope to be S_1^renumbered; renumber the variables
in this scope to be V_1^renumbered, V_2^renumbered ..., V_n^renumbered;
S_union := the selected scope;
nextScopeNumber:= 2; nextVariableNumber:= n + 1;
repeat
   Let T_1 = {S_h ∈ Σ_old | S_h is not a subset of S_union };
   Let T_2 = {S_h ∈ T_1 | the cardinality of S_union ∩ S_h is maximal within T_1};
   Choose from within T_2 a scope S_j such that α_j is maximal;
   remove S_j from Σ_old;
   for each variable V_i in S_j that is not in S_union do
      renumber V_i to be V_nextVariableNumber^renumbered;
      nextVariableNumber:= nextVariableNumber + 1;
   end for;
   S_union := S_union ∪ S_j;
   renumber S_j to be S_nextScopeNumber^renumbered;
   nextScopeNumber:= nextScopeNumber + 1;
until nextVariableNumber = N + 1;
while Σ_old is not empty do
   remove a scope from Σ_old and renumber this scope to be S_nextScopeNumber^renumbered;
   nextScopeNumber:= nextScopeNumber + 1;
end while;
```

Fig. 11. A partition resequencing routine.

Table 1
Six scopes (a) before and (b) after partition resequencing

	9	8	7	6	5	4	3	2	1	0
(a)										
1	–	A	A	A	–	–	–	A	–	A
2	–	–	B	B	–	B	B	B	–	–
3	–	–	–	–	C	C	C	C	C	–
4	–	D	D	–	D	–	–	–	D	D
5	E	–	–	E	E	–	–	–	E	E
6	F	–	F	F	F	–	–	F	–	–
(b)										
1	–	–	–	–	–	F	F	F	F	F
2	–	–	–	A	A	–	A	A	–	A
3	–	–	D	D	D	–	D	–	D	–
4	B	B	–	–	–	–	B	B	–	B
5	C	C	C	–	–	–	–	–	C	C
6	–	–	E	–	E	E	–	E	E	–
	4	3	1	8	0	9	7	6	5	2

The 10 variables are numbered 0, 1, ..., 9 from right to left as indicated in the top row. In (b) the bottom row shows how the variables were numbered prior to resequencing.

$W_i = \{V_1, \ldots, V_i\}$ have been instantiated such that V_1 has the value v_1, V_2 has the value $v_2, \ldots, V_i$ has the value v_i, we define $w_i = \{v_1, \ldots, v_i\}$. Additionally, we define $W_{e(j)} = W_i$ such that V_i is the last variable in the ordered set S_j. (In $e(j)$ the 'e' stands for 'end'.) Note that with partition resequencing we have by construction $S_j^a = S_j \cap W_{e(j-1)}$.

Partition search instantiates $S_1, S_2^b, \ldots, S_j^b$ so that, at every stage, $w_{e(j)}$ is a tuple in $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_j$. To achieve this, partition search instantiates S_1 to an tuple in R_1, thereby instantiating S_2^a because $S_2^a \subset S_1$. Partition search instantiates S_2^b to a tuple in $\pi_{S_2^b}(R_2 \bowtie w_{e(1)})$ so as to ensure that $w_{e(2)} \in R_2 \bowtie R_1$. Subsequently, partition search instantiates S_3^b to a tuple in $\pi_{S_3^b}(R_3 \bowtie w_{e(2)})$ so as to ensure that $w_{e(3)} \in (R_3 \bowtie w_{e(2)})$. Clearly, $(R_3 \bowtie w_{e(2)}) \subset (R_1 \bowtie R_2 \bowtie R_3)$.

Omitting practical details, Fig. 12 outlines a partition search algorithm that works with sets $P_1, \ldots, P_j, \ldots, P_{q_d}$ defined by

$$P_j = \begin{cases} R_1 & \text{if } j = 1 \\ \pi_{S_j^b}(R_j \bowtie w_{e(j-1)}) & \text{otherwise} \end{cases}$$

$P_j = \{P_{j1}, P_{j2}, \ldots, P_{j|P_j|}\}$ is the set of all tuples to which S_j^b could be instantiated so that $w_{e(j)} \in R_j \bowtie w_{e(j-1)}$ and therefore $w_{e(j)} \in R_1 \bowtie R_2 \bowtie \cdots \bowtie R_j$.

Scopes that are not in Σ_d are used for consistency checking. If there is a constraint R_x such that $S_x \subset W_{e(j)}$ and $w_{e(j)}[S_x] \notin R_x$, then $w_{e(j)}$ cannot be part of a solution N-tuple, and in this case procedure *consistent* returns **false**. For each $S_j \in \Sigma_d$, the (possibly empty) set, Θ_j, of scopes used for checking consistency is defined by

$$\Theta_j = \{S_x \in (\Sigma - \Sigma_d) | S_j^b \cap S_x \text{ is non-empty and } S_x \subset W_{e(j)}\}$$

Theorem 3, in the Appendix, asserts correctness of partition search, viewed as a sequential process. It is worth mentioning that partition search is not restricted to purely serial implementation. Database query processors may speed up single and multiple joins by using a number processors working in parallel [19,42,45]. To speed up depth-first search for constraint satisfaction, McCall et al. [56] subdivide the search into a number of disjoint concurrent searches. A single-processor implementation of partition search enumerates $\sigma_{(W_{e(j-1)}=w_{e(j-1)}) \wedge (S_j^b = P_{jh})}(Z)$ between the time of instantiating S_j^b to P_{jh} and the time of instantiating S_j^b to $P_{j(h+1)}$. Using the ideas of [56], a multiprocessor implementation of partition search may employ $|P_j|$ parallel processors each separately enumerating $\sigma_{(W_{e(j-1)}=w_{e(j-1)}) \wedge (S_j^b = P_{jh})}(Z)$ for a different value of h, $1 \leqslant h \leqslant |P_j|$. This concurrency is applicable whatever the detailed implementation of partition search.

```
procedure consistent(in j: integer): boolean;
begin
    for each scope S_x in Θ_j do
        if w_e(j)[S_x] ∉ R_x then return false end if
    end for;
    return true
end consistent;

partitionResequence; permuteTuples; Z:= empty;
j := 1; next[1]:= 1;
repeat
    h := next[j];
    if h ≤ |P_j| then
        next[j] := h + 1;
        instantiate S_j^b to P_jh;
        if consistent(j) then
            if j = q_d then include w_N in Z
            else j := j + 1; next[j]:= 1
            end if
        end if
    else j := j − 1
    end if
until j = 0;
```

Fig. 12. Partition search algorithm. The body of the algorithm follows after **end** consistent. Array element next[j] identifies the next tuple in $\{P_{j1}, P_{j2}, \ldots, P_{j|P_j|}\}$.

We now formulate conditions for best performance of partition search, again regardless of its detailed implementation. We define a *spurious* match to be a match $w_{e(j-1)}[S_j^a] = t[S_j^a]$ with a tuple $t \in R_j$ such that there is no z in Z for which $t \bowtie w_{e(j-1)} = z[W_{e(j)}]$. Thus a spurious match is one that leads to an instantiation of S_j^b such that $w_{e(j)}$ is not part of a solution. Spurious matches are less likely, and therefore partition search is faster, if

K is decreased so there are fewer tuples in constraint relations and therefore less chance that a tuple in a constraint relation will match spuriously.

$|S_j^a|$ *is increased* so that a spurious match is less likely because more variables are matched. Thus spurious matches are less likely with higher arity and with more overlap between scopes.

δ *is increased* so that a spurious match is less likely because domains are larger.

4.3. Hash join implementation of partition search

4.3.1. Hash buckets

In the definition of P_j, the join $R_j \bowtie w_{e(j-1)}$ concatenates $w_{e(j-1)}$ with each tuple $t \in R_j$ such that $t[S_j \cap W_{e(j-1)}] = w_{e(j-1)}[S_j \cap W_{e(j-1)}]$. We have noted previously that partition resequencing ensures $S_j^a = S_j \cap W_{e(j-1)}$. Thus partition search concatenates $w_{e(j-1)}$ with tuples of R_j that match $w_{e(j-1)}$ in S_j^a. This is achieved by instantiating S_j^b to $t[S_j^b]$, where t is a tuple in R_j such that $t[S_j^a] = w_{e(j-1)}[S_j^a]$. For use in this instantiation, partition search could examine successive tuples in R_j seeking a tuple t such that $t[S_j^a] = w_{e(j-1)}[S_j^a]$.

Instead it is very much more efficient to use a hash join [33,37,58,69] that examines only a few tuples of R_j instead of checking many of them. For all $j > 1$, a pre-search process partitions constraint relation R_j into hash buckets organized so that any two tuples which are in the same hash bucket have the same value of a hash function. This hash function must be such that every tuple $t \in R_j$ such that $t[S_j^a] = w_{e(j-1)}[S_j^a]$ has the same hash function value and therefore belongs to the same bucket. Every tuple t in R_j that concatenates with $w_{e(j-1)}$ in

$R_j \bowtie w_{e(j-1)}$ is in this bucket. To enumerate $R_j \bowtie w_{e(j-1)}$ it is only necessary to search this single bucket instead of searching through many tuples in R_j.

For all $j > 1$ we define S_j^μ to be the first m_j variables in the ordered set S_j^a, where

$$m_j = \begin{cases} \min(\mu_j, |S_j^a|) & \text{for } 1 < j \leqslant q_d \\ \mu_j & \text{for } q_d < j \leqslant q \end{cases}$$

and μ_j is a control parameter. Defining δ to be the maximum number of values in a domain, our hash function of a tuple $t = \langle t_1, \ldots, t_k, \ldots, t_{m_j}, \ldots, t_{n_j} \rangle$ is

$$hash(t[S_j^\mu]) = \sum_{k=1}^{k=m_j} t_k \delta^{k-1}$$

We have $S_j^a \subset w_{e(j-1)}$ and $S_j^\mu \subseteq S_j^a$, so at the time of instantiation of S_j^b, S_j^μ has already been instantiated to $w_{e(j-1)}[S_j^\mu]$. At this time, every tuple $t \in R_j$ such that $t[S_j^a] = w_{e(j-1)}[S_j^a]$ has $t[S_j^\mu] = w_{e(j-1)}[S_j^\mu]$ and is therefore located in the single hash bucket that is identified by $hash(w_{e(j-1)}[S_j^\mu])$.

For R_j the number of hash buckets is δ^{m_j}, which is the number of distinct hash function values. To prevent this number from being too large, we have $m_j = \min(\mu_j, |S_j^a|)$. We normally choose μ_j so that $\delta^{\mu_j} < |R_j|$.

The tuples of all constraint relations are stored in a single two-dimensional array r, where they remain without change during the search. For $j = 1$ the array elements $r[1,1], r[1,2], \ldots, r[1,|R_1|]$ are the tuples of R_1 in any sequence. For $j > 1$ the array elements $r[j,1], r[j,2], \ldots, r[j,|R_j|]$ are the tuples of R_j sorted into hash buckets so that all tuples which have hash value 0 are located in the first bucket, all tuples that have hash value 1 are located in the second bucket, and so on. If, for example, three tuples have hash value 0, six have hash value 1, five have hash value 2, none have hash value 3 and two have hash value 4, then the first bucket comprises $r[j,1], \ldots, r[j,3]$, the second bucket comprises $r[j,4], \ldots, r[j,9]$, the third bucket comprises $r[j,10], \ldots, r[j,14]$, the fourth bucket is empty, the fifth comprises $r[j,15], r[j,16]$, and so on.

After sorting the tuples of constraint relations into buckets, we construct an index that is a two dimensional array of records. This is initialized so that $index[j,i].first = h$ such that $r[j,h]$ is the first tuple in the sequence $r[j,1], r[j,2], \ldots, r[j,|R_j|]$ that has hash value i. Moreover, $index[j,i].last = k$ such that $r[j,k]$ is the last element in the sequence $r[j,1], r[j,2], \ldots, r[j,|R_j|]$ that has hash value i. We also arrange that $index[j,i].first = 0$ iff there is no tuple t in R_j that has hash value i. For the example in the previous paragraph, $index[j,2].first = 10$ and $index[j,2].last = 14$.

When $|R_j|$ is the same for all j we omit the subscript of μ_j. Similarly, when $|S_j|$ is the same for all j we omit the subscript of n_j. In partition search, the memory requirement for array r is $O(nqK)$, where K is the maximum $|R_j|$. The index contains $O(q\delta^\mu)$ records, so when $\delta^\mu < K$ the overall memory requirement for partition search is $O(nqK)$.

4.3.2. A hash join partition search algorithm

Fig. 13 shows a hash join implementation of partition search. While $j = 1$, successive iterations of the outer **repeat** loop assign to t successive tuples of R_1 in Line 5; in Line 13 S_1 is instantiated to the tuple t, since $S_j^b = S_j$ when $j = 1$.

At Line 14 a call of *consistent*(*j*) with empty Θ_j would return **true**. At Line 18, *bucketNumber* is a hash value calculated from the first m_j variables of $w_{e(j-1)}[S_j^a]$; this is used in the initialization of *next*[*j*] and *final*[*j*]. If *next*[*j*] = 0 then there is no tuple in R_j that matches $w_{e(j-1)}$ in the overlap $S_j \cap W_{e(j-1)}$ so P_j is empty and it will be impossible to instantiate S_j^b. In this case j is decremented in Line 21.

The **repeat** loop in Lines 7, 8 and 9 finds the next tuple $t \in R_j$ that matches $w_{e(j-1)}$ in the overlap $S_j \cap W_{e(j-1)}$. It is important that this **repeat** loop searches only the one bucket that contains tuples t such that $t[S_j^\mu] = w_{e(j-1)}[S_j^\mu]$. If a tuple t within this bucket is such that $t[S_j^a - S_j^\mu] = w_{e(j-1)}[S_j^a - S_j^\mu]$ then we have $t[S_j^\mu] = w_{e(j-1)}[S_j^\mu]$ and also $t[S_j^a - S_j^\mu] = w_{e(j-1)}[S_j^a - S_j^\mu]$ so $t[S_j^a] = w_{e(j-1)}[S_j^a]$. In this case $t \in R_j \ltimes w_{e(j-1)}$ and therefore $t[S_j^b] \in P_j$.

The function *consistent*(*j*) that is called in Fig. 13 is shown in Fig. 14. We have $S_x \in \Theta_j$ only if $S_x \in W_{e(j)}$, so S_x^μ has been instantiated when S_j^b has been instantiated. Therefore only a single bucket of R_x need be searched by the **loop** at Lines 8–13. If *index*[*x*, *bucketNumber*].*first* is zero then this bucket is empty, so there is no tuple t

```
1  partitionResequence; sortTuplesIntoHashbuckets; initialize; Z := empty;
2  j := 1; next[1]:= 1; final[1]:= |R_1|;
3  repeat
4      h := next[j];
5      if j = 1 then t := r[j, h]
6      else
7          repeat
8              t := r[j, h]; h := h + 1
9          until (t[S_j^a − S_j^μ] = w_e(j−1)[S_j^a − S_j^μ]) or (h > final[j])
10     end if;
11     if h ≤ final[j] then
12         next[j] := h + 1;
13         instantiate S_j^b to t[S_j^b];
14         if (Θ_j is empty) or consistent(j) then
15             if j = q_d then include w_N in Z
16             else
17                 j := j + 1;
18                 bucketNumber:= hash(w_e(j−1)[S_j^μ]);
19                 next[j]:= index[j, bucketNumber].first;
20                 final[j]:= index[j, bucketNumber].last;
21                 if next[j] = 0 then j := j − 1 end if;
22             end if
23         end if
24     else j := j − 1
25     end if
26 until j = 0;
```

Fig. 13. Hash join implementation of partition search.

```
1  procedure consistent(in j: integer): boolean;
2  begin
3      for each scope S_x in Θ_j do
4          bucketNumber:= hash(w_e(j)[S_x^μ]);
5          h := index[x, bucketNumber].first;
6          lastInBucket:= index[x, bucketNumber].last;
7          if h = 0 then return false end if;
8          loop
9              t := r[x, h];
10             matches:= (t[(S_x − S_x^μ) ∩ W_e(j)] = w_e(j)[(S_x − S_x^μ) ∩ W_e(j)])
11             if matches then exit end if;
12             if h = lastInBucket then exit else h := h + 1 end if
13         end loop;
14         if not matches then return false end if;
15     end for;
16     return true
17 end consistent;
```

Fig. 14. Function *consistent* called by the partition search algorithm in Fig 13.

such that $t[S_x^\mu] = w_{e(j)}[S_x^\mu]$. In this case $w_{e(j)}$ is not consistent with R_x because $w_{e(j)}[S_x] \notin R_x$ and the function therefore returns **false** at Line 7.

The tabular procedure *reduce* (at Line 5 in Fig. 2) scans all tuples in R_j, whereas for each $S_x \in \Theta_j$ the function *consistent*(j) in Fig. 14 scans only one bucket which is a small subset of R_x. This bucket contains tuples t

such that $t[S_x^\mu] = w_{e(j)}[S_x^\mu]$. If a tuple t within this bucket is such that $t[(S_x - S_x^\mu)] = w_{e(j)}[(S_x - S_x^\mu)]$ then we have $t = w_{e(j)}[S_x]$ and $w_{e(j)}[S_x] \in R_x$. Whilst scanning a bucket, the function *consistent*(j) only examines values in $(S_x - S_x^\mu)$, whereas procedure *reduce* (Fig. 2) examines all values in a tuple until a mismatch is found. A toy example of the working of partition search, using the scopes from Table 1b, is shown in Table 2.

4.4. Decision diagram implementation of partition search

4.4.1. Decision diagrams

Partition search is a general strategy that can be implemented in different ways. Instead of using hash joins, we can implement partition search by using decision diagrams, as follows.

Table 2
Partition search example with $\delta = 8$ and $\mu = 1$

(a)

r[1,1]	–	–	–	–	–	3	5	7	2	6
r[1,2]	–	–	–	–	–	0	3	7	4	6
r[1,3]	–	–	–	–	–	0	1	1	3	7
r[2,1]	–	–	–	6	4	–	5	7	–	6
r[2,2]	–	–	–	3	1	–	1	1	–	7
r[2,3]	–	–	–	7	6	–	1	1	–	7
r[3,1]	–	–	2	6	4	–	5	–	2	–
r[3,2]	–	–	4	7	6	–	1	–	3	–
r[3,3]	–	–	0	3	1	–	1	–	3	–
r[4,1]	7	7	–	–	–	–	6	6	–	2
r[4,2]	5	2	–	–	–	–	5	7	–	6
r[4,3]	4	2	–	–	–	–	1	1	–	7
r[5,1]	3	7	1	–	–	–	–	–	1	3
r[5,2]	5	2	2	–	–	–	–	–	2	6
r[5,3]	4	2	4	–	–	–	–	–	3	7
r[6,1]	–	–	2	–	4	3	–	7	2	–
r[6,2]	–	–	4	–	6	0	–	1	3	–
r[6,3]	–	–	5	–	2	2	–	1	5	–

(b)	2	3	4	5	6
0	0	0	0	0	0
1	0	0	0	0	0
2	0	1	1	0	1
3	0	2	0	1	2
4	0	0	0	0	0
5	0	0	0	0	3
6	1	0	2	2	0
7	2	0	3	3	0

(c)

j	$w_{e(j)}$										Comment
1	–	–	–	–	–	3	5	7	2	6	S_1 instantiated
2	–	–	–	6	4	3	5	7	2	6	Using the only matching tuple in R_2
3	–	–	2	6	4	3	5	7	2	6	Using the only matching tuple in R_3
4	5	2	2	6	4	3	5	7	2	6	Included in Z. Backtrack to $j = 1$
1	–	–	–	–	–	0	3	7	4	6	No matching tuple in R_2
1	–	–	–	–	–	0	1	1	3	7	S_1 re–instantiated
2	–	–	–	3	1	0	1	1	3	7	Using first matching tuple in R_2
3	–	–	0	3	1	0	1	1	3	7	No matching tuple in R_5, so backtrack
2	–	–	–	7	6	0	1	1	3	7	Using second matching tuple in R_2
3	–	–	4	7	6	0	1	1	3	7	Using the only matching tuple in R_3
4	4	2	4	7	6	0	1	1	3	7	Included in Z. Backtrack to $j = 0$

(a) Constraint relations. (b) Index: the entry in column j and row i is *index*[j,i].*first*. (c) Working of Fig. 13 algorithm.

An arc in a directed graph points from a parent node to a child node. A node is *terminal* if it has no children. A decision diagram (DD) is a directed acyclic graph in which each non-terminal node, $\mathcal{N}$, is associated with some variable V_i and every child of $\mathcal{N}$ is associated with a different value of V_i. In the present work, a DD has a single root node, which is a non-terminal node that has no parent. A DD is designed so that, if we start at the root node, proceed thence to the child associated with value a, proceed thence to the child associated with value b, and so on, until we reach the terminal node selected by value p of its parent, then information associated with this terminal node is relevant to the tuple $\langle a, b, \ldots, p\rangle$ [84].

If the sequence of variables along every path from a root node to a terminal node conforms to a single given sequence, then the DD is *ordered* [18,84]. In the implementation of partition search, we use ordered DDs to represent constraint relations. The ordering sequence is determined by partition resequencing. In the DD that represents R_j, the information associated with a terminal node is a single bit which is 1 if and only if the tuple that directed us from the root to this terminal node belongs to R_j.

A tuple $t = \langle t_1, t_2, \ldots, t_n\rangle$ can be converted to a tuple of bits by concatentating a bit-pattern representing t_1 with a bit-pattern representing $t_2 \cdots$ with a bit-pattern representing t_n, as described in [72, Section 3.1]. After this conversion, a constraint relation can be represented by a binary DD (BDD). A BDD is a DD in which no node has more than two children [18]. An advantage of using BDDs is that they have been well studied and have many applications [18,29,39]. Section 5.3.2 reports experiments with BDDs and also with multivalued DDs. In a multivalued DD (MDD) a non-terminal node may have more than two children [84, chapter 9].

4.4.2. Construction of tries from constraint relations

MDD and BDD implementation of partition search will be introduced in Section 4.4.4. This implementation is most easily understood in terms of *tries*, which are DDs that are restricted so that no non-terminal node has more than one parent [2,3]. Starting from a constraint relation R_j given as a table of tuples, the simple routine shown in Fig. 15 constructs a multivalued trie. A node is represented by a record that has two fields: *varNo* and *child*. Here *varNo* is the local variable number of the variable whose value will select one of the node's children. The field *child* is an array such that *child*[v] is a pointer to the child that will be selected by value v of the variable whose local number is *varNo*. Array element *root*[j] points to the root node of the trie that represents R_j.

Procedure *buildTrie* (Fig. 15) builds a trie in which there are exactly two terminal nodes: these are the '1' *sink* node and the '0' *sink* node. *zeroPtr* is a pointer to the '0' sink node; *onePtr* is a pointer to the '1' sink node.

```
procedure buildTrie(in j: integer);
begin
  new(root[j]);
  for v := 0 to δ − 1 do root[j] ↑ .child[v]:= zeroPtr end for;
  for t:= each tuple in R_j do
    parentPtr:= root[j];
    for k := 1 to n − 1 do
      v := t[k];
      if parentPtr↑ .child[v] = zeroPtr then
        new(newPtr);
        for k := 1 to δ do newPtr[j] ↑ .child[k]:= zeroPtr end for;
        parentPtr↑ .child[v]:= newPtr; parentPtr:= newPtr;
      else parentPtr:= parentPtr↑ .child[v]
      end if;
    end for;
    parentPtr↑ .child[t[n]] := onePtr;
  end for;
end buildTrie;
```

Fig. 15. A procedure that constructs a trie representing constraint relation R_j which is given as a collection of tuples. *new(newPtr)* creates a new node and makes *newPtr* point to it.

For each tuple in R_j there is a path from the root node to the '1' sink node. Conversely, each path from the root node to the '1' sink node corresponds to a tuple in R_j, so the trie is a *full* trie [21].

Fig. 16 shows an example of a trie that represents a constraint relation comprising fifteen 4-tuples with $\delta = 3$. Nodes are represented by circles. The number within a circle is the local (within-scope) number of the variable whose value selects a child of that node. The number beside an arc is the value that selects the child to which the arc is directed. As is usual in trie literature (e.g. [2,3]) arcs to the 0 sink node are not shown; the 0 sink node itself is not shown. All arcs at the bottom of Fig. 16 go to the 1 sink node, which is not shown.

For example, successive values 1, 1, 0, 2 take us from the root node at the top of Fig. 16 to the 1 sink node; this 4-tuple is in the constraint relation that the trie represents. Starting again at the root node, successive values 1 then 0 take us to the 0 sink node; there is no 4-tuple in this constraint relation that has 1, 0 as its first two values.

4.4.3. Trie implementation of partition search

Fig. 17 shows an implementation of partition search, using the following arrays:

V is such that $V[i]$ is the current value of the variable V_i.
vNext is such that the search will next try to instantiate V_i to $vNext[i]$.
ult is such that $ult[i]$ is the number of scopes that include the variable V_i.
which is a two-dimensional array of records that have two fields: *scopeNo* and *varNo*. For the gth scope that includes the variable whose global variable number is i, the record $which[i, g]$ contains the scope number and the local (within the gth scope) variable number of this variable. Array *which* is initialized so that, for all $1 \leqslant i \leqslant N$, $which[i, 1].scopeNo = j$ such that $V_i \in S_j^b$.
at is a currency-indicator array. $at[j, k]$ points to a node in the trie that represents R_j. During the search, $at[j, k]$ points to the current node that is associated with local (within scope j) variable k.

To illustrate the working of the routine shown in Fig. 17, suppose (as an initial example) that $n = 4$ and Fig. 16 trie represents R_1. Suppose also that the consistency checking procedure *doCheck*, which is shown in Fig. 18, always returns *consistent* = **true** when $i < 4$.

The loop at Lines 6–9 finds the next value v to be used in the instantiation of $V[i]$ at Line 11. In our initial example, $v = 0$ selects the leftmost child of the root node in Fig. 16 and we have $V[1] := 0$ at Line 11. After i has been incremented at Line 15, we have $j := 1$ and $k := 2$ at Line 16. Line 17 makes $at[1, 2]$ point to the leftmost child of the root node. After returning to Line 4, now with $i = 2$, the loop at Lines 6–9 selects $v = 2$, because $at[1, 2] \uparrow .child[0] = zeroPtr$ and $at[1, 2] \uparrow .child[1] = zeroPtr$. After returning again to Line 4, now with $i = 3$ and with $at[1, 3] = at[1, 2] \uparrow .child[2]$, the value $v = 2$ is again selected. During the next iteration, now with $i = 4$, $v = 1$ is selected because $at[1, 4] \uparrow .child[1] \neq zeroPtr$. Line 11 completes the instantiation of S_1 to the first tuple $\langle 1, 2, 2, 1 \rangle$ in R_1. At this time Line 11 also assigns $vNext[4] := 2$.

A result of this assignment to $vNext[4]$ is that when the search next returns to Line 4 with $i = 4$, the value $v = 2$ is selected by the loop at Lines 6–9, because $at[1, 4] \uparrow .child[2] \neq zeroPtr$. Line 11 completes the

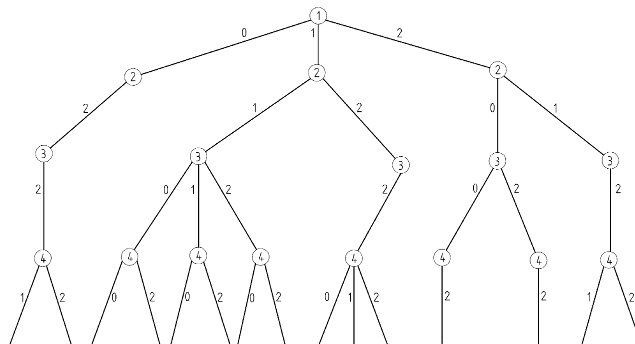

Fig. 16. An example of a trie representing a constraint relation that has $n = 4$.

```
1  partitionResequence; permuteTuples; Z := empty;
2  for j := 1 to q do at[j, 1] := root[j] end for;
3  i := 1; j := 1; k := 1; vNext[1]:= 0;
4  repeat
5      v:= vNext[i];
6      loop
7          if (v ≥ δ) or (at[j, k] ↑ .child[v] ≠ zeroPtr) then exit end if;
8          v := v + 1;
9      end loop;
10     if v < δ then
11         V[i] := v; vNext[i] := v + 1;
12         doCheck(i, consistent);
13         if consistent then
14             if i = N then
15                 include contents of array V in Z
14             else
15                 i := i + 1; vNext[i]:= 0; jPrevious:= j;
16                 j := which[i, 1].scopeNo; k := which[i, 1].varNo;
17                 if jPrevious = j then at[j, k]:= at[j, k − 1] ↑ .child[v] end if;
18             end if
19         end if
20     else
21         i := i − 1;
22         if i > 0 then j := which[i, 1].scopeNo; k := which[i, 1].varNo end if;
23     end if
24 until i = 0;
```

Fig. 17. Trie implementation of partition search.

```
1  procedure doCheck(in i: integer; out consistent: boolean);
2  begin
3      if ult[i] < 2 then consistent:= true return end if;
4      g:= 2;
5      loop
6          h:= which[i, g].scopeNo; k:= which[i, g].varNo;
7          childPtr:= at[h, k] ↑ .child[V[i]];
8          if childPtr = zeroPtr then exit end if;
9          if k < n then at[h, k + 1]:= childPtr end if;
10         if g = ult[i] then exit else g := g + 1 end if;
11     end loop;
12     consistent:= (childPtr ≠ zeroPtr)
13 end doCheck;
```

Fig. 18. Consistency checking procedure that is called in Line 12 of Fig. 17.

instantiation of S_1 to the second tuple $\langle 1, 2, 2, 2\rangle$ in R_1. Subsequently, when the search next returns to Line 4 with $i = 4$, we have $v = \delta$ at Line 5, which signifies that there are, at this stage, no further values to which $V[4]$ should be instantiated. The search backtracks to $j = 1$ and now chooses the middle child of the root node in Fig. 16. Continuing in this way, the search eventually instantiates S_1 to each in turn of the tuples in R_1.

Throughout the search, the instantiation of $V[i]$ is determined by the trie that represents R_j such that $V_i \in S_j^b$. Here we have, by initialization, *which*$[i, 1]$.*scopeNo* $= j$. Immediately after instantiation of $V[i]$, procedure *doCheck* checks consistency by using the trie that represents R_h for each h such that *which*$[i, g]$.*scopeNo* $= h$ and $2 \leqslant g \leqslant$ *ult*$[i]$. If there is no such R_h then procedure *doCheck* returns *consistent* = **true** at Line 3 in Fig. 18.

We now give a second example, which is emphatically not a continuation of our first example. In this second example, the trie in Fig. 16 no longer represents R_1. Instead, this trie represents R_6, such that $S_6 \in \Sigma_d$. Suppose that within scope S_6 the local (within S_6) variables $V_1, \ldots, V_4$ are the same as global variables V_5, V_8, V_9 and V_{10}, respectively. Suppose, moreover, that $S_6^b = \{V_9, V_{10}\}$. Initialization at Line 2 in Fig. 17 ensures that, at the time of instantiation of V_5, *at*[6,1] points to the root node of Fig. 16 trie. Suppose for example that V_5 is instantiated to 1. In this case Lines 7, 8 and 9 in Fig. 18 yield $at[6,2] := at[6,1] \uparrow .child[1]$, which points to the middle child of the root node in Fig. 16.

Suppose that procedure *doCheck* returns *consistent* = **true** after instantiation of V_5, V_6 and V_7. If V_8 is instantiated to $V_8 = 0$, then $at[6,2] \uparrow .child[0] = zeroPtr$, as can be seen at the middle child of the root node in Fig. 16. In this case procedure *doCheck* returns *consistent* = **false**, so i is not incremented at Line 15 of Fig. 17; instead the loop at Lines 6–9 of Fig. 17 selects the next value to which V_8 can be instantiated. Suppose that this value is 2. Immediately after instantiation $V[8] := 2$, procedure *doCheck* assigns $at[6,3] := at[6,2] \uparrow .child[2]$.

Continuing this example, suppose that procedure *doCheck* returns *consistent* = **true** after the instantiation $V[8] := 2$. At line 15 in Fig. 17, i is incremented to $i = 9$. In this example the global variable V_9 is the first in S_6^b, so at Line 16 we have *which*[9,1].*scopeNo* = 6. Furthermore, *which*[9,1].*varNo* = 3 because global variable V_9 is the same as local variable V_3 within scope 6. When the search returns to Line 4, Fig. 17, we have v:=0 at Line 5, because of the previous assignment *vNext*[i]:=0 at Line 15. At Line 7, *at*[6,3] has the value that was previously assigned to it at Line 9 of procedure *doCheck*, as mentioned above. Thus *at*[6,3] points to the right hand child of the middle child of the root node in Fig. 16. The only possibility now is the instantiation $V[9] := 2$, because $at[6,3]\uparrow.child[0] = zeroPtr$ and $at[6,3]\uparrow.child[1] = zeroPtr$.

Assuming that procedure *doCheck* returns *consistent* = **true** after instantiation of V_9, we have $j := 6$ and $k := 4$ at Line 15 of Fig. 17. In this case *jPrevious* = j, so we have $at[6,4] := at[6,3] \uparrow .child[2]$ at Line 17. The search continues similarly.

To see how this routine implements partition search, consider $S_j \in \Sigma_d$ with $j > 1$. Let k' be such that the local variable $V_{k'}$ in S_j is the first variable in S_j^b. The key idea is that when $i = G_j(k')$ at Line 5 of Fig. 17, procedure *doCheck* has previously ensured that $at[j, k']$ points to the root node of a subtrie of the R_j trie. This subtrie represents the set P_j of tuples that was defined in Section 4.2. If, after instantiation of $V_{e(j-1)}$, procedure *doCheck* never returns *consistent* = **false** until $V_{e(j)}$ has been instantiated, then Fig. 17 routine instantiates S_j^b to each in turn of the tuples in the set P_j, as in Fig. 12.

Because procedure *doCheck* is called after instantiation of each variable, Fig. 17 routine is not exactly an implementation of partition search as formulated in Fig. 12. Fig. 17 routine could be amended so as to call procedure *doCheck* only after instantiation of the whole of each S_j^b, thus becoming an exact implementation of partition search; but this would entail complications which would probably not enhance speed.

4.4.4. Reduction of decision diagrams

When arity is high and constraint relations have many tuples, trie representation makes heavy demands on memory. Memory requirements can be reduced substantially by removing duplicate and redundant nodes as explained by Bryant [18]. Bryant's reduction algorithm [17, Fig. 4] can readily be extended to the non-binary case where a node may have more than two children; we will not give details because the extension is so straight-forward that it does not require explanation. Henceforward we will use BDD to mean a fully reduced ordered binary decision diagram (OBDD) as defined in [18]. Henceforward an MDD is a fully reduced ordered multivalued decision diagram [84, p. 215]. The MDD shown in Fig. 19, which has 10 non-terminal nodes, represents the same constraint relation as the Mtrie shown in Fig. 16, which has 17 non-terminal nodes.

The search routine shown in Fig. 17 can easily be amended so as to work with BDDs or MDDs instead of tries. Amendment, which is required because redundant nodes have been deleted from BDDs and MDDs, is completed by replacing

- Line 17 in Fig. 17 by the fragment shown in Fig. 20a and
- Line 9 in Fig. 18 by the fragment shown in Fig. 20b.

Array *xt* is of the same type as array *at*. An element of array *xt* may point to a redundant node that has temporarily been re-instated. For example, suppose that the MDD in Fig. 19 represents constraint relation R_1.

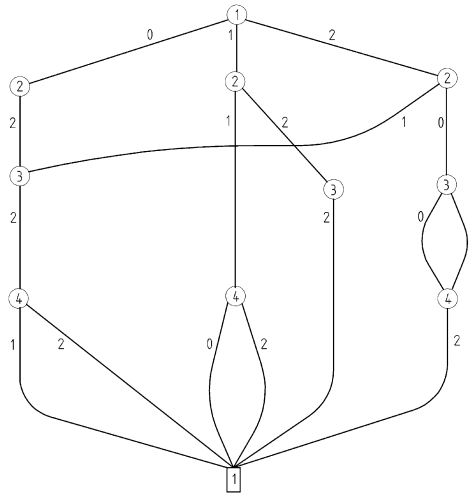

Fig. 19. An MDD representing the constraint relation that is represented by Fig. 16. The '1' sink node is shown at the bottom of the diagram. The '0' sink node and arcs pointing to it are not shown.

a

```
if j = jPrevious then
    childPtr:= at[j, k − 1] ↑ .child[v];
    if childPtr↑ .varNo > k then
        for u:= 0 to δ − 1 do xt[j, k] ↑ .child[u]:= childPtr end for;
        childPtr:= xt[j, k];
    end if;
    at[j, k]:= childPtr
end if;
```

b

```
if (k < n) and (childPtr↑ .varNo > k + 1) then
    for u:= 0 to δ − 1 do xt[x, k + 1] ↑ .child[u]:= childPtr end for;
    childPtr:= xt[x, k + 1];
end if;
```

Fig. 20. Fragments used in amendment of Figs. 17 and 18.

Consider the situation where, immediately after execution of Line 16 in Fig. 17, we have $j = 1$, $v = 1$, $i = 3$, and $k = 3$. Suppose that *at*[1,2] points to the middle child of the root node in Fig. 19. In this case, $at[1,2] \uparrow .varNo = 2$ and $at[1,2] \uparrow .child[1] \uparrow .varNo = 4$. Fragment (a) in Fig. 20 makes all the children of extra node $xt[1,3] \uparrow$ to be $at[1,2] \uparrow .child[1]$. This extra node is redundant because all of its children are the same. Just before the end of fragment (a), $xt[1,3]$ is assigned to $at[1,3]$ for indispensable use at Line 7 in the next iteration of the **repeat** loop in Fig. 17. Fragment (b) can be understood similarly.

4.5. Relationship between dual and partition search

Every constraint satisfaction problem has a *dual* in which the variables are $S_1, \ldots, S_q$ and there is a binary constraint between variables S_h and S_j such that $S_h \cap S_j$ is not empty. For a binary constraint between S_h and S_j, the constraint relation is $R_h \bowtie R_j$ [86]. Dual representation was originally introduced in database theory [53] and has been used to transform non-binary into binary constraint satisfaction problems [9,27,24,44,67]. *Dual search* is a search for one or more solutions of the original problem, working with dual variables and dual constraints instead of original variables and original constraints. Sections 5.6 and 5.7 report experiments

with Cn dual search implemented as in Fig. 1 except that incremental restoration is used instead of push/pop save/restore.

If there is a dual constraint between S_h and S_j then R_j can be replaced by $R_j \ltimes R_h$, as in Section 3.1. During dual search, instantiation of S_h to a tuple v_h replaces the operation $R_j \ltimes R_h$ by $R_j \ltimes v_h$. As a result of deletion of tuples from R_j, there may be tuples in other dual domains that do not concatentate with any tuple in R_j and can therefore be deleted. Deletion of tuples may propagate.

To simplify the following paragraphs, suppose that dual search instantiates dual variables in the static sequence $S_1, S_2, \ldots, S_q$, and that redundant pairs have not been removed. Suppose also that S_j is instantiated to a tuple v_j at a time when, for all $1 \leqslant h < j, S_h$ has previously been instantiated to v_h. The set of solutions consistent with these instantiations of $S_1, \ldots, S_j$ is

$$Z' = v_1 \bowtie v_2 \bowtie \cdots \bowtie v_{j-1} \bowtie v_j \bowtie R_{j+1} \bowtie \cdots \bowtie R_q$$

Z' is empty if $v_h \bowtie v_j$ is empty for any h such that $h < j$ and $S_h \cap S_j \neq \emptyset$. At the time of instantiation of S_j, the previous operation $R_j := R_j \ltimes v_h$ has deleted from R_j every tuple v_j such that $v_j \bowtie v_h$ is empty. Dual search does not instantiate S_j to any tuple that has been deleted from R_j. Thus dual search avoids instantiations of S_j that would yield empty Z' and so be a waste of time. Dual search avoids futile instantiations by using binary dual constraints to delete tuples from dual domains.

Partition search does not use binary constraints between dual variables and does not delete tuples from dual domains. Whereas dual search deletes from R_j every tuple to which S_j should not be instantiated, partition search uses instead a data structure (e.g. hash or DD) to identify every tuple $t \in R_j$ to which S_j should be instantiated when $W_{e(j-1)}$ has been instantiated to $w_{e(j-1)}$. At this time, S_j^a has already been instantiated; partition search completes the instantiation of S_j by instantiating S_j^b. Scope S_j should be instantiated to a tuple t in R_j only if t concatenates with $w_{e(j-1)}$ in the join $w_{e(j-1)} \bowtie t$. The partition search data structure provides rapid access to tuples in the set

$$\{t \in R_j | w_{e(j-1)} \bowtie t \neq \emptyset\} = \{t \in R_j | t[W_{e(j-1)} \cap S_j] = w_{e(j-1)}[W_{e(j-1)} \cap S_j]\} = \{t \in R_j | t[S_j^a] = w_{e(j-1)}[S_j^a]\}$$

remembering that $S_j^a = W_{e(j-1)} \cap S_j$. This (hash or DD) data structure is set up before commencement of partition search and remains unchanged thereafter. Before commencement of dual search there is no analogous construction of a data structure.

For each possible instantiation, $x = w_{e(j-1)}[S_j^a]$, of S_j^a, the partition search data structure provides rapid access to tuples in $\sigma_{S_j^a=x}(R_j) = \{t \in R_j | t[S_j^a] = x\}$. This data structure depends only on the tuples within R_j, regardless of which tuples are, or are not, in other constraint relations. In dual search, deletion of a tuple from a constraint relation depends on absence of specific tuples from other constraint relations.

Partition resequencing is intended to make S_j^a as large as possible, so as to make $\sigma_{S_j^a=x}(R_j)$ as small as possible. This is expected to make partition search faster by lessening the number of distinct instantiations of S_j^b. Partition search requires partition resequencing, but dual search does not. Dynamic variable ordering can be used with dual search but not with partition search. The partition search data structure works only with a static (i.e. unchanging) instantiation sequence.

Another difference is that semijoin reduction can be applied before commencement of search; but partition search cannot be used for preprocessing because P_j is defined in terms of $w_{e(j-1)}$.

Dual search must restore tuples to constraint relations when the absence of these tuples has lost logical justification. Save/restore operations are particularly burdensome when the dual domains $R_1, \ldots, R_q$ have high cardinality. Because partition search does not delete anything from anywhere it involves no save/restore operations. The application of hash joins or DDs in partition search is practical because constraint relations remain unchanged throughout the search. The membership of constraint relations does not remain unchanged throughout dual search.

Compared with dual search, partition search may make more instantiations, but do less work associated with each instantiation. To explore this we now consider dual S1 reduction search, as in [8], without dynamic variable ordering and without eliminating redundant dual constraints. In other words, we consider the original version of forward checking [57, Fig. 5] applied to tuple instantiation and dual constraints. We apply partition resequencing, which can be expected to speed up dual search for the same reasons that it speeds up partition

search. After this static reordering and renumbering of dual variables, our dual search again instantiates scopes in the static sequence $S_1, S_2, \ldots, S_q$.

The following paragraphs relate only to hash-join partition search because this instantiates $S_1^b, S_2^b, \ldots, S_{q_d}^b$. The DD implementation of partition search in Section 4.4.3 instantiates one variable at a time, which would complicate comparison with dual search.

After dual S1 search has instantiated a scope S_h to a tuple $u \in R_h$, the forward checking routine is:

for each $j > h$ such that $S_h \cap S_j$ is non empty **do** $R_j := R_j \ltimes u$ **end for**;

The semijoin reduction $R_j := R_j \ltimes u$ can be written explicitly

$$R_j := \{t \in R_j | t[S_j \cap S_h] = u[S_j \cap S_h]\}$$

and at the time of instantiation of S_j this can be written

$$R_j := \{t \in R_j | t[S_j \cap S_h] = w_{e(j-1)}[S_j \cap S_h]\}$$

At this time, forward checking has previously been performed for all $h < j$, thereby reducing R_j to

$$R_j = \{t \in R_j^o | (\forall h_{(h<j) \wedge (S_h \cap S_j \neq \emptyset)})(t[S_j \cap S_h] = w_{e(j-1)}[S_j \cap S_h])\}$$

where R_j^o is the set R_j given initially as part of the problem formulation. After partition resequencing we have

$$\cup_{(h<j) \wedge (S_h \cap S_j \neq \emptyset)}(S_j \cap S_h) = S_j^a$$

whence, at the time of instantiation of S_j,

$$R_j = \{t \in R_j^o | t[S_j^a] = w_{e(j-1)}[S_j^a]\} = R_j^o \ltimes w_{e(j-1)}$$

Dual search instantiates S_j to a tuple in this reduced set R_j. When hash-join partition search instantiates S_j^b it thereby completes the instantiation of S_j to a tuple in $R_j^o \ltimes w_{e(j-1)}$ which is the same as for dual search.

After instantiating S_j^b, hash-join partition search calls procedure *consistent* which vetoes this instantiation if there is any $S_x \in \Theta_j$ such that $R_x \ltimes w_{e(j)}$ is empty. In S1 dual search, procedure *reduce* vetoes the instantiation of S_j if there is any k (not restricted to $S_k \in \Theta_j$ and $S_k \subset W_{e(j)}$) such that $k > j$, $S_k \cap S_j \neq \emptyset$ and $R_k \ltimes w_{e(j)}$ is empty. Hence we see that the set of search-space nodes visited by S1 dual search is a subset of the set of search-space nodes visited by hash-join partition search.

On the other hand, dual search does much more work at each node in the search space. Hash-join partition search implements $R_j \ltimes w_{e(j-1)}$ using a single hash join, whereas dual search achieves this by means of a separate semijoin operation for each h such that $h < j$ and $S_h \cap S_j \neq \emptyset$. Each semijoin operation involves serial search seeking deletable tuples.

4.6. Relationship between partition search and variable elimination

We now introduce a development of partition search that makes the search backtrack-free, which here means that the first q_d iterations of the **repeat** loop in Fig. 12 find a solution if one exists. This development is not immediately practical; its immediate purpose is to elucidate the relationship between partition search and variable-elimination algorithms [47,48].

Partition search at Line 15 in Fig. 21 works in the usual sequence $1, 2, \ldots, q_d$. A preliminary process at Lines 5–14 works in the reverse sequence. In [24, Fig. 8.2] the analogous preliminary process puts constraint relations into buckets, with one bucket corresponding to each variable. Fig. 21 works with sets $\widehat{\Theta}_1, \ldots, \widehat{\Theta}_j, \ldots, \widehat{\Theta}_{q_d}$, which can be regarded as buckets. Line 3 initializes $\widehat{\Theta}_j$ to include every constraint relation R_x such that S_x is in Θ_j as defined in Section 4.2.

To see why partition search at Line 15 in Fig. 21 is backtrack-free, assume inductively that $W_{e(j-1)}$ has been instantiated and that $w_{e(j-1)} = z[W_{e(j-1)}]$ for some solution $z \in Z$. This assumption implies $w_{e(j-1)} \in \widehat{R}_1 \bowtie \cdots \bowtie \widehat{R}_{j-1}$, whence $w_{e(j-1)}[\widehat{S}_h] \in \widehat{R}_h$ for all $1 \leqslant h < j$. Line 11 ensures that $\pi_{\widehat{S}_j^a}(\widehat{R}_j) \in \widehat{\Theta}_h$ for some $h \in \{1, \ldots, j-1\}$. A subsequent join at Line 6 ensures that $\widehat{R}_h$ is the join of $\pi_{\widehat{S}_j^a}(\widehat{R}_j)$ with other relations in $\widehat{\Theta}_h$. Thus $w_{e(j-1)} \in \widehat{R}_1 \bowtie \cdots \bowtie \widehat{R}_{j-1}$ implies $w_{e(j-1)} \in \pi_{\widehat{S}_j^a}(\widehat{R}_j) \bowtie$ (join of other relations). Hence, because $\widehat{S}_j^a$

```
1  partitionResequence; permuteTuples;
2  for j := 1 to q_d do
3      Θ̂_j := {R_j} ∪ {R_x | (S_x ∩ S_j^b ≠ ∅) ∧ (S_x ⊂ W_e(j))}
4  end for;
5  for j := q_d downto 1 do
6      R̂_j :=  ⋈_{(∀y)(R_y ∈ Θ̂_j)} R_y;
7      if R̂_j = ∅ then
8          report that there is no solution; terminate
9      elsif j > 1 then
10         define Ŝ_j = scope of R̂_j; define Ŝ_j^a = Ŝ_j − S_j^b;
11         include π_{Ŝ_j^a}(R̂_j) in Θ̂_h where h is such that
12             (Ŝ_j^a ∩ S_h^b ≠ ∅) ∧ (Ŝ_j^a ⊂ W_e(h))
13     end if
14 end for;
15 perform partition search using R̂_1, R̂_2, ..., R̂_{q_d} instead of
16     the original constraint relations R_1, R_2, ..., R_q;
```

Fig. 21. Backtrack-free partition search. All variables in S_j^b are eliminated by the projection operation in Line 11.

is the scope of $\pi_{\widehat{S}_j^a}(\widehat{R}_j)$, we have $w_{e(j-1)}[\widehat{S}_j^a] \in \pi_{\widehat{S}_j^a}(\widehat{R}_j)$. Line 7 has ensured that $\widehat{R}_j$ is non-empty, so there is a tuple $u \in \widehat{R}_j$ such that $u[\widehat{S}_j^a] = w_{e(j-1)}[\widehat{S}_j^a]$. Partition search at Line 15 instantiates $\widehat{R}_j$ to u only if $u[\widehat{S}_j^a] = w_{e(j-1)}[\widehat{S}_j^a]$. The construction of $\widehat{R}_j$ is such that $u[S_x] \in R_x$ for all $R_x \in \widehat{\Theta}_j$. Therefore the extension $w_{e(j)} = u \bowtie w_{e(j-1)}$ is such that $w_{e(j)} = z[W_{e(j)}]$ for some solution $z \in Z$; c.f. [26, Theorem 2.12].

When constraint scopes are random, $\widehat{R}_j$ may be so large that this approach is not practical without development. If limits are imposed on the size of relations then the search is not guaranteed to be backtrack-free, but performance may nevertheless be enhanced [28].

5. Experiments

5.1. Generation of random problems

Frost and Dechter [31] explain that testing algorithms on randomly generated problems has the advantage of facilitating systematic experimentation over large statistical populations but has the possible disadvantage of not reflecting the real challenges of practical applications. The present paper works mainly with random problems because these enable exploration of trends of algorithm performance when problem parameters are changed. Randomly generated problems can actually be more difficult than related practical applications. For example, determination of subgraph isomorphism [57] is a binary constraint satisfaction problem that is soluble only on a very small scale with purely random unlabeled graphs. Molecule-matching applications [4,16] use additional information that enables completion of search within reasonable time limits.

A random problem is trivial as a benchmark for search algorithms if Z can be shown to be empty without the need for any search [1,34,85]. A random problem is also unsatisfactory if it has so many solutions that one of these may be found very quickly, making the benchmark too easy. Between these extremes there is a crossover point where the parameters N, n, etc., are such that the probability of a problem having a solution is 0.5. From the viewpoint of algorithm design, it is natural to attack the hardest problems, which occur near crossover points [65,70]. Confining attention to crossover points may, however, be unwise. This is because, in the worst case of an NP-hard problem, algorithms can only be expected to work within a reasonable time on problems that may be too small to be of practical interest. Practical applications that happen to be at crossover points may never have serviceable algorithms. On the other hand, it is very easy to find constraint satisfaction problems that are far away from crossover points and are at the same time far beyond the capabilities of

available algorithms. With such problems it may be possible to make greater practical progress than will ever be possible near crossover points.

We use a generator that randomly chooses a set Σ of q scopes such that any two scopes in Σ differ in at least one variable. Moreover, as for example in [57,67,76], our generator ensures that Σ cannot be partitioned into a disjoint set of subsets such that the constraint satisfaction problem could be decomposed into smaller problems, one per subset, thereby reducing overall complexity. To prevent another simple short cut, our generator also ensures that every variable belongs to at least two scopes.

Each experimental trial starts afresh with new random scopes and new random constraint relations. Constraint relations are generated using a hybrid of methods used previously [15,80]. This involves both of the following procedures:

n-random. For each scope in Σ generate a given number, J, of distinct pseudo-random n-tuples of integers in the range $0, 1, \ldots, \delta - 1$. Let R_j^J be the set of n-tuples thus generated on scope S_j. Note that J is the cardinality of R_j^J after removal of duplicates.

N-random. Generate a set Y that consists of a given number, H, of distinct pseudo-random N-tuples of integers in the range $0, 1, \ldots, \delta - 1$. The set Y of N-tuples is a relation on $V = \{V_1, \ldots, V_i, \ldots, V_N\}$. Let R_j^H be the relational projection $\pi_{S_j}(Y)$.

For each $S_j \in \Sigma$, our generator generates constraint relations $R_j = R_j^H \cup R_j^J$. The projection operation removes duplicates and therefore the total number of n-tuples in R_j is generally less than $K = H + J$.

Section 5.4.2 will provide experimental evidence that increasing the ratio H/K makes enumeration of Z harder. Working with $H > 1$ ensures that Z cannot be proved to be empty without performing any search. Working with $H > 1$ and $J > 1$ we can model a wider range of practical applications than would be possible with $H = 0$.

When working with $H > 1$ we always enumerate Z completely. If, instead, the search terminated after finding a single solution then the search could be faster for larger H because there would be more solutions. Enumeration of the whole of Z, as in [85], is appropriate for various possible applications [77,79,80].

5.2. Elective and implied instantiation

All our experiments with reduction search use both dynamic and static variable ordering. For S-reduction and C-reduction we use different versions of the dynamic ordering procedure *choose* (in Fig. 1). The difference can be introduced in terms of the distinction [51] between elective and implied instantiation of a variable. *Elective* instantiation means instantiation by the search algorithm, as at Line 13 in Fig. 1. *Implied* instantiation means instantiation by domain reduction that removes all except one value from a domain.

For C-reduction search we use a version of procedure *choose* that returns *allSingleValued* = **true** when all domains are indeed single valued. When *allSingleValued* = **false** this version always returns i such that the cardinality of D_i is greater than one. Here a single-valued domain is regarded as instantiated, regardless of whether this instantiation is elective or implied. Using this version of procedure *choose*, reduction search may include an N-tuple in Z when many fewer than N variables have been instantiated electively. This may be very much faster than waiting until all of the N variables have been instantiated electively.

With S-reduction we use a version of procedure *choose* that returns *allSingleValued* = **true** only when all variables have been electively instantiated. This version may return i such that V_i has been implied-instantiated so D_i is already single-valued. The reason for this is that procedure *reduce*, when called after elective instantiation of an implied-instantiated variable, may remove values from further domains, thus speeding up the search.

When $n = 3$, $N = 50$, $q = 80$, $\delta = 5$, $J = 48$ and $H = 0$, distributive S2-reduction search is slowed down by a factor of 43.25 if the C version of procedure *choose* is used instead of the S version. When $n = 3$, $N = 128$, $q = 240$, $\delta = 2$, $J = 6$ and $H = 0$, distributive C1-reduction search is slowed down by a factor of 1.926 if the S version is used instead of the C version. These results are the average over 50 trials, near crossover points, of the time to find one solution or report that there is none, without preprocessing.

5.3. Comparisons without preprocessing

5.3.1. Reduction search and hash-join partition search

Table 3 shows results of experiments done with $H = 0$ and with other parameters chosen so that the probability of existence of a solution is near to 0.5. Table 3a shows the average time in seconds, over 50 trials, to find one solution or report that Z is empty, with no preprocessing. The leftmost column contains row numbers that identify rows in Table 3b. For each row number in Table 3b there are three rows that have that number in Table 3a: for all of these the row in Table 3b shows the values of N, q, etc., that were used. The column headed P(sol) shows the probability that a solution exists. The second column in Table 3a shows the arity of the constraints and the third shows $\rho = nq/N$, which is the average degree of the variables. The fourth column indicates which implementation was used:

D signifies distributive reduction search, as in Section 2.5,
T signifies simple tabular reduction search,
B signifies GAC2001 [14] with *currentSupport* not re-initialized at the beginning of every invocation of the domain reduction procedure. Instead, *currentSupport* is subject to incremental save/restore.

The rightmost six columns show results obtained with S1, S$n-1$ and Cn reduction. The column headed av in Table 3b shows average times for partition search. Columns headed sd show the (estimated) standard

Table 3
Results of 50 trials near crossover points

Row	n	ρ	DTB	S1		S$n-1$		Cn	
				av	sd	av	sd	av	sd
(a)									
1	3	30	D	**0.106**	0.007	0.60	0.04	3.36	0.25
1	3	30	T	9.39	0.70	6.56	0.48	5.56	0.41
1	3	30	B	9.52	0.69	9.98	0.74	9.89	0.73
2	3	4.8	D	5.52	0.91	1.24	0.18	1.64	0.24
2	3	4.8	T	22.44	3.70	2.02	0.30	**1.06**	0.15
2	3	4.8	B	33.28	5.60	3.37	0.50	1.69	0.25
3	4	28	D	**0.27**	0.02	3.91	0.34	15.7	1.3
3	4	28	T	75.1	6.3	23.4	1.8	26.9	2.2
3	4	28	B	101.4	8.19	91.60	8.15	94.28	8.39
4	4	3.7	D	93.9	20.2	2.10	0.23	3.62	0.40
4	4	3.7	T	309.1	72.6	1.57	0.17	**1.41**	0.15
4	4	3.7	B	503.5	117	4.16	0.46	2.93	0.33
5	5	28	D	**0.31**	0.02	7.31	0.68	17.6	1.6
5	5	28	T	182.2	16.4	31.2	2.6	31.8	2.7
5	5	28	B	251.5	21.6	237.8	22.4	237.6	22.3
6	5	3.5	D	123.0	36.7	0.51	0.05	0.75	0.08
6	5	3.5	T	341.1	9.7	0.32	0.03	**0.30**	0.03
6	5	3.5	B	517.4	159.0	0.86	0.09	0.70	0.07

(b)

Row	n	N	q	δ	J	P(sol)	av	sd	μ
1	3	10	100	10	792	0.46	0.263	0.018	3
2	3	50	80	5	48	0.54	211.5	54.9	3
3	4	10	70	8	3035	0.54	0.457	0.033	4
4	4	50	46	4	59	0.46	5.39	1.69	3
5	5	10	56	6	5610	0.50	0.469	0.024	5
6	5	50	35	3	52	0.50	0.586	0.082	4

(a) Experimental timings for reduction search; the fastest time is shown in bold print. (b) Parameter values and timings for partition search.

deviation of the average, not of the variate. Here, and throughout Section 5.3.1, *partition search* means partition search implemented using hash joins as in Section 4.3. Experiments with other implementations of partition search will be reported in Section 5.3.2.

As in [15], there is a trade-off between visiting more nodes of the search tree but spending less time on pruning operations at each node, or instead spending more time on pruning operations so that fewer nodes are visited. When ρ is high, distributively implemented S1-reduction search is the fastest in Table 3. In each case where ρ is low, either n or δ is so low that the conditions listed at the end of Section 4.2 are not satisfied, and therefore partition search is not competitive.

Because it is not practical to explore the entire space defined by parameters N, n, δ, q and K, we select combinations of parameter values to provide examples of performance of algorithms. Table 3 has provided examples where partition search is not competitive. In Table 4, the first row provides an example showing that partition search can be competitive when $n = 4$. In the second row partition search is more strongly competitive because δ is greater than in the first row. This increase in δ yields better compliance with conditions listed at the end of Section 4.2. The third row has been chosen to show that partition search can be competitive when $\delta = 3$; the fourth row has a greater value of n. The time taken by reduction search indicates non-triviality even though $P(\text{sol}) = 0$ in all four examples in Table 4.

Throughout Tables 3 and 4, GAC2001 is slower than simple tabular reduction. To illustrate one reason for this, let us say that when the first variable is instantiated, the depth of search is 0; when the next variable is instantiated, the depth of search is 1, and so on. For Cn reduction in Row 5 of Table 3, Fig. 22a shows, on a log scale, the number of calls of the domain reduction procedure at each depth. This histogram is the same for GAC2001 and simple tabular reduction search. Fig. 22a also shows the average $|R_j|$ versus depth for simple tabular reduction. Fig. 22b shows the number of evaluations of $(\forall k_{1\leqslant k\leqslant n_j})(t_k \in D_{G_j(k)})$ at each depth for GAC2001 and for simple tabular reduction search (STR). The greatest number of calls of the domain reduction procedure occurs at depth 5, where for simple tabular reduction search the average $|R_j| = 32.99$ and for GAC2001 the average $|R_j| = 5610$. At depth 5 in Fig. 22b the average numbers of evaluations of $(\forall k_{1\leqslant k\leqslant n_j})$

Table 4
Times in seconds in 50 trials with $N = 50$ and $H = 0$

Simple tabular reduction				GAC2001				Partition search	
S$n-1$		Cn		S$n-1$		Cn			
av	sd	av	sd	av	sd	av	sd	av	sd
1.925	0.207	2.285	0.244	6.584	0.719	4.510	0.482	**0.467**	0.097
2.293	0.300	3.469	0.414	12.72	1.643	7.662	0.926	**0.093**	0.029
0.233	0.023	0.291	0.028	0.918	0.096	0.603	0.059	**0.067**	0.012
3.753	0.317	5.437	0.481	31.11	4.452	27.63	3.047	**0.084**	0.008

Row 1: $\delta = 6$, $n = 4$, $q = 46$, $J = 125$. Row 2: $\delta = 12$, $n = 4$, $q = 46$, $J = 400$. Row 3: $\delta = 3$, $n = 6$, $q = 30$, $J = 80$. Row 4: $\delta = 3$, $n = 10$, $q = 18$, $J = 600$.

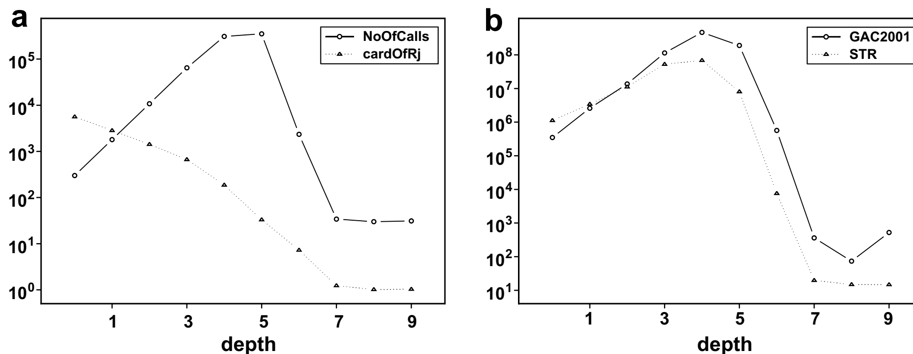

Fig. 22. For Row 5 of Table 3: (a) number of calls and average $|R_j|$ versus depth; (b) number of tuple checks versus depth.

$(t_k \in D_{G_j(k)})$ for GAC2001 and simple tabular reduction are 1.901538192×10^8 and 8.0019250×10^6, respectively. For GAC2001 at depth 5 in Fig. 22b, the average number of tuples checked (in Line 4 of Fig. 4) for $v = t_h$ is 9.216534822×10^8.

5.3.2. Decision diagram implementation of partition search

Table 5 shows results of experiments with partition search implemented using multivalued tries (Mtries), MDDs, binary tries (Btries) and BDDs as in Section 4.4. As in Table 3a, the leftmost column in Table 5 contains a row number. A row number in the range 1–6 selects the row in Table 3b that contains values of N, q, etc., that were used. A row number in the range 7–12 selects the row in Table 6 that contains values of n, q, etc., that were used. Results for hash-implemented partition search are included in Table 3b and in Table 6 because there is not sufficient space in Table 5. The second column of Table 5 indicates which results are shown in further columns. Entries in the second column are:

Tt means total time, in seconds, over 50 trials, to find one solution or report that Z is empty, with no preprocessing. In all our experiments with partition search, total time includes set-up time which is explained below.
Nv means the number of search-tree nodes visited during the search; this is the same as the number of invocations of the consistency-checking procedure during the search.
Nc means the total (over all constraint relations) of the number of nodes in the decision diagrams or tries.
St% is set-up time as a *percentage* of total time. *Set-up time* is the time taken by preliminary computation before the search, starting with constraint relations which are given as explicit lists of tuples. Set-up time includes time for partition resequencing and for permuting the sequence of values in each tuple in each constraint relation. For trie implementation, set-up time includes time taken to construct a trie for each constraint relation. For DD implementation, set-up time includes time taken to construct tries and reduce them to DDs. For binary tries and BDDs, set-up time also includes time for conversion to binary representation. For hash-join implementation, set-up time includes time taken to sort tuples into buckets and construct the index.

The Nv results confirm that Mtries and MDDs have the same search space. Btries and BDDs also have the same search space. In every row Mtries are faster than MDDs, Btries and BDDs. Moreover, MDDs are faster than BDDs, in concurrence with Wegener's observation [84, p. 216] that MDDs are useful when it is natural to work with multivalued variables. Note that when Nc for BDDs exceeds Nc for Mtries, BDDs may nevertheless have a smaller overall memory requirement because each BDD node has provision for only two children.

In Row 2 of Table 5, Nv is large and St% is therefore small. Row 7 provides an example of the opposite extreme, in which Nv is small and St% is large. Nv is smaller in Row 12 than in Row 11, but St% is higher in Row 12 because nearly four times as many constraint relations are processed before commencement of search. Tt is smaller for MDDs than for Btries in Row 8, where St% is small; but MDDs are slower than Btries in Row 10, where the time taken to reduce Mtries to MDDs is high in proportion to Tt.

Mtrie implementation of partition search is faster than hash-join implementation (Table 3b) only in Rows 1, 2, 4 and 6 of Table 5. However, Mtrie implementation of partition search is slower than reduction search (Table 3a) in all of Rows 1–6 of Table 5. Experimentation suggests that for values of parameters n, q, etc., such that partition search is faster than reduction search, hash-join implementation is faster than Mtrie implementation. Henceforward, in subsequent sections, it is to be understood that *partition search* means hash-join partition search.

5.4. Preprocessing

5.4.1. Semijoin reduction preprocesing

In the bottom row of Table 4, partition search was faster than reduction search by a factor of 44.7, but this was without preprocessing. In some cases reduction search can be speeded up substantially by executing a semijoin reduction routine, until convergence, prior to the commencement of search. This attempt to reduce the cardinality of constraint relations is an example of *preprocessing*.

Table 5
DD implementation of partition search: results of 50 trials with parameters as shown in Tables 3a and 6

Row	Res	Mtrie		MDD		Btrie		BDD	
		av	sd	av	sd	av	sd	av	sd
1	Tt	**0.158**	0.010	0.305	0.017	0.649	0.035	1.066	0.054
1	Nv	0.2M	13,791	0.2M	13,791	0.4M	23,415	0.4M	23,415
1	Nc	11,100	0.0	7502	3.69	0.2M	12.29	23,878	7.49
1	St%	13.49		12.93		12.26		20.27	
2	Tt	**48.30**	11.75	67.49	16.50	107.1	27.1	161.7	41.1
2	Nv	91.1M	21.9M	91.1M	21.9M	179M	44.5M	179M	44.5M
2	Nc	2313	1.233	1858	21.7	11,420	4.8	4.717	3.070
2	St%	0.009		0.002		0.010		0.017	
3	Tt	**0.484**	0.038	0.982	0.070	1.786	0.126	2.750	0.184
3	Nv	0.3M	28,588	0.3M	28,588	0.4M	40,196	0.4M	40,196
3	Nc	40,949	0.13	14,489	4.7	276,721	10.2	44,525	6.1
3	St%	18.11		23.77		23.03		30.01	
4	Tt	**4.197**	1.264	5.813	1.751	7.052	2.175	10.18	3.12
4	Nv	7.8M	2.4M	7.8M	2.4M	13M	4.0M	13M	4.0M
4	Nc	2878.8	2.7	1460	1.1	7043	5.8	2889	2.3
4	St%	0.135		0.253		0.064		0.278	
5	Tt	**0.515**	0.030	1.058	0.057	2.151	0.115	3.388	0.137
5	Nv	0.2M	17,659	0.2M	17,659	0.3M	27,231	0.3M	27,231
5	Nc	87,047	0.9	17,895	1.2	554,672	25.3	77,004	8.0
5	St%	34.39		42.65		38.85		54.04	
6	Tt	**0.436**	0.061	0.597	0.081	0.700	0.089	1.030	0.132
6	Nv	0.8M	106,871	0.8M	0.1M	1.3M	0.2M	1.3M	0.2M
6	Nc	2759	2.1	1373	1.2	6536	4.4	2761	2.1
6	St%	0.525		2.681		0.327		1.444	
7	Tt	**0.485**	0.007	1.371	0.004	2.259	0.025	4.829	0.008
7	Nv	340.7	155.9	340.7	155.9	337.3	67.9	337.3	67.9
7	Nc	0.3M	14.5	56,596	0.0	1.6M	59.9	0.2M	14.5
7	St%	99.99		99.99		99.99		99.99	
8	Tt	**14.06**	2.811	18.63	3.45	48.25	9.24	51.52	9.57
8	Nv	14M	3.0M	14M	3.0M	44M	9.2M	44M	9.2M
8	Nc	0.3M	23.5	73,464	14.9	1.7M	81.9	0.3M	42.1
8	St%	1.876		9.381		2.369		8.460	
9	Tt	**3.278**	0.130	15.56	0.18	11.80	0.48	39.91	0.43
9	Nv	0.5M	79,638	0.5M	79,638	1.3M	0.2M	1.3M	0.2M
9	Nc	2.9M	54.8	0.5M	45.7	9.6M	163.7	1.5M	115.2
9	St%	72.68		92.60		75.03		93.10	
10	Tt	**3.556**	0.043	27.89	0.04	13.98	0.36	63.82	0.08
10	Nv	0.1M	4471	0.1M	4471	0.4M	16,830	0.4M	16,830
10	Nc	3.7M	40.8	1.3M	53.4	16.3M	165.3	5.2M	161.9
10	St%	94.04		99.01		94.05		98.82	
11	Tt	**2.205**	0.077	14.88	0.09	8.246	0.324	33.64	0.22
11	Nv	0.3M	57,896	0.3M	57,896	1.0M	0.2M	1.0M	0.2M
11	Nc	1.9M	31.1	0.7M	40.5	8.49M	109.7	2.7M	124.8
11	St%	77.87		96.42		80.93		96.00	

Table 5 (*continued*)

Row	Res	Mtrie		MDD		Btrie		BDD	
		av	sd	av	sd	av	sd	av	sd
12	Tt	**6.979**	0.092	55.81	0.06	27.56	0.73	128.4	0.26
12	Nv	82,779	2946	82,779	2946	0.3M	11,397	0.3M	11,397
12	Nc	7.5M	74.9	2.5M	82.3	32.6M	275.8	10.4M	315.9
12	St%	96.61		99.34		96.00		99.2	

M means millions.

Table 6
Parameter values used in Table 5, together with timings for hash-implemented partition search

Row	n	q	δ	J	av	sd	St%	μ
7	6	35	6	20,000	0.303	0.006	99.63	5
8	6	35	10	5000	4.969	1.152	1.467	4
9	10	35	6	20,000	0.532	0.028	65.56	5
10	10	35	10	20,000	0.344	0.007	93.91	4
11	10	18	10	20,000	0.232	0.013	72.86	4
12	10	70	10	20,000	0.656	0.007	96.30	4

All of the six rows have $N = 50$, $H = 0$ and $P(\text{sol}) = 0$. The columns headed av and sd show the average, and standard deviation (of the average), of total time over 50 trials with hash-implemented partition search. For these 50 trials, the column headed St% shows average set-up time as a percentage of total time.

For preprocessing we use procedure ZM, with Bloom filter implementation, because this has very much smaller memory requirements than PW-AC, as explained in Section 3.3. In all our experiments, Bloom filter array B is always an array of arrays of 2^{17} bits, so $\zeta = 5$ when $\delta = 10$. The times shown in Table 7 are for procedures ZM (with Bloom filter implementation) and PW-AC [67] to reach convergence and then terminate without any attempt at backtrack search.

5.4.2. Wipe-out by preprocessing

Semijoin reduction preprocessing is most effective under the conditions listed at the end of Section 4.2 for best performance of partition search. It is therefore appropriate to use semijoin reduction preprocessing when comparing partition search with reduction search. Moreover, reduction search experiments that will be reported in Section 5.6 would have been intolerably slow without preprocessing.

When constraints are tight and the constraint relations are random, semijoin reduction may wipe out the constraint relations entirely, so there is no backtrack search. When there is no search, a random problem is an unsatisfactory benchmark because the absence of a solution is established too easily [34]. Of course, when J is increased sufficiently, the probability of wipe-out goes to zero. This is illustrated in Fig. 23, which plots *empty domain ratio* versus J. This ratio is the number of trials in which there is wipe-out, divided by the total number of trials.

Table 7
Times in seconds in 500 trials with $N = 48$, $J = 10{,}000$ and $H = 0$

δ	n	q	PW-AC		ZM	
			av	sd	av	sd
5	6	32	**0.648**	0.002	1.189	0.002
5	6	48	**1.088**	0.011	1.916	0.004
10	6	32	10.12	1.535	**1.357**	0.009
10	6	48	24.10	1.900	**2.411**	0.017
10	7	32	91.68	5.335	**3.177**	0.048

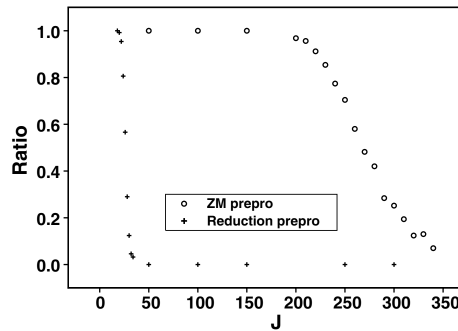

Fig. 23. Results of 500 trials with $N = 24$, $n = 6$, $\delta = 4$, $q = 72$ and $H = 0$. Empty domain ratio versus J.

Instead of ZM preprocessing, we can preprocess by using Cn simple tabular reduction with a minor modification that puts all j, $1 \leqslant j \leqslant q$, in the queue initially. As would be expected from [71, Theorem 4], and as can be seen in Fig. 23, this is very much less effective than ZM preprocessing. Cn-reduction preprocessing is not used henceforward.

To avoid wipe-out, we work with $H > 0$. Parameter H, which was introduced in Section 5.1, is the minimum number of solutions that the set Z must contain. When $H = 1$, preprocessing can yield a single solution without any search. To ensure that there is search, we work with $H > 1$. In this case our experiments are designed to find all solutions in Z, for the reason given in Section 5.1.

Fig. 24a provides evidence that increasing the ratio $H/(H + J)$ makes constraint satisfaction harder. In Fig. 24a, $K = H + J$ is held constant at $K = 6000$ while H increases from $H = 1$ to $H = 6000$. The average times to enumerate Z using partition and simple tabular reduction search are shown on the left and right vertical axes, respectively. As H increases, reduction search slows down very much more rapidly than partition search. When $H = 1$, 600 and 6000, partition search is faster than SN-1 reduction search by factors of 36.6, 88.9 and 390, respectively. One reason is that ZM preprocessing becomes less effective when H increases, as can be seen in Fig. 24b. When $H = K$ the constraint relations are minimal [60] (or totally consistent [53]) and therefore semijoin reduction has no effect.

The fact that $H > 0$ favours partition search is unhelpful to a fair comparison between partition and reduction search. In subsequent experiments, the use of $H/K = 0.1$ avoids wipe-out but may approximately double the ratio of reduction search time to partition search time. In the comparison of timings of partition and reduction search, we are primarily interested in factors of at least 10. Nevertheless, the effect of $H/K = 0.1$ should be taken into account.

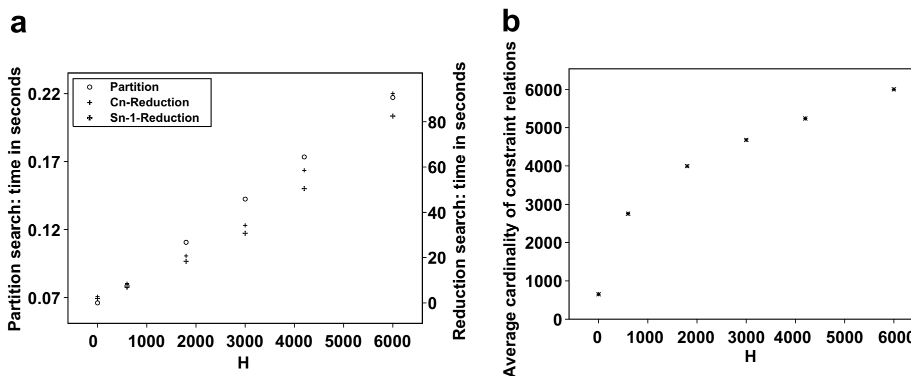

Fig. 24. Results of 500 trials with $K = 6000$, $\delta = 10$, $N = 48$, $n = 10$, $q = 15$ and $\mu = 3$: (a) average time to enumerate Z; (b) average cardinality of constraint relations after ZM preprocessing.

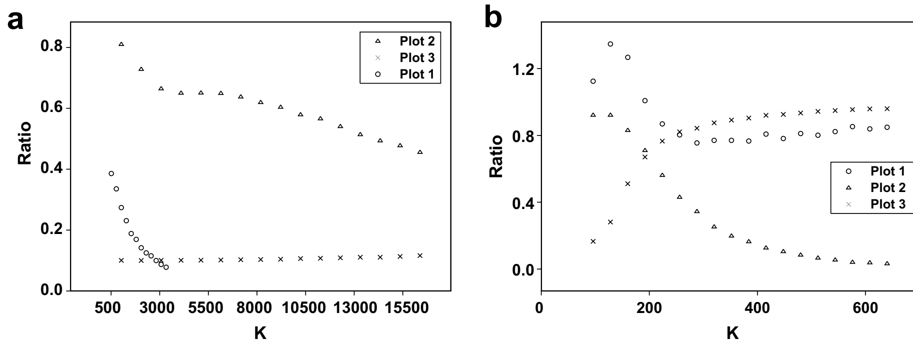

Fig. 25. Reduction search preprocessing experiments: results of 500 trials with $N = 48$ and $H/K = 0.1$: (a) ratios versus K with $q = 20$, $n = 12$ and $\delta = 10$; (b) ratios versus K with $q = 48$, $n = 6$ and $\delta = 6$.

5.4.3. Effects of preprocessing

Fig. 25 shows results of experiments using ZM preprocessing before Cn simple tabular reduction search. The three plots are:

Plot 1: Time with preprocessing divided by time without preprocessing.
Plot 2: Time taken by preprocessing divided by total time to enumerate Z.
Plot 3: Average $|R_j|$ after preprocessing divided by average $|R_j|$ before preprocessing.

When $K = H + J$ is small in Fig. 25a, the time taken by ZM preprocessing is such a large proportion of the total time to enumerate Z that ZM preprocessing is only slightly beneficial. The downward slope of Plot 1 in Fig. 25a indicates that when K is large, ZM preprocessing greatly reduces the total time taken to enumerate Z. For $K = 3328$ the times with and without preprocessing are 0.413 and 5.288 s, respectively. In this case, five hundred trials without preprocessing take about 44 min; this is why Plot 1 is truncated at $K = 3{,}238$. Plots 2 and 3 continue to higher K because they do not require trials omitting preprocessing.

The conditions listed at the end of Section 4.2 are not so well satisfied in Fig. 25b. When $K = 128$ in Fig. 25b, the times to enumerate Z with and without preprocessing are 0.063 and 0.047 s, respectively; the ZM preprocessing time is 0.058 s. In this case it is better to omit preprocessing.

ZM preprocessing can be applied before partition search. Table 8 shows examples of experimental timings with and without ZM preprocessing; we have not found any case where this preprocessing is cost-effective. Henceforward, partition search is always used without any preprocessing.

5.5. Backjumping

Dechter [24] provides an introduction to backjumping. Combination of reduction search with backjumping is certainly valuable in the determination of propositional satisfiability [49,51]. However, within the range of constraint satisfaction problems considered in the present paper, we have found experimentally that conflict-directed backjumping does not improve performance. This is because the backjump distance (i.e. the number

Table 8
Time to enumerate Z using partition search

N	Without prepro	With prepro	
	Time	Total time	Prepro time
32	0.104	1.515	1.447
44	0.427	1.217	0.825
48	1.253	1.905	0.724

Times are averages over 500 trials with $H/K = 0.1$, $K = 8192$, $q = 16$, $n = 8$, $\delta = 10$ and $\mu = 3$.

of variables that are jumped back over) turns out to be too small. We have also found experimentally that combination of backjumping with partition search is unhelpful for the same reason. We will not report these experiments in detail; backjumping is not used henceforward.

5.6. Time growth curves

Fig. 26a shows plots of time versus K for partition search with $\mu = 2, 3, 4, 5$. We would expect partition search to be faster the smaller the number of tuples in each hash bucket. For example, when $\mu = 4$, $\delta = 10$ and $K = 10{,}000$, each bucket contains one tuple on average. For $K > 11{,}264$ in Fig. 26a a faster time is obtained with $\mu = 4$ than with $\mu = 3$. If K were increased far beyond 32,768 we would expect a point to be reached where a faster time would be obtained with $\mu = 5$ than with $\mu = 4$.

In Fig. 26b the set-up time is a remarkably large part of the total time to enumerate Z. The set-up time is greater with $\mu = 4$ than with $\mu = 3$; calculation of $hash(t[S_j^{\mu}])$ processes m_j elements of t. This is why in Fig. 26a the total time with $\mu = 4$ is not less than with $\mu = 3$ when, for example, $K = 8192$.

Plots 5 and 6 in Fig. 26a are for Cn and S$n - 1$ simple tabular reduction search, respectively, with ZM preprocessing. Reduction search slows down more rapidly than partition search as K increases. In Fig. 26a and for $n > 8$ in Table 9, Cn-reduction search is faster than S$n - 1$-reduction but the difference between these is less than the difference between the speeds of reduction and partition search. When $n = 16$ in Table 9, ZM preprocessing perfectly minimizes the constraint relations.

When $n = 12$ in Table 9, partition search is faster than Cn dual search by a factor of 562. When $K = 3000$ in Fig. 27a, partition search is faster than Cn dual search by a factor of 1,184. When $K = 3000$, partition search is faster than Cn simple tabular reduction (STR) by a factor of 810 in Fig. 27a and by a factor of 52 in Fig. 27b.

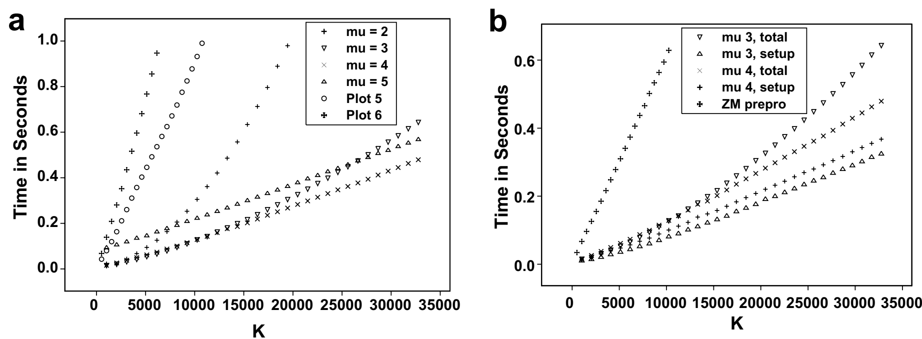

Fig. 26. Average time to enumerate Z, with $H/K = 0.1$, $N = 48$, $n = 16$, $q = 12$, $\delta = 10$ and 500 trials: (a) time versus K with $\mu = 2, 3, 4, 5$; (b) total time and set-up time versus K for $\mu = 3$ and $\mu = 4$. Also ZM preprocessing time versus K.

Table 9
Time in seconds to enumerate Z in 500 trials with $\delta = 10$, $N = 48$, $n = 6 \cdots 16$, $K = 2048$, $H/K = 0.1$ and $q = 192/n$ rounded down to the nearest integer, so $\rho = qn/N$ is approximately four

n	6	7	8	9	12	16
Partition set-up time	0.031	0.025	0.022	0.024	0.007	0.014
Partition total time	**0.147**	**0.042**	**0.031**	**0.032**	**0.010**	**0.020**
Sn − 1-reduction total time	38.63	5.427	1.701	0.624	0.271	0.283
Cn-reduction total time	62.66	9.702	1.912	0.596	0.168	0.167
Cn-dual total time	195.9	55.68	10.70	3.696	3.937	6.004
Reduction preprocess time	0.377	0.481	0.576	0.377	0.124	0.128
Reduction av $\|R_j\|$	1941	1723	1125	446.7	205.3	205.0

The bottom row shows the average cardinality of the constraint relations after ZM preprocessing which was used with reduction and dual search.

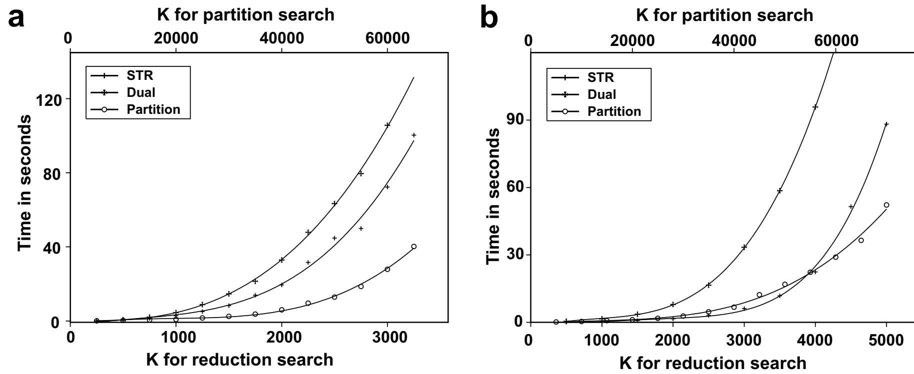

Fig. 27. Time to enumerate Z versus K, for partition search, Cn simple tabular reduction search (STR) and Cn dual search, with $N = 100$, $n = 10$, $\delta = 10$, $q = 50$ and 500 trials. The upper horizontal axis is for partition search; the lower horizontal axis is for STR and dual search: (a) $H = 0$ with no preprocessing; (b) $H/K = 0.1$ with ZM preprocessing for STR and dual search.

The smaller factor in Fig. 27b exemplifies our reason for working with ZM preprocessing and $H/K = 0.1$ in Fig. 28 and elswhere.

Fig. 28a–d plot time to enumerate Z versus N, δ and q for partition search and for Cn simple tabular reduction search with ZM preprocessing. In Fig. 28c and d the left hand vertical axis is for partition search and the right hand vertical axis is for reduction search, which is very much slower. If the time-growth curves in Fig. 28a and b were simply of the form $y = e^x$ then there would not be much chance of achieving practical results before being overwhelmed by very rapid growth of time. In fact there is a region of each curve where growth is slow enough for practical work to be possible. Time growth in Fig. 26a is near-linear, but would eventually become obviously exponential if continued to higher K.

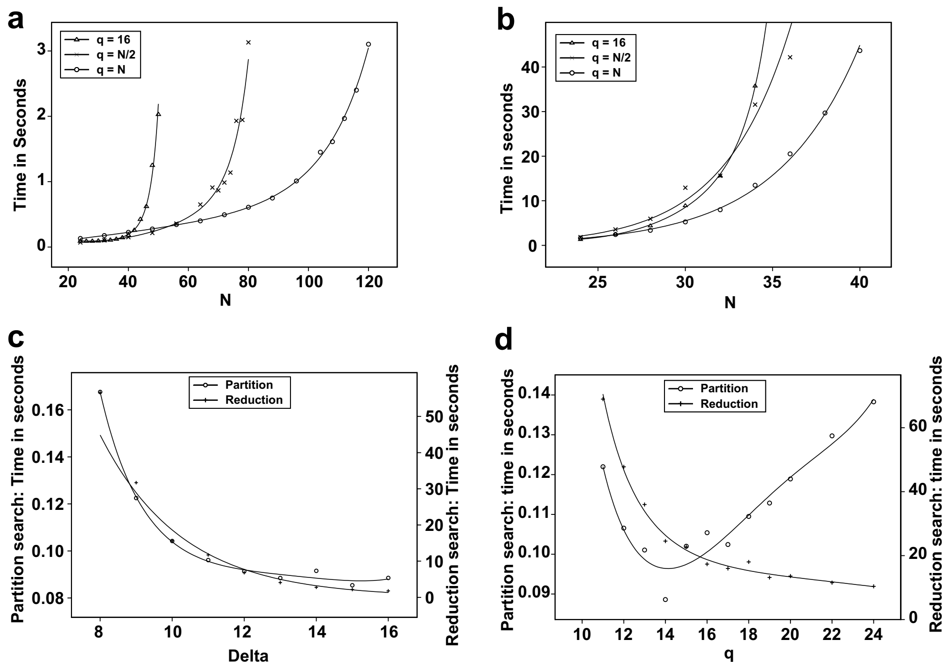

Fig. 28. Experiments with $K = 8192$, $H/K = 0.1$, $n = 8$, $\mu = 3$ and 500 trials: (a) time versus N using partition search with $\delta = 10$; (b) time versus N using reduction search with $\delta = 10$; (c) time versus δ with $N = 32$ and $q = 16$; (d) time versus q with $N = 32$ and $\delta = 10$.

Table 10
Summary of timings on 37 crossword puzzles with and without ZM preprocessing

	STR		Dual		Partition
	+prepro	−prepro	+prepro	−prepro	
Did not terminate in 2 h	21	17	22	22	31
30 min < time < 2 h	0	1	1	0	1
Time < 700 s	16	19	14	15	5
Average of times < 700 s	44.3	63.1	42.1	114.2	74.9

The first three rows show counts. The column headed STR shows results for C*n* simple tabular reduction search.

5.7. Experiments with crossword puzzles

Crossword puzzles have been used in [8,11,35,71] as part of an assessment of various constraint satisfaction algorithms. In this section we report experiments with benchmark crossword puzzles and a dictionary of 24,974 words used in [67].

The time taken to solve a crossword puzzle may depend strongly on which variable is instantiated first. For example, we ran simple tabular reduction search on puzzle 15.01 with two different choices of the first variable. With one of these a solution was found in 1.71 s. With the other choice, the algorithm ran for 2 h without finding a solution.

As another example of this phenomenon, there were 16 equally plausible choices of the initial instantiation in partition search on puzzle 19.01. With six of these choices, partition search ran for 2 h without finding a solution. With another two choices, partition search found a solution in 6.32 and 4.23 s, respectively. With each of the remaining eight choices, partition search found a solution in less than 2 s.

These and similar examples suggest that algorithms should be assessed by comparing their performance aggregated over a number of puzzles. Table 10 summarizes experimental results with 37 puzzles of size 15×15 or more, excluding puzzles 19.05, 19.10 and 23.01 because these demand words of a length such that no word in the given dictionary has this length. These results illustrate the tendency either to find a solution (or find that there is none) in less than 12 min or to run for 2 h without terminating.

Table 10 indicates that partition search is not suitable for crossword puzzles. This result is attributable to the incidence of cases where the cardinality of S_j^a is one. For example, over the ten 21×21 benchmark puzzles, the average number of values of j such that $|S_j^a| = 1$ was 9.9 (and the sd of this mean was 1.06). For all crossword puzzles, $|S_2^a| = |S_3^a| = 1$, which is particularly unhelpful to partition search.

6. Conclusion

A positive conclusion is that with tight constraints it is easy to find cases where semijoin reduction preprocessing, using Bloom filter implementation, makes reduction search substantially faster. A negative observation is that we have not found any case where GAC2001 is faster than simple tabular reduction search.

Our main conclusion is that, with tight constraints, partition search may be very much faster than reduction search, depending on the problem parameters and on the constraint structure. For example, in Fig. 28a and b, when $N = q = 40$, partition search is faster than simple tabular reduction search by a factor of 193.3. This may be of interest because partition search is definitely not a reduction search process; partition search does not delete values nor tuples during backtrack search.

Acknowledgements

Thanks are due to Dr. K. Stergiou for providing crossword puzzles and a dictionary. Thanks are also due to the referees for their comments.

Appendix

Theorem 1. *The set Z is not changed as a result of deletion from any domain D_i of any value v such that $v \notin \pi_{V_i}(R_j \cap (D_{j_1} \times D_{j_2} \times \cdots \times D_{j_n}))$ for some $S_j \in \Sigma_i$.*

Proof. By contradiction. Consider any N-tuple z that was in Z but is no longer in Z because a value $v \notin \pi_{V_i}(R_j \cap (D_{j_1} \times \cdots \times D_{j_n}))$ has been deleted from a domain D_i. There must be some $j \in \Sigma_i$ such that the tuple $t = z[S_j]$ was in $R_j \cap (D_{j_1} \times \cdots \times D_{j_n})$ before v was deleted from D_i but t was not in $R_j \cap (D_{j_1} \times \cdots \times D_{j_n})$ after this deletion because $t[V_i] = v$. But before deletion we had $t \in R_j \cap (D_{j_1} \times \cdots \times D_{j_n})$ and $t[V_i] = v$ whence $v \in \pi_{V_i}(R_j \cap (D_{j_1} \times \cdots \times D_{j_n}))$. □

Theorem 2. *The set Z is not changed as a result of deletion, from any constraint relation R_h, of any tuple t such that $t \notin \pi_{S_h}(R_h \bowtie R_j)$ for some R_j such that $S_h \cap S_j \neq \emptyset$.*

Proof. By contradiction. Consider any N-tuple z that was in Z but is no longer in Z because a tuple t such that $t \notin \pi_{S_h}(R_h \bowtie R_j)$, for some j, has been deleted from R_h. This deletion has removed z from Z because $z[S_h] = t$ and t is not now in R_h. Before deletion, $z \in R_1 \bowtie \cdots \bowtie R_q$ and therefore $z[S_h] \in \pi_{S_h}(R_1 \bowtie \cdots \bowtie R_q)$. But $\pi_{S_h}(R_1 \bowtie \cdots \bowtie R_q) \subseteq \pi_{S_h}(R_h \bowtie R_j)$, whence $t \in \pi_{S_h}(R_h \bowtie R_j)$. □

Theorem 3. *Partition search enumerates $Z = \{z | (\forall_{1 \leqslant j \leqslant q} j)(z[S_j] \in R_j)\}$.*

Proof. We assume that $\pi_{V_i}(R_j) \subseteq D_i$ for all j and for all $V_i \in S_j$; if this is not true initially it can be made true by deleting tuples from constraint relations before commencement of search. Let $Y'_{e(j)}$ be the set of all instantiations of $W_{e(j)}$ by partition search. Moreover, let $Y_{e(j)}$ be the set of all instantiations of $W_{e(j)}$ such that *consistent*(j) returns **true**. The set Θ_1 is empty and therefore $Y_{e(1)} = R_1$.

We now assume inductively for $1 < j \leqslant q_d$ that $Y_{e(j)} = Y_{e(j-1)} \bowtie R_j \bowtie_{\forall S_x \in \Theta_j} R_x$. For all $t \in R_{j+1}$ such that $t[S^a_{j+1}] = w_{e(j+1)}[S^a_{j+1}]$, partition search extends $w_{e(j)}$ to $w_{e(j+1)} = w_{e(j)} \cup t$. Therefore

$$Y'_{e(j+1)} = \{w_{e(j+1)} | w_{e(j+1)}[W_{e(j)}] \in Y_j \wedge w_{e(j+1)}[S_{j+1}] \in R_{j+1}\} = Y_j \bowtie R_{j+1}$$

The call *consistent*($j+1$) returns **true** iff $w_{e(j+1)}[S_x] \in R_x$ for all S_x in Θ_{j+1}. Therefore

$$Y_{e(j+1)} = \{w_{e(j+1)} | w_{e(j+1)} \in Y'_{e(j+1)} \wedge (\forall x_{S_x \in \Theta_{j+1}})(w_{e(j+1)}[S_x] \in R_x)\} = Y_{e(j)} \bowtie R_{j+1} \bowtie_{\forall S_x \in \Theta_{j+1}} R_x.$$

After partition resequencing, every scope that is not in Σ_d is in Θ_j for some j. Therefore by induction

$$Y_{e(q_d)} = R_1 \bowtie R_2 \bowtie \cdots \bowtie R_q = Z. \quad \square$$

References

[1] D. Achlioptas, M.S.O. Molloy, L.M. Kirousis, Y.C. Stamatiou, E. Kranakis, D. Krizanc, Random constraint satisfaction: a more accurate picture, Constraints 6 (4) (2001) 329–344.

[2] M. Al-Suwaiyel, E. Horowitz, Algorithms for trie compaction, ACM Trans. Database Syst. 9 (1984) 243–263.

[3] J. Aoe, K. Morimoto, M. Shishibori, K.-H. Park, A trie compaction algorithm for a large set of keys, IEEE Trans. Data Knowledge Eng. 8 (1996) 476–491.

[4] P.J. Artymiuk, H.M. Grindley, A.R. Poirrette, D.W. Rice, E.C. Ujah, P. Willett, Identification of β-sheet motifs, of ψ-loops and of patterns of amino-acid residues in three-dimensional protein structures using a subgraph-isomorphism algorithm, J. Chem. Inform. Comput. Sci. 34 (1994) 54–62.

[5] E. Babb, Implementing a relational database by means of specialized hardware, ACM Trans. Database Syst. 4 (1979) 1–29.

[6] B. Babcock, S. Chaudhuri, Towards a robust query optimizer: a principled and practical approach, in: Proceedings ACM SIGMOD, Baltimore, MD, 2005, pp. 119–130.

[7] F. Bacchus, P. van Run, Dynamic variable ordering in CSPs, Lecture Notes in Computer Science, vol. 976, Springer, New York, 1995, pp. 258–275.

[8] F. Bacchus, P. van Beek, On the conversion between non-binary and binary constraint satisfaction problems, in: Proceedings of AAAI'98, Madison, WI, 1998, pp. 310–318.

[9] F. Bacchus, X. Chen, P. van Beek, T. Walsh, Binary vs. non-binary constraints, Artif. Intell. 140 (2002) 1–37.

[10] S. Bandyopadhyay, Q. Fu, A. Sengupta, A cyclic multi-relation semijoin operation for query optimization in distributed databases, in: Proceedings of IEEE International Conference on Performance, Computing, and Communications (IPCCC'00), 2000, pp. 101–107.
[11] A. Beacham, X. Chen, J. Sillito, P. van Beek, Constraint programming lessons learned from crossword puzzles, Lecture Notes in Computer Science, vol. 2056, Springer, New York, 2001, pp. 78–87.
[12] P.A. Bernstein, D-M.W. Chiu, Using semi-joins to solve relational queries, JACM 28 (1) (1981) 25–40.
[13] C. Bessière, J.C. Régin, MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems, in: Proceedings of CP'96, Cambridge, MA, 1996, pp. 61–75.
[14] C. Bessière, J.C. Régin, Refining the basic arc consistency algorithm, in: Proceedings of IJCAI-01, Seattle, WA, 2001, pp. 309–315.
[15] C. Bessière, P. Meseguer, E.C. Freuder, J. Larrosa, On forward checking for non-binary constraint satisfaction, Artif. Intell. 141 (2002) 205–224.
[16] I.J. Bruno, N.M. Kemp, P.J. Artymiuk, P. Willett, Representation and searching of carbohydrate structures using graph-theoretic techniques, Carbohyd. Res. 304 (1997) 61–67.
[17] R.E. Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Trans. Comput. C-35 (1986) 677–691.
[18] R.E. Bryant, Symbolic Boolean manipulation with ordered binary-decision diagrams, ACM Comput. Surv. 24 (1992) 293–318.
[19] M-S. Chen, P.S. Yu, K-L. Wu, Optimization of parallel execution for multi-join queries, IEEE Trans. Knowledge Data Eng. 8 (1996) 416–428.
[20] D. Cohen, P. Jeavons, R. Gault, New tractable classes from old, Constraints 8 (2003) 263–282.
[21] D. Comer, Analysis of a heuristic for full trie minimization, ACM Trans. Database Syst. 6 (1981) 513–537.
[22] M.C. Cooper, Estimating optimal parameters for parallel database hardware, Int. J. Syst. Sci. 23 (1992) 119–125.
[23] R. Debruyne, C. Bessière, Domain filtering consistencies, J. Artif. Intell. Res. 14 (2001) 205–230.
[24] R. Dechter, Constraint Processing, Morgan Kaufman, San Fransisco, 2003.
[25] A. Dechter, R. Dechter, Removing redundancies in constraint networks, in: Proceedings of the AAAI-87, Seattle, WA, 1987, pp. 105–109.
[26] R. Dechter, J. Pearl, Network-based heuristics for constraint-satisfaction problems, Artif. Intell. 34 (1988) 1–38.
[27] R. Dechter, J. Pearl, Tree clustering for constraint networks, Artif. Intell. 38 (1989) 253–266.
[28] R. Dechter, I. Rish, Mini-buckets: a general scheme for bounded inference, JACM 50 (2003) 107–153.
[29] R. Dreschsler, D. Sieling, Binary decision diagrams in theory and practice, Int. J. Software Tools Technol. Transf. 3 (2001) 112–136.
[30] D. Frost, R. Dechter, Look-ahead value ordering for constraint satisfaction problems, in: Proceedings of IJCAI'95, Montreal, Canada, 1995, pp. 572–578.
[31] D. Frost, R. Dechter, Maintenance scheduling problems as benchmarks for constraint algorithms, Ann. Math. Artif. Intell. 26 (1999) 149–170.
[32] A. Galindo-Legaria, Outerjoins as disjunctions, in: Proceedings of the 1994 ACM SIGMOD Conference, Minneapolis, MN, 1994, pp. 348–358.
[33] H. Garcia-Molina, J.D. Ullman, J. Widom, Database Systems: The Complete Book, Prentice Hall, Upper Saddle River, NJ, 2002.
[34] I.P. Gent, E. Macintyre, P. Prosser, B.M. Smith, T. Walsh, Random constraint satisfaction: flaws and structure, Constraints 6 (2001) 345–372.
[35] M. Ginsberg, M. Frank, M. Halpin, M. Torrance, Search lessons learned from crossword puzzles, in: Proceedings of AAAI-90, Boston, MA, 1990, pp. 210–215.
[36] S.W. Golomb, L.D. Baumert, Backtrack programming, JACM 12 (4) (1965) 516–524.
[37] R.D. Gopal, R. Ramesh, S. Zionts, Criss-cross hash joins: design and analysis, IEEE Trans. Knowledge Data Eng. 13 (4) (2001) 637–653.
[38] M. Gyssens, P.G. Jeavons, D.A. Cohen, Decomposing constraint satisfaction problems using database techniques, Artif. Intell. 66 (1994) 57–89.
[39] T. Hadzic, S. Subbarayan, R.M. Jensen, H.R. Andersen, J. Møller, H. Hulgaard, Fast backtrack-free product configuration using a precompiled solution space representation, in: Proceedings of International Conference on Economic, Technical and Organisational Aspects of Product Configuration Systems, Copenhagen, 2004, pp. 131–138.
[40] R.M. Haralick, L.S. Davis, A. Rosenfeld, D.L. Milgram, Reduction operations for constraint satisfaction, Inform. Sci. 14 (1978) 199–219.
[41] R.M. Haralick, G.L. Elliott, Increasing tree search efficiency for constraint satisfaction problems, Artif. Intell. 14 (1980) 263–313.
[42] H-I. Hsiao, C-S. Chen, P.S. Yu, Parallel execution of hash joins in parallel databases, IEEE Trans. Parallel Distrib. Syst. 8 (1997) 872–883.
[43] I.F. Ilyas, V. Markl, P. Haas, P. Brown, A. Aboulnaga, CORDS: automatic discovery of correlations and soft functional dependencies, in: Proceedings of ACM SIGMOD, Paris, France, 2004, pp. 647–658.
[44] P. Janssen, P. Jégou, R. Nouguier, M.C. Vilarem, A filtering process for general constraint satisfaction problems, in: Proceedings of the IEEE International Workshop on Tools for Artificial Intelligence, Fairfax, VA, 1989, pp. 420–427.
[45] D. Kossmann, State of the art in distributed query processing, ACM Comp. Surv. 32 (2000) 422–469.
[46] D. Kossmann, K. Stocker, Iterative dynamic programming: a new class of query optimization algorithms, ACM Trans. Database Syst. 25 (2001) 43–82.

[47] J. Larrosa, R. Dechter, Boosting search with variable elimination in constraint optimization and constraint satisfaction problems, Constraints 8 (2003) 303–326.
[48] J. Larrosa, E. Morancho, D. Niso, On the practical use of variable elimination in constraint optimization problems: 'Still-life' as a case study, J. Artif. Intell. Res. 23 (2005) 421–440.
[49] C. Lecoutre, F. Boussemart, F. Hemery, Backjump-based techniques versus conflict directed heuristics, in: Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'04), 2004, pp. 549–557.
[50] Z. Li, K.A. Ross, PERF join: an alternative to two-way semijoin and bloomjoin, in: Proceedings of the Fourth International Conference on Information and Knowledge Management, Baltimore, MD, 1995, pp. 137–144.
[51] I. Lynce, J.P. Marques-Silva, An overview of backtrack search satisfiability algorithms, Ann. Math. Artif. Intell. 37 (3) (2003) 307–326.
[52] A.K. Mackworth, Consistency in networks of relations, Artif. Intell. 8 (1) (1977) 99–118.
[53] D. Maier, The Theory of Relational Databases, Computer Science Press, Rockville, MD, 1983.
[54] N. Mamoulis, K. Stergiou, Solving non-binary CSPs using the hidden variable encoding, in: T. Walsh (Ed.), Proceedings of CP 2001, Lecture Notes in Computer Science, vol. 2239, Springer, Berlin, 2001, pp. 168–182.
[55] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, M. Cilimdzic, Robust query processing through progressive optimization, in: Proceedings of ACM SIGMOD Conference, Paris, France, 2004, pp. 659–670.
[56] J.T. McCall, J.G. Tront, F.G. Gray, R.M. Haralick, W.M. McCormack, Parallel computer architectures and problem solving strategies for the consistent labelling problem, IEEE Trans. Comput. C-34 (1985) 973–980.
[57] J.J. McGregor, Relational consistency algorithms and their application in finding subgraph and graph isomorphisms, Inform. Sci. 19 (1979) 229–250.
[58] P. Mishra, M.H. Eich, Join processing in relational databases, ACM Comput. Surv. 24 (1992) 63–113.
[59] M. Mitzenmacher, Compressed Bloom filters, IEEE/ACM Trans. Network. 10 (2002) 604–612.
[60] U. Montanari, Networks of constraints: Fundamental properties and applications to picture processing, Inform. Sci. 7 (1974) 95–132.
[61] J.M. Morrissey, W.K. Osborn, Y. Liang, Collisions and reduction filters in distributed query processing, in: Proceedings of 2000 Canadian Conference on Electrical and Computer Engineering, Halifax, Nova Scotia, vol. 1, 2000, pp. 240–244.
[62] J.K. Mullin, Optimal semijoins for distributed database systems, IEEE Trans. Software Eng. 16 (1990) 558–560.
[63] S. Nagarajan, S.D. Goodwin, A. Sattar, Extending dual arc consistency, Int. J. Pattern Recogn. Artif. Intell. 17 (5) (2003) 781–815.
[64] N. Narodytska, T. Walsh, Constraint and variable ordering heuristics for compiling configuration problems, in: Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, 2007, pp. 149–154.
[65] P. Prosser, An empirical study of phase transitions in binary constraint satisfaction problems, Artif. Intell. 81 (1996) 81–109.
[66] D. Sabin, E.C. Freuder, Contradicting conventional wisdom in constraint satisfaction, in: A. Borning (Ed.), Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, PPCP'94, Rosario, Orcas Island, Washington, published as Lecture Notes in Computer Science, vol. 874, 1994, pp. 10–20.
[67] N. Samaras, K. Stergiou, Binary encodings of non-binary constraint satisfaction problems: algorithms and experimental results, J. Artif. Intell. Res. 24 (2005) 641–684.
[68] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, T.G. Price, Access path selection in a relational database management system, in: Proceedings of ACM 1979 SIGMOD Conference, Boston, MA, 1979, pp. 23–34.
[69] L.D. Shapiro, Join processing in database systems with large main memories, ACM Trans. Database Syst. 11 (1986) 239–264.
[70] B.M. Smith, M.E. Dyer, Locating the phase transition in binary constraint satisfaction problems, Artif. Intell. 81 (1996) 155–181.
[71] K. Stergiou, T. Walsh, Encodings of non-binary constraint satisfaction problems, in: Proceedings of AAAI'99, 1999, pp. 163–168.
[72] S. Subbarayan, R.M. Jensen, T. Hadzic, H.R. Andersen, J. Møller, H. Hulgaard, Comparing two implementations of a complete and backtrack-free interactive configurator, in: Proceedings of the CP-04 Workshop on CSP Techniques with Immediate Application, Toronto, Canada, 2004, pp. 97–111.
[73] R.E. Tarjan, M. Yannakakis, Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs, SIAM J. Comput. 13 (1984) 566–579.
[74] J.D. Ullman, Principles of Database and Knowledge-Base systems, vol. 1, Computer Science Press, Rockville, MD, 1988.
[75] J.R. Ullmann, Associating parts of patterns, Inform. Control 9 (6) (1966) 583–601.
[76] J.R. Ullmann, An algorithm for subgraph isomorphism, JACM 23 (1) (1976) 31–42.
[77] J.R. Ullmann, A binary n-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words, Comput. J. 20 (2) (1977) 141–147.
[78] J.R. Ullmann, A relational view of text image processing, in: R.M. Haralick, J.C. Simon (Eds.), Issues in Digital Image Processing, Sijthoff and Noordhoff, Alphen aan den Rijn, The Netherlands, 1980, pp. 139–169.
[79] J.R. Ullmann, Discrete optimization by relational constraint satisfaction, IEEE Trans. Pattern Anal. Mach. Intell. PAMI-4 (5) (1982) 544–550.
[80] J.R. Ullmann, Fast implementation of relational operations via inverse projection, Comput. J. 31 (2) (1988) 147–154.
[81] J.R. Ullmann, R.M. Haralick, L.G. Shapiro, Computer architecture for solving consistent labeling problems, Comput. J. 28 (2) (1985) 105–111.
[82] P. Van Hentenryck, Y. Deville, C-M. Tang, A generic arc-consistency algorithm and its specializations, Artif. Intell. 57 (1992) 291–321.

[83] M.Y. Vardi, Constraint satisfaction and database theory: a tutorial, in: Proceedings of the 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, Dallas, TX, 2000, pp. 76–85.
[84] I. Wegener, Branching programs and binary decision diagrams, SIAM Monographs on Discrete Mathematics and Applications, Philadelphia, PA, 2000.
[85] K. Xu, W. Li, An average analysis of backtracking on random constraint satisfaction problems, Ann. Math. Artif. Intell. 33 (2001) 21–37.
[86] Y. Zhang, A.K. Mackworth, Parallel and distributed algorithms for finite constraint satisfaction problems, in: Proceedings of the third IEEE Symposium on parallel distributed computing, Dallas, TX, 1991, pp. 394–397.
[87] Y. Zhang, R.Yap, Making AC-3 an optimal algorithm, in: Proceedings of 17th International Joint Conference on Artificial Intelligence, 2001, pp. 316–321.