

# Bit-vector Algorithms for Binary Constraint Satisfaction and Subgraph Isomorphism

JULIAN R. ULLMANN  
King's College London, UK

---

A copy-edited and re-paginated version of this article is published in the *ACM Journal of Experimental Algorithmics*, Vol 15, No 1, Article 1.6 (January 2011), 64 pages.

---

A solution to a binary constraint satisfaction problem is a set of discrete values, one in each of a given set of domains, subject to constraints that allow only prescribed pairs of values in specified pairs of domains. Solutions are sought by backtrack search interleaved with a process that removes from domains those values that are currently inconsistent with provisional choices already made in the course of search. For each value in a given domain, a bit-vector shows which values in another domain are or are not permitted in a solution. Bit-vector representation of constraints allows bit-parallel, therefore fast, operations for editing domains during search. This article revises and updates bit-vector algorithms published in the 1970's, and introduces focus search, which is a new bit-vector algorithm relying more on search and less on domain-editing than previous algorithms. Focus search is competitive within a limited family of constraint satisfaction problems.

Determination of subgraph isomorphism is a specialized binary constraint satisfaction problem for which bit-vector algorithms have been widely used since the 1980's, particularly for matching molecular structures. This article very substantially updates the author's 1976 subgraph isomorphism algorithm, and reports experimental results with random and real-life data.

Categories and Subject Descriptors: G.2.1 [**Discrete Mathematics**]: Combinatorics—*Combinatorial algorithms*; G.2.1 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: alldifferent constraint, backtrack, binary constraints, bit-vector, constraint satisfaction, domain reduction, constraint propagation, focus search, forward checking, graph indexing, molecule matching, prematching, signature file, subgraph isomorphism

---

## 1. INTRODUCTION

Determination of subgraph isomorphism, that is, finding whether a given graph is a subgraph of a further given graph, is an example of a binary constraint satisfaction problem [Mackworth 1977], in which we seek one or more combinations of discrete values of a given set of  $V = \{V_1, \dots, V_i, \dots, V_n\}$  of  $n$  variables so as to satisfy:

*Unary constraints.* Values of a variable  $V_i$  must belong to a given finite domain,  $D_i$ , of discrete values.

*Binary constraints.* Values of variables  $V_i$  are constrained to satisfy a given set  $C$  of binary predicates  $P_{ij}$ , such that if  $V_i$  takes on the value  $u$ , then  $V_j$  can take on the value  $v$  if and only if  $P_{ij}(u, v)$ .

A solution,  $z = (z_1, \dots, z_i, \dots, z_n)$ , is an  $n$ -tuple such that

---

Author's address: Department of Informatics, King's College London, Strand, London WC2R 2LS, UK; email: jrullmann@acm.org.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 2011 ACM 0000-0000/2011/0000-0001 \$5.00

— $z_i \in D_i$  for all  $1 \leq i \leq n$ , and

—If there is a constraint between variables  $V_i$  and  $V_j$  then  $P_{ij}(z_i, z_j)$ .

We seek solutions by using enhanced backtrack search [Golomb and Baumert 1965], which instantiates successive variables. *Instantiate* means *assign a trial value*. After instantiating some variables, the search process checks whether constraints that should now be satisfied are actually satisfied. If they are satisfied, then a further variable is instantiated; otherwise the most recently assigned value is changed. When all values have been tried, the search backtracks to re-instantiate a previously-instantiated variable, and so on.

After a set,  $W$ , of variables have been instantiated, let  $Z_W$  be the set of solutions such that in each solution in  $Z_W$ , each variable in  $W$  has its instantiated value. If there is no solution in  $Z_W$  that includes a value  $v$  of a variable  $V_i$  then the set  $Z_W$  of solutions is unchanged if  $v$  is removed from domain  $D_i$ . *Domain reduction* means removal of values that cannot be included in a solution. Domain reduction is intended to reduce the number of combinations explored during backtrack search.

Although this constraint satisfaction problem is NP-Complete [Cook 1971], algorithms that interleave domain reduction with backtrack search have a substantial community of practical users, for example [Boutselakis et al. 2003; Artymiuk et al. 2005; Klukas et al. 2005; Durand et al. 2006]. Practical effectiveness inevitably depends upon implementational detail, which in turn depends upon the representation of constraints. To avoid re-evaluation of predicates, all evaluation can be done just once, prior to commencement of backtrack search, which now works with a stored representation of constraints. For example, we can store a tabular representation of  $P_{ij}$ , which is the set of all ordered pairs  $(u, v)$  of values such that  $P_{ij}(u, v)$  is **true** [Lecoutre 2009]. A different tabular representation of  $P_{ij}$  is the set of all ordered pairs  $(u, v)$  of values such that  $P_{ij}(u, v)$  is **false**.

This article is primarily concerned with representation of a predicate  $P_{ij}$  by a one-dimensional array  $M_j^i$  of sets such that  $M_j^i[u] = \{v | P_{ij}(u, v)\}$ . Thus  $P_{ij}(u, v) \equiv v \in M_j^i[u]$ . As is very well known, a set can be represented by a bit-vector that has one bit corresponding to each possible member. A bit is ‘1’ iff the corresponding member is present in the set. We represent sets by bit-vectors, on which an ordinary CPU can perform bit-parallel intersection and union operations, thus gaining speed.

Bit-vector representation was used in the 1970’s for seeking subgraph isomorphism [Ullmann 1976] and also for general binary constraint satisfaction [McGregor 1979]. During the next thirty years, bit-vector isomorphism algorithms were widely employed in practice, particularly in molecule matching applications [Willett et al. 1998]. During the same period the artificial intelligence community developed constraint satisfaction algorithms working directly with predicates, or instead with tabular representations thereof [Bessière 2006], but usually not with bit-vector representation. Perhaps this was because bit-vector representation provides no complexity-theoretic benefit. However, from the viewpoint of practical performance, authoritative experimentation [Lecoutre and Vion 2008] suggests strongly that, for binary constraints, bit-vector implementation has not been surpassed in the artificial intelligence literature.

In order to benefit from bit-parallelism, domain reduction procedures in Section 3 of this article represent domains, as well as binary constraints, by bit-vectors. When the search backtracks, domains must be restored to an appropriate previous state. Bit-vectors representing domains can be saved/restored by push/popping a stack, incurring considerable cost in time and memory. Section 4 introduces focus search which is a new algorithm that *avoids* save/restore. In Section 5 the use of surrogate values can decrease the length of bit-vectors, thus reducing memory requirements and increasing the speed of search.

Section 6 applies the technology of previous sections in the determination of subgraph isomorphism, which is a specialized constraint satisfaction problem wherein:

- No two values in a solution  $z$  may be the same.
- Specialized unary constraints may greatly reduce the work done by a search process.
- Bit-vector memory requirements are reduced when the same constraint predicate  $P_{ij}$  applies to more than one pair  $(i, j)$  of variables.

Subgraph isomorphism has well-established practical relevance in chemoinformatics [Leach and Gillet 2003; Brown 2009].

Before reporting experiments with isomorphism, Section 7 reports experiments with randomly generated constraint satisfaction problems and also with radio frequency assignment problems, emphasizing that isomorphism is not the only important practical application for binary constraint satisfaction algorithms.

Domain reduction in Sections 3 and 4 is based on the well-known idea of support, which is revised in Section 2.

## 2. SUPPORT

### 2.1 Full support

After instantiating a set  $W$  of variables, we need to identify values that cannot now belong to any solution. Identification is based on simple inference, for which we use the following well-established definitions. A *graph* is a set of vertices and a set of edges that are pairs of these vertices. The *degree* of a vertex is the number of edges that include it. Two vertices are said to be *adjacent* if they belong to the same edge.

Corresponding to a binary constraint satisfaction problem we define a *constraint graph* wherein vertices correspond to variables and edges correspond to pairs of variables that are subject to binary constraints. Two variables are adjacent iff they are the subject to the same binary constraint. We define  $A_i$  to be the set of variables that are adjacent to the variable  $V_i$ . Moreover,  $C_{ij}$  is the constraint between variable  $V_i$  and a variable  $V_j \in A_i$ .

If there is a constraint between  $V_i$  and  $V_j$ , then in any solution  $z$  we must have  $P_{ij}(z_i, z_j)$ . Therefore a value  $u$  in the domain  $D_i$  of  $V_i$  cannot belong to any solution unless there is a value  $v \in D_j$  such that  $P_{ij}(u, v)$ . A value  $v \in D_j$  such that  $P_{ij}(u, v)$  is said to *support*  $u \in V_i$  [Bessière 2006; Naanaa 2009]. A value  $u \in D_i$  that is supported by at least one value in domain  $D_j$  is said to be *supported in* constraint  $C_{ij}$ . A value  $u \in D_i$  that is not supported in constraint  $C_{ij}$  cannot belong to any solution. Going further, a value  $u \in D_i$  is said to be *fully supported* iff it is supported in every constraint  $C_{ij}$  such that  $V_j \in A_i$ . A value  $u \in D_i$  that is not fully supported cannot belong to any solution and can therefore be removed from domain  $D_i$ .

The set of solutions remains unchanged if values that are not fully supported are removed from domains. Removal of values from domains may remove support for values in other domains, so there may be propagation of removal from domains as in [Ullmann 1965]. At *convergence*, which is the situation where there is no further removal because every value in every domain is fully supported, the constraint structure is said to be *arc consistent* [Mackworth 1977; Bessière 2006].

After each instantiation we invoke a domain reduction procedure that attempts to simplify subsequent search by removing unsupported values from domains. Some versions of this procedure propagate removal until convergence<sup>1</sup>, others do not.

<sup>1</sup>*Maintaining Arc Consistency*, abbreviated to MAC, is a widely-used name for a family of constraint satisfaction backtrack algorithms which, after each instantiation, propagate removal until arc consistency is achieved, as in [Ullmann 1976; 1977; McGregor 1979; Sabin and Freuder 1994].

If domain reduction empties any domain when a set  $W$  of variables have been instantiated, then there is no solution that includes these instantiations. In this case nothing can be achieved by instantiating further variables; instead the backtrack search re-instantiates a variable in  $W$ .

When domain  $D_i$  is inferentially reduced to a single value, we have *implied* instantiation of the variable  $V_i$ , whereas instantiation by the search process is *elective* [Lynce and Marques-Silva 2003]. Suppose that immediately after elective or implied instantiation of  $V_i$ , domain reduction continues until convergence. Consider a constraint  $C_{ij}$  such that domain  $D_j$  is multivalued at convergence. At this time we can be sure that every value in  $D_j$  is supported by the single value of  $V_i$ . Equally surely, the single value of  $V_i$  is supported by every value in  $D_j$ . While further variables are instantiated, the single value of  $V_i$  remains supported in constraint  $C_{ij}$  so long as  $D_j$  is not empty. If any domain is empty the search does not proceed. Therefore we do not need to check whether the value of  $V_i$  is supported in constraint  $C_{ij}$  after further instantiation. This is true for every constraint  $C_{ik}$  such that  $V_k \in A_i$ , so if domain  $D_i$  is single-valued at the time of convergence, then nothing is achieved by checking whether the value in  $D_i$  is supported after further instantiations. Lecoutre [2008] has emphasized that it is unnecessary to check whether electively-instantiated values are supported, and this idea appears also at Line 5 in Figure 2 in Bessière et al [2008]. We now go further by observing that it is also unnecessary to check whether *implied* instantiated values are supported. This means that after an elective instantiation we only need to check whether values in multivalued domains are supported. Here *multivalued* means multivalued at the time of convergence in a previous invocation of a domain reduction procedure, since when there has been neither backtrack nor re-instantiation.

If, to improve efficiency, we apply domain reduction only to multivalued domains, then the first application requires attention. If domain  $D_i$  is single-valued initially, before the first application of domain reduction, then we cannot then be sure that its value is fully supported. We deal with this simply by removing all unsupported values from domains during preprocessing prior to commencement of search, as in Section 3.3.

## 2.2 Bit vector representation of support sets

We represent predicate  $P_{ij}$  by a one-dimensional array,  $M_j^i$ , of sets initialized so that  $M_j^i[u] = \{v | P_{ij}(u, v)\}$ . In this  $M_j^i$  notation, the superscript identifies the variable whose values select elements of the one-dimensional array; the subscript identifies the other variable involved in the constraint. Each set  $M_j^i[u]$  is represented by a bit-vector.

The bit-vector  $M_j^i[u]$  can be regarded as a row of a bit-matrix  $M_j^i$ . If the rows of this matrix are packed into words to allow bit-parallel operations, the columns are not. The row  $M_j^i[u]$  allows bit-parallel operations on the set of all values of  $V_j$  that can be supported in constraint  $C_{ij}$  by value  $u \in D_i$ . We also require bit-parallel operations on the set of all values of  $V_i$  that can be supported in constraint  $C_{ij}$  by value  $u \in D_j$ . For this purpose, we employ a second bit-matrix  $M_i^j$ , initialized so that  $M_i^j[u] = \{v | P_{ij}(v, u)\}$ , with rows selected by values of the superscript variable. For this second bit-matrix,  $P_{ij}(v, u) \equiv (v \in M_i^j[u])$ . Representing  $P_{ij}$  both by  $M_j^i$  and also by its transpose  $M_i^j$  obviously requires twice as much memory as just one of these representations. All bit-matrices have the same dimensions such that their rows are bit-vectors in which the number of bits is always the maximum cardinality,  $\delta$ , of any domain.

Constraints are *symmetric* if all predicates in a given binary constraint satisfaction problem are such that  $P_{ij}(u, v) = P_{ij}(v, u)$ ; otherwise, constraints are *asymmetric*. For symmetric constraints we have  $M_j^i[u] = \{v | P_{ij}(u, v)\}$  and  $M_i^j[u] = \{v | P_{ij}(v, u)\}$ ,

```

procedure search;
input:  Dsets which is an array  $[1..n]$  of bit-vectors such that  $Dsets[i]$  represents domain  $D_i$ 
        initialized according to unary constraints; also
        initialized bit-matrices (to be referenced by the domain reduction procedure);
output: Line 13 records all solutions.
begin
1  initialize domains, stacks, instantiation sequence and weights;
2  choose(i, terminal); (* Select variable  $V_i$  to be instantiated next and
    also return terminal = (all domains are single-valued). *)
3  initialize search for untried values in domain  $D_i$ ;
4  repeat
5    if there are any untried values in domain  $D_i$  then
6      push(i, Dsets); (* This saves a copy of i and also of Dsets *)
7      v := next untried value in domain  $D_i$ ;
8       $D_i := \{v\}$ ; (* Instantiating Variable  $V_i$  to value v *)
9      reduce(i, Dsets, consistent); (* Domain reduction procedure *)
10     if consistent then (* No domain is empty, so proceed *)
11       choose(i, terminal); (* Select variable  $V_i$  to be instantiated and
    also return terminal = (all domains are single-valued). *)
12     if terminal then (* All domains are single-valued *)
13       output this solution;
14       pop(i, Dsets) (* Restore before trying another value *)
15     else initialize search for untried values in newly chosen  $D_i$ ;
16       (* The next iteration will instantiate the newly chosen  $V_i$  *)
17     end if;
18   else pop(i, Dsets) (* Restore before trying another value of the same variable *)
19   end if;
20 else (* Backtrack to the previous variable *)
21   if stack is not empty then pop(i, Dsets) (* to restore these *) end if
22 end if;
23 until stack is empty;
end search;

```

Fig. 1. Outline of a backtrack search algorithm that finds all solutions to an instance of the constraint satisfaction problem.

so  $M_j^i = M_i^j$  and we only need one bit-matrix to represent each predicate. As will be explained in Section 6.1, subgraph isomorphism with unordered graphs is an example of a symmetric constraint satisfaction problem: Adjacency matrices of unordered graphs are symmetric.

### 3. BASIC BIT VECTOR DOMAIN REDUCTION ALGORITHMS

#### 3.1 A backtrack search algorithm

The backtrack algorithm<sup>2</sup> outlined in Figure 1 employs dynamic variable ordering [Lecoutre 2009, Section 9.1.1], which means that the instantiation sequence may vary during the search. Procedure *choose* uses the *dom/wdeg* heuristic [Boussemart et al. 2004] employing weights  $\{w_{ij}\}$  to select the next variable,  $V_i$ , for elective instantiation<sup>3</sup>. Procedure *choose* is shown in Appendix Section A.2.

<sup>2</sup>This algorithm generally tries many successive values for each variable. Binary branching, which is an alternative strategy, tries just one value before selecting another variable [Lecoutre 2009, Page 362].

<sup>3</sup>This heuristic associates a score with each variable that is a candidate for elective instantiation because its domain is not already single-valued [Lecoutre 2009, Section 9.3.1]. For the variable  $V_i$  this score is  $|D_i| / (\sum_{j \in \bar{A}_i} w_{ij})$ , where  $|D_i|$  is the current cardinality of domain  $D_i$ , and  $\bar{A}_i = \{V_k | (V_k \text{ is adjacent to } V_i) \wedge (|D_k| > 1)\}$  and  $w_{ij}$  is a weight associated with the unordered pair  $\{V_i, V_j\}$  as follows. Initialization assigns  $w_{ij} := 1$ . During the search, the weight  $w_{ij}$  is increased by one when no value in  $D_i$  is supported by  $D_j$  or when no value in  $D_j$  is supported by  $D_i$ . Procedure *choose* selects the variable for which the score is minimal.

To process values in a domain in turn serially, as at Line 7 in Figure 1, we can search for successive 1's in the domain's bit-vector. Instead, to save time, we can avoid visiting 0's in bit-vectors by representing each domain by an array, as well as by a bit-vector, so we have duplicate representation. During the search, the duplicate representation must be saved and restored<sup>4</sup> together with the bit-vector representation. Appendix Section B gives details.

Save/restore via a stack, as in Figure 1, is *pop-stack* domain restoration. The stack must be able to accommodate  $n$  copies of an array of  $n$  bit-vectors. When the number,  $n$ , of variables is large, e.g.  $n > 1000$ , and when the maximum number,  $\delta$ , of values in a domain is also large, the amount of memory required for  $\delta n^2$  bits may require attention. Instead of pushing the entire array,  $Dsets$ , of domain bit-vectors onto a stack, we only need to push bit-vectors that have changed since they were last pushed. These are easily identified, but at the cost of slowing the search.

If we use array (as well as bit-vector) representation of domains, we do not need a stack of arrays of domain bit-vectors. Array representation readily shows which values should be restored to domains, so individual bits can be changed from '0' to '1' accordingly. Appendix Section B.1 gives further details of this method of restoration, which we call *incremental* restoration<sup>5</sup>. Incremental restoration reduces memory requirements but is usually slower than pop-stack restoration that uses microprogrammed iteration to move a block of data within memory.

### 3.2 A bit-vector direct reduction procedure

This section introduces the first of three versions of the domain reduction procedure *reduce* that is called at Line 9 in Figure 1. The set of values in  $D_j$  that support  $u \in D_i$  is  $\{v | P_{ij}(u, v)\} \cap D_j$ . This set is represented by the bit-vector  $M_j^i[u] * Dsets[j]$ , where '\*' denotes bit-wise **and** and  $Dsets[j]$  is a bit-vector representing  $D_j$ . In bit-vector *direct* reduction [Ullmann 1976; Lecoutre and Vion 2008], a value  $u$  is removed<sup>6</sup> from domain  $D_i$  as soon as procedure *reduce* finds that  $M_j^i[u] * Dsets[j] = 0$ , where 0 is a bit-vector that represents the empty set.

Following McGregor [1979], the direct reduction procedure in Figure 2 uses a queue<sup>7</sup> of ordinals that identify variables; we think of this as being a queue of variables. If a domain  $D_i$  is reduced by removal of at least one value, then  $i$  is inserted into the queue to enable a subsequent iteration to check whether any value in an adjacent domain is no longer fully supported and should therefore be removed.

Appendix Section A.1 explains how, for each variable, we maintain a linked list of adjacent variables that have multivalued domains at the time of invocation of this procedure. **For each** at Line 4 in Figure 2 is implemented efficiently by traversing this list. Within this list for  $V_j$ , the record for the adjacent variable  $V_i$  includes a pointer to the bit-matrix  $M_j^i$ . This is how the procedure accesses bit-matrices.

At Line 6 we can find values currently in  $D_i$  by searching for 1's in  $Dsets[i]$ . Instead, to avoid visiting 0's in bit-vectors, we can use duplicate (bit-vector and array) representation of domains, as in Appendix Section B.1, although this does not always make the search faster.

<sup>4</sup>Lecoutre and Vion [2008] employ linked list duplicate representation. Array representation, with swapping as in Briggs and Torczon [1993], is somewhat faster because values can be restored to domains in constant time without being processed serially one-by-one.

<sup>5</sup>In other terminology [Schulte 1999], incremental restoration is an example of *trailing* and pop-stack restoration is an example of *copying*.

<sup>6</sup>If  $I$  is a bit-vector that represents  $D_i$ , and if  $U$  is a bit-vector that represents  $\{u\}$  and if  $u \in D_i$ , then  $u$  can be removed from  $I$  by  $I := \text{exclusiveOr}(I, U)$ .

<sup>7</sup>Represented by an array of ordinals and also by a bit-vector to allow rapid checks for set membership. Our experiments employ a simple first-in first-out queue because bit-vector reduction is so fast that the overheads of managing a priority queue [Wallace and Freuder 1992; Boussemart et al. 2004] appear to outweigh the benefit.

```

procedure reduce(in  $h$ : integer; in out  $Dsets$ : array of bit-vectors; out  $consistent$ : boolean);
input:  $h$  identifies the variable that has just been instantiated;
        Bit matrices are accessed via global variables;
input and output:  $Dsets$  is an array of bit-vectors representing domains;
output:  $consistent = (\text{no domain is empty})$ ;
begin
1  initialize queue to be empty; insert  $h$  into queue;
2  repeat
3     $j :=$  variable removed from queue;
4    for each  $i$  such that  $V_i$  is adjacent to  $V_j$  and  $D_i$  is multivalued do
        (* multivalued means multivalued at the time of invocation of this procedure *)
5      changed := false;
6      for  $u :=$  each value that is currently in  $D_i$  do
7        if  $M_j^i[u] * Dsets[j] = 0$  then (* There is no support in  $D_j$  for  $u$  in  $D_i$  *)
8          remove  $u$  from  $D_i$ ; changed := true;
9          if  $D_i$  is empty then  $w_{ij} := w_{ij} + 1$ ;  $consistent := \text{false}$ ; return end if;
10         end if
11      end for;
12      if changed and ( $i \notin \text{queue}$ ) then
          insert  $i$  into queue (*  $i$  will subsequently be a  $j$  that is removed from the queue *)
13      end if;
14  until queue is empty; (* The procedure has reached convergence *)
15   $consistent := \text{true}$ ;
end reduce;

```

Fig. 2. An introductory outline of a bit-vector direct reduction (BVDR) implementation of procedure *reduce*.

### 3.3 A bit-vector cumulative reduction procedure

With direct reduction, the innermost loop (Lines 6 through 11 in Figure 2) is the part of the entire search-and-reduction algorithm that is executed most frequently. McGregor [1979] used cumulative reduction<sup>8</sup> to simplify and optimize the innermost loop. Cumulative reduction is so called because it accumulates supported values (in a bit-vector  $B$  in Figure 3) instead of eliminating unsupported values immediately. Lines 4 and 6 in Figure 3 can be implemented in the same way as Lines 4 and 6 in Figure 2.

It is sometimes helpful to call procedure *reduce* between Lines 1 and 2, as well as at Line 9, in Figure 1. Domain reduction before commencement of search is an example of *preprocessing*. When used for preprocessing, the bit-vector cumulative reduction (BVCR) procedure is modified so that

- All variables are put in the queue initially, and
- All adjacent variables are processed, whether or not their domains are initially multivalued, and
- There is no provision for restoring eliminated values, because these are permanently eliminated from domains.

When any domain is empty there can be no solution, so the search does not proceed. Note also that if a domain is single-valued before commencement of search, we cannot be sure that its single value is fully supported. In this case, the direct and cumulative domain reduction procedures work correctly only after preprocessing.

<sup>8</sup>Cumulative reduction has also been employed in [Ullmann 1977; Lecoutre 2008; Cheng and Yap 2010].

```

procedure reduce(in  $h$ : integer; in out  $Dsets$ : array of bit-vectors; out  $consistent$ : boolean);
input:  $h$  identifies the variable that has just been instantiated;
        Bit matrices are accessed via global variables;
input and output:  $Dsets$  is an array of bit-vectors representing domains;
output:  $consistent = (\text{no domain is empty})$ ;
begin
1  initialize queue to be empty; insert  $h$  into queue;
2  repeat
3     $j :=$  variable removed from queue;
4    for each  $i$  such that  $V_i$  is adjacent to  $V_j$  and  $D_i$  is multivalued do
5       $B := 0$ ; (* A bit-vector wherein all bits are 0. This represents an empty set. *)
6      for  $v :=$  each value that is currently in  $D_j$  do
7        (*  $M_i^j[v]$  is the set of possible values of variable  $V_i$  supported by value  $v$  in  $D_j$  *)
8         $B := B + M_i^j[v]$  (* '+' denotes bitwise inclusive-or, implementing set union *)
9      end for;
10      $B := Dsets[i] * B$ ; (*  $B := \{u \in D_i \mid u \text{ is supported by value(s) in } D_j\}$  *)
11     if  $Dsets[i] \neq B$  then (*  $D_i$  has been reduced *)
12       if  $B = 0$  then  $w_{ij} := w_{ij} + 1$ ;  $consistent := \text{false}$ ; return end if
13        $Dsets[i] := B$ ; (* reducing domain  $D_i$  *)
14       if ( $i \notin \text{queue}$ ) then insert  $i$  into queue end if;
15     end if;
16   end for;
17   until queue is empty; (* at convergence *)
18    $consistent := \text{true}$ ;
end reduce

```

Fig. 3. An introductory outline of a bit-vector cumulative reduction (BVCR) implementation of procedure *reduce*.

### 3.4 A bit-vector forward checking procedure

The domain reduction procedures in Sections 3.2 and 3.3 usually reduce a domain most when they process it for the first time. If procedures are curtailed so that no domain is reduced more than once, this obviously takes less time but achieves less reduction than when reduction continues until convergence. McGregor [1979, Figure 5] introduced a simple curtailed reduction procedure that was given the name *Forward checking* by Haralick and Elliott [1980]. Forward checking<sup>9</sup> deserves attention because, in some circumstances, constraint satisfaction problems can be solved more quickly and simply by using this instead of the procedures in Sections 3.2 and 3.3. With forward checking the number of elective instantiations is much higher, but much less domain reduction work is done after each instantiation.

The forward checking procedure in Figure 4 is called instead of procedure *reduce* at Line 9 in Figure 1. Line 2 in Figure 4 is executed for adjacent variable  $V_i$  even if domain reduction has already reduced  $D_i$  to a single value. This is because the single value in  $D_i$  may not be supported by the value  $v$  of  $V_j$ . At the time of invocation of the forward checking procedure, every value in the domain of every variable that has not yet been electively instantiated is certainly supported by the value of every adjacent variable that was electively instantiated previously. But a value in the domain of a variable that was not electively instantiated previously may not be supported by any value in the domain of an adjacent variable that was not electively instantiated previously: forward checking does not check this.

With forward checking we use a version of procedure *choose* that may return  $i$  such that  $V_i$  has already been instantiated by implication. This is because procedure *forwardCheck*, when called after elective instantiation of an implied-instantiated variable, may remove values from further domains, thus speeding up the search [Ullmann 2007]. Appendix Section A.2 includes this version of procedure *choose*, and explains how Line 1 in Figure 4 avoids visiting adjacent variables that have not

<sup>9</sup>Forward checking is an embodiment of *preclusion* [Golomb and Baumert 1965].



```

procedure forwardCheck(in  $j, v$  : integer; in out  $Dsets$ : array of bit-vectors;
                      out  $consistent$ : boolean);
input:  $j$  identifies the variable that has just been instantiated;
        $v$  is the value that has been assigned to  $j$  by elective instantiation;
       Bit matrices are accessed via global variables;
input and output:  $Dsets$  is an array of bit-vectors representing domains;
output:  $consistent =$  (no domain is empty);
begin
1  for each variable  $V_i$  adjacent to  $V_j$  that has not been electively instantiated do
2     $Dsets[i] := Dsets[i] * M_i^j[v]$ ; (*  $D_i := D_i \cap \{u | u \text{ is supported by } v \in D_j\}$ .
    This assignment eliminates from  $D_i$  every value that is not supported by  $v \in D_j$  *)
3    if  $Dsets[i] = 0$  then  $w_{ij} := w_{ij} + 1$ ;  $consistent := \text{false}$ ; return end if;
4  end for
5   $consistent := \text{true}$ ;
end forwardCheck;

```

Fig. 4. An introductory outline of a forward checking domain reduction procedure.

been electively instantiated.

If the forward checking version of procedure *choose* finds that all domains are single-valued, then it returns *terminal* = **true**, but there is no guarantee that values of variables which have not been electively instantiated are fully supported. In other words, it is possible that the single values now in domains may not constitute a solution because there may be constraints that are not satisfied. To deal with this, the search could continue until all except one of the variables have been electively instantiated, but this would be unnecessarily inefficient. Instead, after the forward checking version of procedure *choose* returns *terminal* = **true**, the search process calls a procedure *allSatisfied* which explicitly checks that all binary constraints are satisfied. A combination of values, one in each domain, is accepted as a solution only if all constraints are satisfied.

## 4. FOCUS SEARCH

### 4.1 The static instantiation sequence in focus search

When domain reduction is interleaved with backtrack search, a domain may be restored before any benefit has been obtained from its reduction. In this respect, domain reduction processes are inevitably inefficient. Focus search is so called because it focuses reduction primarily on a small number of uninstantiated domains that may soon be instantiated electively. Focus search does not entirely avoid fruitless reduction operations, but it does avoid saving and restoring the results of these operations during the search. Like forward checking, focus search does not apply domain reduction iteratively until convergence; instead, again like forward checking, focus search relies on a very large number of invocations of a simple domain reduction procedure. Compared with bit vector cumulative reduction, forward checking performs a very much larger number of elective instantiations, and therefore the total cost of saving and restoring domains during the search is very much higher. It is important that focus search entirely avoids save/restore during the search.

For forward checking, another consequence of the large number of instantiations is that a large total amount of time is spent selecting successive variables for instantiation. Focus search avoids spending this time by working instead with a static instantiation sequence that is fixed before commencement of search and remains unchanged thereafter.

Section 2.2 explained that the procedures in Section 3 require duplicate representation of asymmetric constraint predicates: a predicate  $P_{ij}$  is represented both

```

1: AA-----
2: B-B-----
3: -C-C-----
4: ---DD-----
5: E---EE-----
6: --F-FFF-----
7: --G-G--G-----
8: -----H-H-----
9: -----III-----
10: J-----J-J-----
11: K-----K--K-----
12: -----L--L-----
13: -----M--MM-----
14: -----NNN-----
15: -----OO-O-----
16: ---P-----P--P---
17: ---Q-----QQ---
18: --R-----R-R--
19: -----S-----SS-
20: ---T-----T-----T

```

Fig. 5. An example in which columns correspond to successive variables in the static instantiation sequence  $0, 1, \dots, n-1$ . There is one row corresponding to each variable except the first. In the row corresponding to  $V_h$ , the *location* of the rightmost letter identifies  $V_h$ . Locations of other letters in this row identify preceding variables in  $Y_h$  that are adjacent to  $V_h$ .

by  $M_j^i$  and also by the transpose  $M_i^j$ . Because of its static instantiation sequence, focus search does not require this duplicate representation. For example, if  $V_7$  precedes  $V_3$  in the instantiation sequence, then  $M_3^7[v]$  will at some time be required but  $M_7^3[v]$  will never be required because  $V_3$  will never be instantiated before  $V_7$ .

In the static instantiation sequence for focus search, let  $Y_i$  be the set of variables that precede and include the variable  $V_i$ . The variable  $V_{i+1}$  is chosen so as to maximize<sup>10</sup> the number of variables in  $Y_i$  that are adjacent to  $V_{i+1}$ . An example in Figure 5 indicates the variables in  $Y_i$  to which a variable  $V_{i+1}$  is adjacent. In many cases, more than one variable not in  $Y_i$  is adjacent to the largest number of variables in  $Y_i$ : such ties are broken by a heuristic that favors the variable for which the sum of the branches of its adjacent variables is maximal. Here *branches* of a variable  $V_i$  means the sum of the degrees of the variables to which  $V_i$  is adjacent.

## 4.2 Focussed domain reduction

Like the algorithms in Section 3, focus search is a backtrack search algorithm that calls a domain reduction procedure *reduce* immediately after each elective instantiation. Each of the algorithms in Section 3 achieves both of the following effects in the same way:

- (1) Reduction of the number of values to which a variable  $V_i$  will be instantiated, thus reducing the number of elective instantiations.
- (2) Avoidance of parts of the search space that do not include any solution. If after instantiation of  $V_i$  the domain of any not-yet-instantiated variable contains no value that is fully supported, then the search proceeds no further until at least one variable has been re-instantiated.

In focus search, these two effects are achieved in different ways, both involving  $pastAdj(V_i)$ , which is the set of variables in  $Y_i$  that are adjacent to  $V_i$ .  $V_i$  is not included in  $pastAdj(V_i)$ . Within  $pastAdj(V_i)$ ,  $\lambda(i)$  is the last, and therefore closest

<sup>10</sup>This differs from the *maximum cardinality* algorithm of Tarjan and Yannakakis [1984] only in that ties are not broken arbitrarily.

to  $V_i$ , in the static sequence. In Figure 6, which outlines procedure *reduce* for focus search,  $f(h)$  is the current instantiated value of the variable  $V_h$ .

To achieve the first effect, focus search waits until all variables in  $pastAdj(V_j)$  have been instantiated and *then*, at Line 8 in Figure 6, makes  $D_j$  be the set of all values of  $V_j$  that are supported by values of variables in  $pastAdj(V_j)$ . The intention is that domain reduction is focussed on the single domain  $D_j$  from which values of  $V_j$  will next be selected for instantiation. Actually domain reduction is not so exactly focussed because  $V_i$  may possibly be the last variable in  $pastAdj(V_j)$  for more than one  $j$ , as can be seen in Figure 5. For example, if  $V_{\lambda(j)} = V_{\lambda(j+3)}$  then, immediately after instantiation of  $V_{\lambda(j)}$ , Lines 5 through 9 in Figure 6

- remove from  $D_j$  all values that are not supported by values of variables in  $pastAdj(V_j)$  and
- remove from  $D_{j+3}$  all values that are not supported by values of variables in  $pastAdj(V_{j+3})$

If any domain is now empty, procedure *reduce* returns *consistent* = **false**.

If, between the times of instantiation of  $V_{\lambda(j)}$  and  $V_j$ , there is backtrack such that a variable  $V_k$  after  $V_{\lambda(j)}$  and before  $V_j$  in the static sequence is re-instantiated, then  $D_j$  is not affected because, by definition of  $V_{\lambda(j)}$ ,  $V_k$  is not adjacent to  $V_j$ . If backtrack is such that  $V_{\lambda(j)}$  is re-instantiated then Lines 5 through 9 in Figure 6 recompute  $D_j$ . For these reasons, focus search does not save/restore  $D_j$ .

Lines 1 through 4 in Figure 6 achieve the second of the two effects mentioned in the first paragraph of this section<sup>11</sup>, after instantiation of the variable  $V_i$ . In Line 1 the condition  $(i \neq \lambda(j))$  prevents duplication of work that will be done at Lines 5 through 9. At Line 2,  $prec(i, j) = pastAdj(V_j) \cap Y_i$ , which means that  $prec(i, j)$  is the set of instantiated adjacent predecessors of  $V_j$ . At Line 2 focus search does not store the result bit-vector  $jD$  and therefore does not need to save/restore it. As well as avoiding save/restore, an advantage of not saving  $jD$  is that when this consists of hundreds of bits, copying it to memory takes appreciable time. A disadvantage is that, during subsequent invocations, Line 6 in Figure 6 in some cases re-computes intersections that were computed at Line 2, and could have been stored, during previous invocations of procedure *reduce*.

The focus search procedure *reduce* assumes that initialization has ensured that every value in  $M_j^h[v]$  is also in the initial  $D_j$  which is specified as part of the original formulation of the problem. This is why  $D_j^{initial}$  is not included in the intersection at Lines 2 and 6 in Figure 6.

### 4.3 The search procedure in focus search

Like the previous backtrack search procedure in Figure 1, focus search instantiates a variable  $V_i$  to successive values that are currently in domain  $D_i$ . Focus search employs a linked list of values that could possibly be in  $D_i$ , and refers to  $Dsets[i]$  to check whether each successive value is currently in  $D_i$ . For  $V_i$  the list could simply contain all values in  $D_i^{initial}$ , that is, the initial contents of  $D_i$  specified as part of the problem formulation. However, the following development improves efficiency.

For each variable  $V_i$ , except the first variable in the static instantiation sequence, we arbitrarily select just one variable,  $V_h$ , say, from the set  $pastAdj(V_i)$ . The selected variable  $V_h$  is the *selector* for  $V_i$ . For each possible value,  $u$ , of the selector  $V_h$ , we initialize  $seLists[i, u]$  to be a linked list representation of the set  $M_i^h[u] \cap D_i^{initial}$ . As before,  $M_i^h[u]$  represents the set of possible values of  $V_i$  that are supported by the value  $u$  of  $V_h$ . Thus  $seLists[i, u]$  represents the set of values in  $D_i$  supported by the value  $u$  in  $D_h$ . At Line 10 in Figure 7,  $selector[i] \uparrow = u$  such that  $u$  is the currently

<sup>11</sup>Focus search works correctly, but not so fast, if Lines 1 through 4 are omitted.

```

procedure reduce(i: integer; out consistent: boolean);
input:  i identifies the variable that has just been instantiated;
        Bit matrices are accessed via global variables;
output: Dsets is an array of bit-vectors representing domains;
        consistent = (no domain is empty);
begin
1  for each j such that ( $V_i \in \text{pastAdj}(V_j) \wedge (i \neq \lambda(j))$ ) do
    (* for each j such that i is an adjacent predecessor, but not the last, of j do *)
2     $jD := \bigcap_{V_h \in \text{prec}(i,j)} M_j^h[f(h)]$ ; (*  $jD := \{v | v \text{ is supported by the}$ 
        instantiated value of every variable in  $Y_i$  that is adjacent to  $V_j\}$  *)
3    if  $jD = 0$  then consistent := false; return end if;
4  end for;
5  for each j such that  $i = \lambda(j)$  do (* for each j such that i is the
    last adjacent predecessor of j do *)
6     $jD := \bigcap_{V_h \in \text{pastAdj}(V_j)} M_j^h[f(h)]$ ; (*  $jD := \{v | v \text{ is supported by the}$ 
        instantiated value of every variable in  $Y_i$  that is adjacent to  $V_j\}$  *)
7    if  $jD = 0$  then consistent := false; return end if;
8     $Dsets[j] := jD$ ;
9  end for;
10 consistent := true;
end reduce;

```

Fig. 6. Focus search procedure *reduce*. Here  $jD$  is a bit vector of the same type as  $Dsets[j]$ .

instantiated value of  $V_h$ , where  $V_h$  is the selector for  $V_i$ . What this achieves is that Line 5 only checks values that could be in  $D_i$  when  $V_h = u$ .

When  $i = \text{lastVar}$  at Line 11, each value in  $Dsets[\text{lastVar}]$  is certainly included in a solution, so there is no need to execute the loop at Line 3 through 24. Instead, Line 12 calls procedure *lastInstantiation*, which is:

```

while valuePtr  $\neq$  nil do
  if valuePtr↑.value in  $Dsets[i]$  then output solution end if;
  valuePtr := valuePtr↑.nextValue
end while;

```

This includes each value currently in  $Dsets[\text{lastVar}]$  in a separate solution. If only one solution is required, we amend this procedure to **return** after the first solution is found.

#### 4.4 List structures for focus search

Focus search needs to identify  $V_{\lambda(j)}$  as the last variable in  $\text{pastAdj}(V_j)$  at the time of instantiation of  $V_{\lambda(j)}$ . This would be impossible if dynamic variable ordering were applied. Another reason for using a purely static instantiation sequence is that we are able to construct lists, prior to commencement of search, that enable procedure *reduce* to find all  $h$  such that  $V_h \in \text{pastAdj}(V_j)$  without searching through any  $h$  such that  $V_h \notin \text{pastAdj}(V_j)$ . Prior to commencement of search, focus search also builds further list structures, as follows.

For each variable,  $V_j$ , there is a linked list comprising one record for each variable,  $V_{j_k}$ , in  $\text{pastAdj}(V_j)$ . The information in each such record is

- An ordinal,  $k$ , that identifies the variable  $V_{j_k}$ ,
- A pointer, which is the same as  $\text{valueAt}[k]$ , pointing to the current value  $f(j_k)$  of  $V_{j_k}$ ,
- A pointer to the bit-matrix that should be used at Lines 2 and 6 in Figure 6, and
- A pointer to the next record in  $\text{pastAdj}(V_j)$ .

```

procedure focusearch;
input:  Dsets which is an array [1..n] of bit-vectors such that Dsets[i] represents domain Di;
        initialized bit-matrices (to be referenced by the domain reduction procedure);
output: All solutions.
begin
1  initialize the instantiation sequence; initialize lists outlined in Section 4.4 ;
   initialize array successor so that successor[i] identifies the successor of Vi in the
   instantiation sequence; initialize array predecessor so that
   predecessor[i] identifies the predecessor of Vi in the instantiation sequence;
   predecessor[firstVar]:= nil; (* firstVar is the first variable in the instantiation sequence *)
2  i := firstVar; next[i]:= initialDlist; valuePtr:= initialDlist; (* points to list of
   possible values of firstVar *)
3  loop
4    if valuePtr ≠ nil then (* there is an untried value of Vi *)
5      if valuePtr.value in Dsets[i] then (* this value is in Di *)
6        v:= valuePtr.value; (* instantiate Vi to this value *) valueAt[i] ↑:= v;
        (* so procedure reduce can access v without dereferencing an array element *)
7        reduce(i, consistent); (* attempt domain reduction *)
8        if consistent then (* search can proceed *)
9          next[i]:= valuePtr.nextValue; (* pointing to next possible value of Vi *)
10         i:= successor[i]; valuePtr:= seLists[i, selector[i] ↑];
            (* valuePtr now points to the first possible value of the new Vi *)
11         if i = lastVar then (* all variables except VlastVar have been instantiated;
            lastVar is the last in the instantiation sequence *)
12           lastInstantiation(valuePtr, Dsets[i]); (* this instantiates Vi to each in
            turn of its available values without calling procedure reduce *)
13           i:= predecessor[i]; (* backtrack *)
14           valuePtr:= next[i]; (* points to the next untried value of this Vi *)
15         end if;
16         else valuePtr:= valuePtr.nextValue; (* points to the next untried value *)
17         end if
18         else valuePtr:= valuePtr.nextValue (* points to the next untried value *)
19         end if;
20     else (* backtrack *)
21       i:= predecessor[i];
22       if i = nil then exit (* end of search *)
23       else valuePtr:= next[i]; (* points to next untried value of Vi after backtrack *)
24       end if;
25     end if;
26 end loop;
27 conclude;
end
    
```

Fig. 7. The search procedure in focus search.

The intersection at Line 6 in Figure 6 is implemented efficiently by traversing this list.

For each variable,  $V_i$ , focus search also has a linked list in which each record points to a  $pastAdj(V_j)$  list, as in the previous paragraph, such that  $i = \lambda(j)$ . This list, which may be empty, enables procedure *reduce* to find quickly all  $j$  such that  $i = \lambda(j)$ . This is how **for each** is implemented at Line 5 in Figure 6.

For each variable  $V_i$ , focus search has yet another linked list, for use at Line 1 in Figure 6. In the list for  $V_i$  there is one record for each  $V_j$  such that  $V_i \in pastAdj(j)$  and  $i \neq \lambda(j)$ . Each such record points to a linked list of members of  $prec(i, j)$ . A list of members of  $prec(i, j)$  does not require additional memory because physically it is a sublist of the  $pastAdj(V_j)$  list. This is achieved by sorting the records in each  $pastAdj(V_j)$  list so that they contain variables in the reverse of the instantiation sequence. Thus the first variable in the  $pastAdj(V_j)$  list is  $V_{\lambda(j)}$ ; the last variable in this list is, within  $pastAdj(V_j)$ , the earliest variable in the instantiation sequence.

The list for  $prec(i, j)$ , which is used for computing the intersection at Line 2 in Figure 6, is implemented by a pointer to the record for  $V_i$  in the  $pastAdj(V_j)$  list.

As mentioned in Section 4.3, for each variable  $V_i$  and for just one of variable,  $V_h$ , that precedes  $V_i$  in the instantiation sequence, focus search has a separate linked list of records for each value,  $u$ , in  $D_h^{initial}$ . This is a linked list representation of the set  $M_i^h[u]$ . The total memory required for these lists depends on the average cardinality of  $\{v | P_{ij}(v, u)\}$ .

All of the lists used by focus search are constructed before the start of the search and are not changed thereafter. Of course these lists contribute to the total memory requirement for focus search. However, focus search does not maintain for each variable  $V_i$  a linked list of currently multivalued variables adjacent to  $V_i$ . Focus search does not have memory requirements, such as a stack of arrays of domains, associated with save/restore. Moreover, focus search does not require duplicate bit-matrix representation of asymmetric predicates.

#### 4.5 The search tree

For analytical purposes we can construct a search tree in which nodes correspond to elective instantiations. A node's descendent branches represent values successively assigned to that node by instantiation. If forward checking (FC) and focus search solve the same problem, both using the same static instantiation sequence, with values in domains remaining in the same static sorted order, their search trees are identical.

To see this, assume inductively that the search trees of both algorithms are identical up to the time immediately before instantiation of  $V_i$ . We need to show that forward checking (FC) and focus search will successively instantiate  $V_i$  to the same set of values, and that the truth-value of *consistent* will be the same for each such instantiation.

After instantiating a variable  $V_h$  a value  $u$ , FC removes from the domains of all uninstantiated adjacent variables all values that are not supported by  $V_h = u$ . When this has been done for  $V_h$  and, previously, for each of its predecessors, the domain of an uninstantiated variable  $V_j$  is

$$D_j^h = \{v \in D_j^{initial} | v \text{ is supported by the values of all variables in } pastAdj(j) \cap Y_h\}.$$

Whereas FC instantiates a variable  $V_j$  to each value in  $D_j^{j-1}$ , focus search instantiates  $V_j$  to each value in  $D_j^{\lambda(j)}$ , which is the set represented by  $Dsets(j)$  at Line 8 in Figure 6. By definition, no predecessor variable later than  $\lambda(j)$  is adjacent to  $V_j$ . Therefore  $pastAdj(j) \cap Y_h = pastAdj(j) \cap Y_{\lambda(j)}$ , whence FC and focus search instantiate  $V_j$  to the same set of values.

After domain reduction following instantiation of a variable  $V_i$ , let  $D_j^i$  be the domain of an uninstantiated variable  $V_j$ . FC returns *consistent* = **false** iff any  $D_j^i$  is empty. At Line 3 in Figure 6, focus search returns *consistent* = **false** iff any  $D_j^i$  is empty, where  $i \neq \lambda(j)$ . At Line 7 focus search returns *consistent* = **false** iff any  $D_j^i$  is empty, where  $i = \lambda(j)$ . Thus FC and focus search return the same truth-value of *consistent*.

When FC is not confined to the static instantiation sequence of focus search, the FC search tree is usually substantially smaller; but for each elective instantiation FC does more work than focus search. Appendix C explains that focus search is not merely a binary-constraint version of partition search, although both of these algorithms do belong to the same family.

## 5. SURROGATE VALUES

Long bit-vectors are physically implemented as arrays of single-word bit-vectors. Operations such as bitwise **and** and **or** require word-by-word processing, taking time that depends on the number of words. For this reason, and, more obviously, to reduce memory requirements, it is desirable to minimize the number of words by minimizing the total length of each bit-vector.

In the initial formulation of a constraint satisfaction problem, the discrete values given in each domain are the *data* values. A domain  $D_i$  may contain  $\delta_i$  data values within a range  $lowerBound \dots upperBound$  such that  $upperBound - lowerBound$  is very much greater than  $\delta_i$ . Examples are given in Section 7.2.

When  $upperBound - lowerBound \gg \delta_i$ , it may be worthwhile to work with *surrogate values* in the range  $0..\delta_i - 1$  instead of the original data values. The advantage is that we now have shorter bit vectors that comprise  $\delta$  bits, where  $\delta$  is the maximum, over all domains, of  $\delta_i$ . Memory requirements are reduced; bit-vector operations are faster.

We use a two dimensional array  $S$  to convert data values to surrogate values. The bounds of the first subscript are  $1..numberOfDomains$ ; the bounds of the second subscript are  $0..\delta - 1$ . Array element  $S[i, u]$  contains a data value,  $x$ , that is unique within domain  $D_i$ . The subscript  $u$  is the surrogate value that we use instead of the data value  $x$ . For each domain,  $D_i$ , the array  $S$  implements a function  $S_i$  such that  $x \equiv S_i(u)$ . The surrogate value  $u$  uniquely represents the data value  $x$  in this domain.

We use the symbol  $D_i$  to represent a domain given as part of the initial formulation of the constraint satisfaction problem; for notational simplicity, we use the same symbol  $D_i$  to represent the domain comprising surrogate values that represent data values in the original domain  $D_i$ . We solve a constraint satisfaction problem by working with domains containing these surrogate values, and applying surrogate predicates  $P_{ij}^s$  defined by

$$P_{ij}^s(u, v) \equiv P_{ij}(S_i(u), S_j(v))$$

where  $P_{ij}(x, y)$  is a predicate given in the original formulation of the problem. A surrogate solution,  $z^s$ , consists of exactly one surrogate value in each domain, satisfying  $P_{ij}^s(z_i^s, z_j^s)$  for all  $i, j$  such that there is a constraint between  $V_i$  and  $V_j$ . Let  $z$  be a set of data values, one in each domain, such that the value in domain  $D_i$  is  $S_i(z_i^s)$ . By construction,  $z$  satisfies  $P_{ij}(S_i(z_i^s), S_j(z_j^s))$  for all  $i, j$  such that there is a constraint between  $V_i$  and  $V_j$ , so  $z$  is a solution to the original problem. A solution to the original problem can be obtained from a surrogate solution  $z^s$  simply by replacing each value  $z_i^s$  in  $z^s$  by  $S[i, z_i^s]$ .

Predicates in frequency assignment problems in Section 7.2 are symmetric, but the derived surrogate predicates are not. Consider, for example, the simple predicate  $P_{ij}(x, y) = |x - y| > 8$ , where  $|x - y|$  denotes the absolute value of  $x - y$ . If

$$\begin{array}{ll} S_i(1) = -5 & S_j(1) = -3 \\ S_i(2) = 6 & S_j(2) = 2 \end{array}$$

then

$$\begin{array}{l} P_{ij}^s(1, 2) = |S_i(1) - S_j(2)| > 8 = \mathbf{false} \\ P_{ij}^s(2, 1) = |S_i(2) - S_j(1)| > 8 = \mathbf{true} \end{array}$$

Because surrogate predicates are asymmetric, the memory requirement for surrogate bit-matrices is  $2e\delta^2$  bits, where  $e$  is the number of binary constraints. With original data values instead of surrogates, and with symmetric predicates, the memory requirement for bit-matrices is  $e\Delta^2$  bits, where  $\Delta = upperBound - lowerBound + 1$ .

## 6. SUBGRAPH ISOMORPHISM

### 6.1 Constraint satisfaction problem formulation

A graph  $G = (V, E)$  consists of a set  $V = \{V_1, \dots, V_n\}$  of vertices and a set  $E$  of edges which are unordered pairs of vertices in  $V$ . There is a subgraph isomorphism between a graph  $G^\alpha = (V^\alpha, E^\alpha)$  and a graph  $G^\beta = (V^\beta, E^\beta)$  iff  $G^\alpha$  is isomorphic to a subgraph of  $G^\beta$ . A *subgraph isomorphism* is a function  $I : V^\alpha \rightarrow V^\beta$  such that  $(V_i^\alpha, V_j^\alpha) \in E^\alpha \Rightarrow (I(V_i^\alpha), I(V_j^\alpha)) \in E^\beta$  and  $(i \neq j) \Rightarrow (I(V_i^\alpha) \neq I(V_j^\alpha))$ . Thus a subgraph isomorphism is an injective function from  $V^\alpha$  to  $V^\beta$  that preserves adjacency. In almost all of the many practical applications of subgraph isomorphism mentioned by Messmer and Bunke [2000], attributes are associated with the vertices and edges of  $G_\alpha$  and  $G_\beta$ ; the attributes of corresponding vertices and edges are required to match. However, to review the basic business of subgraph isomorphism, we start by considering graphs that are not labelled with attributes and are not directed.

From the viewpoint of constraint satisfaction, variables identify vertices of the graph  $G^\alpha$ . Possible values for these variables are identifiers of vertices of the graph  $G^\beta$ . A constraint satisfaction problem is to assign to each vertex in  $V^\alpha$ , a value that identifies a vertex in  $V^\beta$ , subject to:

*Unary constraints.* The set  $D_i$  of allowed values of a variable  $V_i^\alpha$  identify vertices of  $G^\beta$  such that  $u \in D_i$  iff there is no known a priori reason why vertex  $V_u^\beta$  in  $G^\beta$  cannot correspond to vertex  $V_i^\alpha$  of  $G^\alpha$  in a subgraph isomorphism. For example,  $u \notin D_i$  if the degree of vertex  $V_u^\beta$  of  $G^\beta$  is less than the degree of vertex  $V_i^\alpha$  of  $G^\alpha$ .

*Binary constraints.* For each  $(i, j) \in E^\alpha$ , we have a predicate  $P_{ij}(u, v) = (u, v) \in E^\beta$ .

*AllDifferent constraint.* Every variable in  $V^\alpha$  has a value that differs from the value of every other variable in  $V^\alpha$ .

It is immediately clear that a subgraph isomorphism assigns values to variables so as to satisfy all of these constraints [McGregor 1979]. It is equally easy to see that every solution to this constraint satisfaction problem is a subgraph isomorphism.

For general binary constraints, Section 1 introduced  $M_j^i[u] = \{v | P_{ij}(u, v)\}$ . For subgraph isomorphism, this becomes  $M_j^i[u] = \{v | (u, v) \in E^\beta\}$ . Thus  $M_j^i[u]$  is a bit-vector that represents the set of all vertices of  $G^\beta$  that are adjacent to  $V_u^\beta$ . In other words,  $M_j^i$  is exactly the same thing as the adjacency matrix of the graph  $G^\beta$ .

Section 2.2 said that constraints are symmetric iff  $P_{ij}(u, v) = P_{ij}(v, u)$ . When graphs are not directed,  $V_u^\beta$  is adjacent to  $V_v^\beta$  iff  $V_v^\beta$  is adjacent to  $V_u^\beta$ ; in this case, for subgraph isomorphism,  $M_j^i$  is symmetric. Every edge in  $E^\alpha$  has the same constraint predicate, which is represented by the same symmetric matrix, which is the adjacency matrix of  $G^\beta$ . When graphs are not directed and not labelled, the overall total number of matrices required is one. In this context we denote this single matrix by  $M$ .

### 6.2 Invariant domain reduction

Domain reduction in Section 3 is *propagational* in the sense that removal of values from one domain may lead to removal of values from further domains, and so on. For isomorphism, *invariant* domain reduction is another important practical means of removing values from domains. A *vertex invariant* is a vertex property such that two vertices can correspond in a graph isomorphism only if their vertex invariants are identical. A simple example is the degree of a vertex [Corneil and Kirkpatrick 1980]; another example is a tuple whose components are the degrees of adjacent vertices sorted into decreasing order. Yet another [Fowler et al. 1983] is a tuple whose components are distance degrees, sorted into sequence of increasing distance.



The *distance* between two vertices is the number of edges in the shortest path between these two vertices. The *distance- $k$ -degree* of vertex  $V_i$  is the number of vertices at distance  $k$  from  $V_i$ . McKay [2009] lists many further vertex invariants. Invariant domain reduction is achieved by

- Determining vertex invariants of graph  $G^\alpha$  in isolation from graph  $G^\beta$ .
- Determining vertex invariants of graph  $G^\beta$  in isolation from graph  $G^\alpha$ .
- Evaluating, for each pair of vertices  $V_i^\alpha$  and  $V_u^\beta$ , a predicate whose arguments are the vertex invariants of  $V_i^\alpha$  and  $V_u^\beta$ , and removing  $u$  from domain  $D_i$  if this predicate returns **false**.

For graph isomorphism, but not for subgraph isomorphism, this predicate returns **true** iff the vertex invariants of  $V_i^\alpha$  and  $V_u^\beta$  are identical. For graph isomorphism it is useful to partition the set of vertices of a graph into subsets such that no two vertices within the same subset have different vertex invariants. This is *vertex partitioning*, which is fundamental to the success of Nauty [McKay 2009].

For subgraph isomorphism we need to specify allowed differences between the vertex invariants of  $V_i^\alpha$  and  $V_u^\beta$ . This specification can be formulated as a predicate that returns **false** if the vertex invariants of  $V_i^\alpha$  and  $V_u^\beta$  are such that  $V_u^\beta$  cannot correspond to  $V_i^\alpha$  in a subgraph isomorphism. A simple example is a predicate that returns **false** iff the degree of  $V_i^\alpha$  exceeds the degree of  $V_u^\beta$ . Section 6.4.2 provides further examples.

The essential difference between propagational and invariant domain reduction is that propagational domain reduction removes  $u$  from domain  $D_i$  if  $u$  is not fully supported by current domains of variables that are adjacent to  $V_i$ . Invariant domain reduction *ignores* domains of variables that are adjacent to  $V_i$ . From the viewpoint of the constraint satisfaction literature, invariant domain reduction simply removes values that do not satisfy unary constraints. This is not mandatory: The propagational bit-vector cumulative reduction (BVCR) algorithm works correctly if no unary constraints are applied and instead each domain is initialized to be  $\{v \mid V_v^\beta \in V^\beta\}$ . However, by applying unary constraints derived from vertex invariants we may reduce the time taken by subsequent preprocessing and search, without changing the outcome of the search.

Although the VF2 algorithm subgraph isomorphism algorithm [Cordella et al. 2004] is not propagational, it uses vertex-matching properties that vary during the search. Cordella et al have formulated this algorithm for the case where graphs are directed. For simplicity we now consider related domain-reduction rules for undirected graphs, using the following definition of *past* and *future* degree:

- pastDegreeOf*( $V_i^\alpha$ ) = cardinality of  $\{V_j^\alpha \mid V_j^\alpha \text{ is adjacent to } V_i^\alpha \text{ and the variable that corresponds to } V_j^\alpha \text{ is now electively instantiated}\}$ .
- futureDegreeOf*( $V_i^\alpha$ ) = cardinality of  $\{V_j^\alpha \mid V_j^\alpha \text{ is adjacent to } V_i^\alpha \text{ and the variable that corresponds to } V_j^\alpha \text{ is not currently instantiated}\}$ .

We define *pastDegreeOf*( $V_u^\beta$ ) and *futureDegreeOf*( $V_u^\beta$ ) similarly, except that  $V_u^\beta$  is here said to be instantiated electively iff there exists  $V_i^\alpha$  that is currently instantiated electively such that  $D_i = \{u\}$ . The BVCR algorithm could possibly be developed to apply the following extra domain reduction rule: include  $u$  in  $D_i$  only if

$$\begin{aligned} &(\text{pastDegreeOf}(V_i^\alpha) \leq \text{pastDegreeOf}(V_u^\beta)) \text{ and} \\ &(\text{futureDegreeOf}(V_i^\alpha) \leq \text{futureDegreeOf}(V_u^\beta)). \end{aligned}$$

Suppose that, for BVCR, domains are initialized so that  $u \in D_i$  only if  $\text{degreeOf}(V_i^\alpha) \leq \text{degreeOf}(V_u^\beta)$ . Consider a value  $u$  that survives in any domain  $D_i$  immediately after the BVCR domain reduction procedure has returned *consistent* = **true**. Every instantiated vertex  $V_j^\alpha$  that is adjacent to  $V_i^\alpha$  has a single-valued domain at this time.

If  $D_j = \{v\}$  then, since *consistent* = **true**,  $V_v^\beta$  is adjacent to  $V_u^\beta$ . Therefore  $\text{pastDegreeOf}(V_i^\alpha) = \text{pastDegreeOf}(V_u^\beta)$ . Initialization has ensured that  $\text{degreeOf}(V_i^\alpha) \leq \text{degreeOf}(V_u^\beta)$ , so  $\text{futureDegreeOf}(V_i^\alpha) \leq \text{futureDegreeOf}(V_u^\beta)$ . Therefore the proposed extra domain reduction rule has no effect because it is necessarily satisfied when the BVCR domain reduction procedure returns *consistent* = **true**.

Taking a step closer to VF2 [Cordella et al. 2004], the BVCR algorithm could possibly be developed to apply the additional domain reduction rule: include  $u$  in  $D_i$  only if

$$\begin{aligned} & \text{CardinalityOf}(\{V_v^\beta \text{ adjacent to } V_u^\beta \mid \text{pastDegreeOf}(V_v^\beta) > 0\}) \\ & \text{is not less than} \\ & \text{CardinalityOf}(\{V_j^\alpha \text{ adjacent to } V_i^\alpha \mid \text{pastDegreeOf}(V_j^\alpha) > 0\}) \end{aligned}$$

Immediately after the BVCR domain reduction procedure has returned *consistent* = **true** then, as in the previous paragraph, for any value  $u$  in any domain  $D_i$ ,  $\text{pastDegreeOf}(V_i^\alpha) = \text{pastDegreeOf}(V_u^\beta)$ . At this time,  $u \in D_i$  only if for each  $V_j^\alpha$  adjacent to  $V_i^\alpha$  there exists  $v \in D_j$  such that  $V_v^\beta$  is adjacent to  $V_u^\beta$ . Since  $\text{pastDegreeOf}(V_v^\beta) = \text{pastDegreeOf}(V_j^\alpha)$  the proposed additional domain reduction rule is necessarily satisfied and so has no effect within BVCR, although such rules may be valuable within VF2.

### 6.3 Bit-vector algorithms with the allDifferent constraint

**6.3.1 Cumulative and direct reduction.** With direct and with cumulative reduction, it is helpful to propagate the *allDifferent* constraint, as follows. When a domain becomes single-valued due to elective or implied instantiation, this single value should be deleted from all currently multivalued domains. If a further domain thereby becomes single-valued, then this single value should also be deleted from all currently multivalued domains, and so on.

We modify the direct reduction procedure, Figure 2, so that if the domain of a variable removed from the queue is single valued, this value is removed from all other domains that include it. If this makes any domain be empty, then the procedure returns *consistent* = **false**. Otherwise, if a value is removed from domain  $D_h$  then  $h$  is inserted into the queue. With direct reduction, each iteration of the outermost loop deals with single-valued domains before attempting domain reduction using binary constraints.

When a variable,  $j$ , is removed from the queue in the isomorphism version of the cumulative reduction procedure, the cardinality of the  $D_j$  is generally not known. Values in  $D_j$  are counted while using binary constraints to reduce domains of adjacent variables. Afterwards, if  $D_j$  is found to be single-valued, then this value is removed from all other domains that include it, and we proceed as in the previous paragraph. In this version of the cumulative reduction procedure, each iteration of the outermost loop attempts domain reduction using binary constraints before dealing with single-valued domains<sup>12</sup>. For direct and cumulative reduction, Section B.3 gives procedural details using duplicate (i.e. bit-vector plus array) representation of domains.

When seeking isomorphism, we use a cumulative reduction preprocessing routine that starts by putting successively at the head of queue all the variables whose domains are initially single-valued, and also by putting all other variables successively at the tail of the queue, as in Section 3.3. This preprocessing routine applies and

<sup>12</sup>In effect there is a constraint  $P_{i,j}(u,v) = (u \neq v)$  between every pair of variables, but this is not enforced in the same way as other binary constraints. Instead we wait until  $D_i$  or  $D_j$  is single-valued. If  $D_i = \{u\}$  we remove  $u$  from  $D_j$ . If  $D_j = \{v\}$  we remove  $v$  from  $D_i$ . At convergence of direct or cumulative reduction, all such constraints are satisfied.

propagates the *allDifferent* constraint as in the previous paragraph.

**6.3.2 The *allDifferent* constraint in forward checking.** Procedure *forwardCheck* is called just after a variable has been instantiated to a value  $v$ . As well as applying binary constraints, the isomorphism version of this procedure enforces the *allDifferent* constraint simply by removing the value  $v$  from all other domains that include it. If this empties any domain, then the procedure returns *consistent* = **false**. Procedure *forwardCheck* does not propagate the *allDifferent* constraint.

Section 3.4 noted that when the forward checking version of procedure *choose* finds that all domains are single-valued, we cannot be sure that all constraints are satisfied. With isomorphism forward checking, we also cannot be sure that the *allDifferent* constraint is satisfied. We therefore use a version of the constraint checking procedure *allSatisfied* which also checks that all values are different. When procedure *choose* returns *terminal* = **true**, we have a solution only if procedure *allSatisfied* finds explicitly that all constraints, including the *allDifferent* constraint, are satisfied.

**6.3.3 The *allDifferent* constraint in focus search.** For subgraph isomorphism, focus search is always preceded by preprocessing<sup>13</sup>, which initially imposes the *allDifferent* constraint. Preprocessing initializes *allowed* to be the set of all  $i$  such that  $D_i$  is multivalued at completion of preprocessing. This set is represented by a bit-vector of the type that represents a domain. In effect, Line 5 in Fig 7 is changed to

**if** valuePtr↑.value **in** (Dsets[ $i$ ] \* *allowed*) **then**

so a variable can only be instantiated to allowed values. When a variable  $V_j$  is instantiated to a value  $v$ , this value is removed from the set *allowed*, to prevent instantiation of other variables to the value  $v$ . Before  $V_j$  is re-instantiated, and also before the search backtracks to a preceding variable, this value  $v$  is restored to the set *allowed*. The focus search procedure *reduce* is modified so that, in effect,  $jD := jD \cap \text{allowed}$  is inserted between Lines 2 and 3, and also between Lines 6 and 7, in Fig 6. Thus  $jD$  cannot contain values belonging to domains that are already single-valued. Like forward checking, focus search does not propagate the *allDifferent* constraint.

For focus search the static instantiation sequence is determined after completion of preprocessing. Variables whose domains are single-valued after preprocessing and before commencement of search are put at the beginning of the instantiation sequence and are not visited by the search because they do not require re-instantiation. This arrangement is essential because during the search

**if** valuePtr↑.value **in** (Dsets[ $i$ ] \* *allowed*) **then**

prevents instantiation of domains that are already single-valued.

When one or more variables are excluded from the search because their domains are initially single-valued, the following situation arises. An excluded variable may be  $V_{\lambda(i)}$  for a variable  $V_i$  whose domain is initially multivalued, so  $Dsets[i]$  will not be properly determined by the focus search procedure *reduce*. In this case we assign to  $Dsets[i]$  the bit-vector that represents domain  $D_i$  at the time of return from the preprocessing procedure.

## 6.4 The local *allDifferent* constraint

**6.4.1 Bipartite matching.** Let  $A_i^\alpha$  be the set of all vertices adjacent to vertex  $V_i^\alpha$  in  $G^\alpha$ . Similarly, let  $A_u^\beta$  be the set of all vertices adjacent to  $V_u^\beta$  in  $G^\beta$ . If an isomorphism,  $I$ , exists such that  $I(V_i^\alpha) = V_u^\beta$  then there exists an injective function  $L : A_i^\alpha \rightarrow A_u^\beta$  such that

<sup>13</sup>When used before focus search, BVCR preprocessing is implemented using linked-list instead of array representation of domains, because this is appreciably faster when there is no save/restore.

```

procedure prematches(in  $i, u$ : integer): boolean;
input:     $i$  and  $u$  identify vertices in  $G^\alpha$  and  $G^\beta$  respectively;
output:  the predicate prematches returns true if  $V_u^\beta$  could
           correspond to  $V_i^\alpha$  in a subgraph isomorphism;
begin
  if  $\text{degreeOf}(V_i^\alpha) > \text{degreeOf}(V_u^\beta)$  then return false end if;
  for  $k := 1$  to  $\text{degreeOf}(V_i^\alpha)$  do
    if  $\text{degreeOf}(V_{i_k}^\alpha) > \text{degreeOf}(V_{u_k}^\beta)$  then return false end if;
  end for;
  return true
end prematches;

```

Fig. 8. Predicate *prematches* for undirected unlabelled graphs. This requires vertices in  $A_i^\alpha$  and  $A_u^\beta$  to be in sequence of decreasing degree.

- For all  $V_j^\alpha \in A_i^\alpha$ ,  $L(V_j^\alpha) = V_v^\beta$  such that  $(v \in D_j) \wedge (V_v^\beta \in A_u^\beta)$ , and
- $L(V_j^\alpha) \neq L(V_k^\alpha)$  for all  $V_j^\alpha, V_k^\alpha$  in  $A_i^\alpha$  such that  $j \neq k$ .

Here, ‘ $L$ ’ stands for *local*: the function  $L$  involves an *allDifferent* constraint that is local in that it is restricted to  $A_i^\alpha$ . Indeed  $L$  is the restriction of  $I$  to  $A_i^\alpha$ . If no such function  $L$  exists then no function  $I$  exists such that  $I(V_i^\alpha) = V_u^\beta$  and in this case  $u$  can be removed from domain  $D_i$ . Solnon [2010] seeks a function  $L$  by incremental application of the bipartite matching algorithm of Hopcroft and Karp [1973]. For each  $(i, u)$  the worst-case complexity is  $O(|A_i^\alpha| \times |A_u^\beta|)$ .

Algorithms in Section 6.3 certainly ensure that all values in a solution are different without applying the local *allDifferent* constraint. For each  $(i, u)$  the complexity of these bit-vector procedures is  $O(|A_i^\alpha|)$  for

**for each**  $j \in A_i^\alpha$  **do if**  $D_j \cap M_j^i[u] = \emptyset$  **then ...**

but, without applying the local *allDifferent* constraint, we may remove fewer values from domains than does Solnon’s process, and therefore explore more combinations of values. This provides another example of a trade-off between doing more work after each elective instantiation, or instead doing less work but requiring more elective instantiations.

Instead of seeking a function  $L$ , Larrosa and Valiente [2002] apply a necessary but not sufficient condition for existence of a function  $L$ . The effect is to do less work than Solnon [2010] after each elective instantiation but remove fewer values from domains. Compared with our bit-vector procedures, Larrosa and Valiente may remove more values from domains but without the advantage of bit-parallel  $D_j \cap M_j^i[u]$  in the innermost loop of the algorithm.

**6.4.2 Prematching.** Before preprocessing and search, when domains are such that

$$u \in D_i = (\text{degreeOf}(V_i^\alpha) \leq \text{degreeOf}(V_u^\beta))$$

a function  $L$  can be sought by the very simple predicate procedure in Fig 8 which has time complexity  $O(|A_i^\alpha|)$ . This works correctly only if the vertices in

$$A_i^\alpha = \{V_{i_1}^\alpha, \dots, V_{i_k}^\alpha, \dots, V_{i_{\text{degreeOf}(V_i^\alpha)}}^\alpha\}$$

have been sorted into sequence of decreasing degree and the vertices in

$$A_u^\beta = \{V_{u_1}^\beta, \dots, V_{u_k}^\beta, \dots, V_{u_{\text{degreeOf}(V_u^\beta)}}^\beta\}$$

have also been sorted into this sequence. To see this, note that if  $\text{degreeOf}(V_{i_1}^\alpha) > \text{degreeOf}(V_{u_1}^\beta)$  then no function  $L$  exists because  $V_{u_1}^\beta$  is the vertex in  $A_u^\beta$  that has the largest degree. If  $\text{degreeOf}(V_{i_1}^\alpha) \leq \text{degreeOf}(V_{u_1}^\beta)$  then *prematches* associates

$V_{u_1}^\beta$  with  $V_{i_1}^\alpha$  because there is no other member of  $A_i^\alpha$  that has higher degree than  $V_{i_1}^\alpha$  and should therefore be associated with  $V_{u_1}^\beta$ . If  $\text{degreeOf}(V_{i_2}^\alpha) > \text{degreeOf}(V_{u_2}^\beta)$  then no function  $L$  exists because  $V_{u_2}^\beta$  is the member of  $\{V_{u_2}^\beta, \dots, V_{u_{\text{degreeOf}(V_{u_2}^\beta)}}^\beta\}$  that has highest degree and  $V_{u_1}^\beta$  is not available because it has already been associated with  $V_{i_1}^\alpha$ . Similar reasoning can be applied successively to the remaining members of  $A_i^\alpha$ .

The predicate *prematches* compares vertex invariants. For each vertex the invariant is the degree of the vertex together with an ordered list of degrees of adjacent vertices. This is one of the first vertex invariants in a succession employed by Zampelli et al [2010]. The next in this succession is the degree of the vertex together with, for each adjacent vertex  $V_a$ , the degree of  $V_a$  and an ordered list of degrees of vertices adjacent to  $V_a$ . Zampelli et al continue this succession until it yields no further domain reduction.

In what follows, *prematching* means removing from each domain  $D_i$  each value  $u$  such that *prematches*( $i, u$ ) = **false**. For initial domain reduction, before preprocessing and search, prematching usually turns out to be so cost-efficient that we will formulate versions of it for directed and for labelled subgraph isomorphism.

### 6.5 Directed unlabelled graphs

In a directed graph, an edge, which in this context is known as an *arc*, is an ordered pair of vertices;  $(V_i^\alpha \rightarrow V_j^\alpha)$  denotes an arc from  $V_i^\alpha$  to  $V_j^\alpha$ . When directed subgraph isomorphism is formulated as a constraint satisfaction problem, there is a binary constraint corresponding to every arc in  $G^\alpha$ . For an arc  $(V_i^\alpha \rightarrow V_j^\alpha)$  the constraint predicate is  $P_{i,j}(u, v) = (V_u^\beta \rightarrow V_v^\beta) \in E^\beta$  and the bit matrix  $M_j^i$  is such that  $M_j^i[u] = \{v | (V_u^\beta \rightarrow V_v^\beta) \in E^\beta\}$ . For an arc  $(V_i^\alpha \rightarrow V_j^\alpha)$  the constraint satisfaction algorithms in Section 3 require the asymmetric matrix  $M_j^i$  and also its transpose  $M_i^j$ . This pair of matrices is the same for all  $(i, j)$  such that  $(V_i^\alpha \rightarrow V_j^\alpha) \in E^\alpha$ . For an arc  $(V_h^\alpha \leftarrow V_k^\alpha)$  the constraint satisfaction algorithms in Section 3 require the asymmetric matrix  $M_k^h$  such that  $M_k^h[u] = \{v | (V_u^\beta \leftarrow V_v^\beta) \in E^\beta\}$  and also its transpose  $M_h^k$ . Again, this pair of matrices is the same for all  $(h, k)$  such that  $(V_h^\alpha \leftarrow V_k^\alpha) \in E^\alpha$ . Thus for directed unlabelled graphs the overall total number of matrices required is four.

The *indegree* of a vertex is the number of arcs directed towards that vertex; the *outdegree* is the number of arcs directed away from that vertex. For directed subgraph isomorphism we can apply unary constraints that allow  $u \in D_i$  only if the indegree of  $V_u^\beta$  is not less than the indegree of  $V_i^\alpha$  and also the outdegree of  $V_u^\beta$  is not less than the outdegree of  $V_i^\alpha$ . Fig 9 shows a version of the predicate *prematches* that enforces stronger unary constraints which may eliminate more values from domains. For this we define

$$\overrightarrow{A_i^\alpha} = \{V_j^\alpha | (V_i^\alpha \rightarrow V_j^\alpha) \in E^\alpha\}$$

and

$$\overrightarrow{A_i^\alpha} = \{\overrightarrow{V_{i_1}^\alpha}, \dots, \overrightarrow{V_{i_k}^\alpha}, \dots, \overrightarrow{V_{i_{\text{outDegreeOf}(V_i^\alpha)}}^\alpha}\}$$

in which the vertices have been sorted into sequence of decreasing degree, where  $\text{degree} = \text{indegree} + \text{outdegree}$ . We define  $\overleftarrow{A_i^\alpha}$ ,  $\overleftarrow{A_u^\beta}$  and  $\overleftarrow{A_u^\beta}$  similarly.

An example in Table I illustrates what happens if we compare indegrees and outdegrees instead of simply comparing degrees in the last two loops in Fig 9. In this example there are four vertices in  $\overrightarrow{A_i^\alpha}$  and also in  $\overrightarrow{A_u^\beta}$ ; Table I shows their indegrees and outdegrees. We seek a function  $\vec{L} : \overrightarrow{A_i^\alpha} \rightarrow \overrightarrow{A_u^\beta}$  such that for all  $k$  in  $1, \dots, 4$

$$-\text{inDegreeOf}(\overrightarrow{V_{i_k}^\alpha}) \leq \text{inDegreeOf}(\vec{L}(\overrightarrow{V_{i_k}^\alpha})) \text{ and}$$

```

procedure prematches(in i, u: integer): boolean;
input:    i and u identify vertices in  $G^\alpha$  and  $G^\beta$  respectively;
output:  the predicate prematches returns true if  $V_u^\beta$  could
           correspond to  $V_i^\alpha$  in a subgraph isomorphism;
begin
  if (outDegreeOf( $V_i^\alpha$ ) > outDegreeOf( $V_u^\beta$ )) or
    (inDegreeOf( $V_i^\alpha$ ) > inDegreeOf( $V_u^\beta$ )) then return false
  end if;
  for k := 1 to outDegreeOf( $V_i^\alpha$ ) do
    if degreeOf( $\overrightarrow{V_{i_k}^\alpha}$ ) > degreeOf( $\overrightarrow{V_{u_k}^\beta}$ ) then return false end if;
  end for;
  for k := 1 to inDegreeOf( $V_i^\alpha$ ) do
    if degreeOf( $\overleftarrow{V_{i_k}^\alpha}$ ) > degreeOf( $\overleftarrow{V_{u_k}^\beta}$ ) then return false end if;
  end for;
  return true
end prematches;

```

Fig. 9. Predicate *prematches* for directed unlabelled graphs. This requires that vertices adjacent to  $V_i^\alpha$  and  $V_u^\beta$  must be in sequence of decreasing degree.

$\overrightarrow{A_i^\alpha}$		$\overrightarrow{A_u^\beta}$	
indegree	outdegree	indegree	outdegree
4	2	4	3
3	2	3	4
2	4	3	2
2	3	2	3

Table I. Example of indegrees and outdegrees of four vertices adjacent to  $V_i^\alpha$  and  $V_u^\beta$ .

—outDegreeOf( $\overrightarrow{V_{i_k}^\alpha}$ )  $\leq$  outDegreeOf( $\overrightarrow{V_{i_k}^\alpha}$ ) and  
 — $\vec{L}$  satisfies the *allDifferent* constraint.

For this example there is no lexicographic ordering on (indegree, outdegree) pairs in  $\overrightarrow{A_i^\alpha}$  and in  $\overrightarrow{A_u^\beta}$  such that a function  $\vec{L}$  exists if and only if for all  $k$  in  $1, \dots, 4$

$$(\text{inDegreeOf}(\overrightarrow{V_{i_k}^\alpha}) \leq \text{inDegreeOf}(\overrightarrow{V_{u_k}^\beta})) \text{ and } (\text{outDegreeOf}(\overrightarrow{V_{i_k}^\alpha}) \leq \text{outDegreeOf}(\overrightarrow{V_{u_k}^\beta}))$$

For the example in Table I, a function  $\vec{L}$  does exist but finding it requires a process that has worst-case time complexity greater than  $O(|\overrightarrow{A_i^\alpha}|)$ .

## 6.6 Labelled undirected graphs

Vertices and edges may be labelled with information of any kind whatsoever. For example, labels may be numbers, or characters, or vectors, or sets. We now seek subgraph isomorphisms,  $I$ , such that

—attributeOf( $I(V_i^\alpha)$ ) = attributeOf( $V_i^\alpha$ ) **and**

—( $V_i^\alpha, V_j^\alpha$ )  $\in E_\alpha \Rightarrow (I(V_i^\alpha), I(V_j^\alpha)) \in E_\beta$  **and**  
 attributeOf( $I(V_i^\alpha), I(V_j^\alpha)$ ) = attributeOf( $V_i^\alpha, V_j^\alpha$ ) **and**

— $i \neq j \Rightarrow (I(V_i^\alpha) \neq I(V_j^\alpha))$

In molecule-matching applications [Willett 1999] attributes of vertices are letters that identify atoms, e.g., ‘C’, ‘H’, ‘O’, and attributes of edges are bond-types, e.g., single, double, or aromatic. In the present section we insist, for simplicity, that attributes match exactly, although a more general requirement is that a predicate

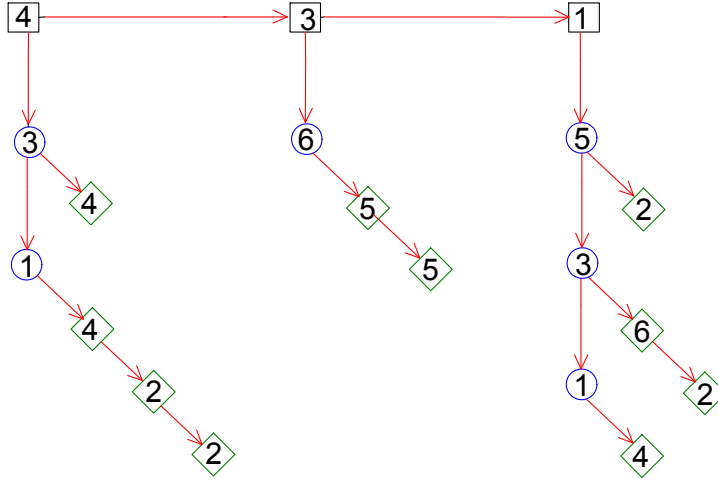


Fig. 10. Triple list structure. Numbers in the horizontal list are edge attribute values. Numbers in vertical lists are vertex attribute values. Numbers in diagonal lists are degrees of individual vertices.

whose arguments may include attributes of  $V_i^\alpha$  and  $I(V_i^\alpha)$  returns **true** when there is a match within prescribed tolerances [Proschak et al. 2007].

When edges are not labelled, we have  $M[u] = \{v | (u, v) \in E^\beta\}$ , as in Section 6.1. When edges are labelled, more than one edge in  $E^\alpha$  may have the same attribute. Let  $\lambda_1, \dots, \lambda_h, \dots, \lambda_\eta$  be the set of distinct edge attributes that occur in  $E^\alpha$ . For an edge  $(i, j)$  in  $E^\alpha$  that has edge-attribute  $\lambda_h$ , and for the case where exact match is required, the constraint predicate is

$$P_h(u, v) = ((u, v) \in E^\beta) \wedge \text{attributeOf}(u, v) = \lambda_h$$

and this is represented by a bit-matrix  $M_h[u] = \{v | P_h(u, v)\}$ . Thus we require  $\eta$  distinct bit-matrices, one for each distinct edge-attribute that occurs in  $E^\alpha$ . For each edge in  $E^\alpha$  a domain reduction procedure such as BVCR now works with whichever bit-matrix has the same attribute as that edge. The binary constraint predicates are symmetric, so each is represented by a single bit-matrix, and the total number of such matrices is  $\eta$ . In molecule matching applications,  $\eta$  is usually very much less than the number of edges in  $E^\alpha$ , indeed often less than eight.

We take the liberty of saying that vertex  $V_j$ , adjacent to vertex  $V_i$ , has edge-attribute  $\lambda_h$ , meaning that the edge  $(i, j)$  has edge-attribute  $\lambda_h$ . Let  $\nu_1, \dots, \nu_k, \dots, \nu_\zeta$  be the set of different vertex attributes that occur in  $V^\alpha$ . We define the  $hk$ -attribute-degree of vertex  $V_i$  to be the number of vertices  $V_j$  adjacent to  $V_i$  such that  $V_j$  has edge-attribute  $\lambda_h$  and vertex-attribute  $\nu_k$ . It is reasonable to include  $u \in D_i$  only if for all  $h \in [1, \dots, \eta]$  and for all  $k \in [1, \dots, \zeta]$  the  $hk$ -attribute-degree of vertex  $V_i^\alpha$  is not greater than the  $hk$ -attribute-degree of a vertex  $V_u^\beta$ .

Going further, the version of predicate *prematches* in Fig 11 applies the local *allDifferent* constraint. For each  $hk$  pair, this version of *prematches* seeks correspondence between degrees of individual adjacent variables. Before the first call of *prematches*, we construct a separate triple list structure for each vertex. This vertex invariant structure is of the same type for every vertex in  $V^\alpha$  and in  $V^\beta$ , and remains unchanged during all invocations of *prematches*. This structure is now described for a vertex  $V_i$ .

For  $V_i$  there is an outer list that has one member for each different edge attribute of any vertex adjacent to  $V_i$ . The member of this list for edge attribute  $\lambda_h$  points to a further list that has one member for each different vertex attribute of any vertex

adjacent to  $V_i$  that has edge attribute  $\lambda_h$ . The member of this further list that corresponds to edge attribute  $\lambda_h$  and vertex attribute  $\nu_k$  points to a final list that has one member for each vertex (adjacent to  $V_i$ ) that has edge attribute  $\lambda_h$  and vertex attribute  $\nu_k$ . In this final list, the member for each vertex  $V_j$  includes the degree of  $V_j$ . Every final list must be in sorted order of decreasing degree; more than one vertex in a final list may have the same degree. There are no duplicates in the outermost list and in the middle lists, which must be in a sorted order that is the same for all vertices. These lists are here assumed to be in decreasing sequence of attribute value<sup>14</sup>. Figure 10 shows the list structure for an example in which the (edge-attribute, vertex-attribute) pairs for the ten vertices adjacent to  $V_i$  are (4, 3), (4, 1), (3, 6), (1, 5), (1, 3), (1, 1). In practice many final lists have only one member.

In Fig 11 the loop at Lines 24 through 30 simply compares degrees. To compare  $hk$ -attribute-degrees instead, we would require an algorithm that has non-linear time complexity, for the reason that was illustrated by the example in Table I. However there is no problem when the loop at Lines 23 through 29 is amended to compare distance-2-degrees instead of distance-1-degrees.

When a single graph  $G^\alpha$  is compared with a single graph  $G^\beta$ , the memory occupied by the lists can be released after the initial membership of domains has been determined. In systems that have dynamic memory allocation, this area of memory can be re-used for bit-matrices, which can be constructed after all unary constraints have been applied.

### 6.7 Graph retrieval

Most practical applications of subgraph isomorphism involve comparison of many pairs of graphs. In image matching applications, many small graphs are compared with a single bigger graph [Conte et al. 2004]. Contrariwise, in chemical applications [Brown 2009; Willett 1999] a single molecular substructure is compared with many bigger molecules. More generally, graph retrieval is concerned with a large collection of graphs, which are known in this context as *targets*. Given a smaller graph, here known as a *query*, a common practical requirement is to identify all targets to which the query is subgraph isomorphic [Cheng et al. 2009; Zou et al. 2008]. A more general requirement, which is not addressed in this article, is to find all targets to which the query is *approximately* subgraph isomorphic [Yan et al. 2006].

The simple routine:

**for** each target **do** seek subgraph isomorphism between query and target

would usually be intolerably slow. Instead, normal practice is to employ a set, sometimes known as an *alphabet*, of small subgraphs, known as *fragments*. We identify all fragments that are in the query and also in the alphabet, and we do the same for every target. For the set of targets, this information can be re-used for any number of queries. If the set of fragments in a given target does not include the set of fragments in the query, then the query cannot possibly be subgraph isomorphic to this target, and a search for isomorphism should therefore be omitted. The alphabet of fragments is chosen with the aim of maximizing avoidance of fruitless searches.

For each fragment in the alphabet, Yan et al [2005] compile an index list of (identifiers of) all target graphs that include this fragment. Yan et al seek detailed subgraph isomorphism between the query and each of the target graphs within the intersection of the index lists for those fragments that are included in the query. If the indexed fragments are small then we may need many of them, each associated with a list of many thousands of target identifiers. Intersecting many long lists

<sup>14</sup>When these attributes are letters such as 'C', 'H', 'O', a list is sorted in decreasing sequence of ordinals uniquely associated with letters. This association must be the same for all vertices.



```

procedure prematches(in  $i, u$  : integer): boolean;
input:     $i$  and  $u$  identify vertices in  $G^\alpha$  and  $G^\beta$  respectively;
output:  the predicate prematches returns true if  $V_u^\beta$  could
           correspond to  $V_i^\alpha$  in a subgraph isomorphism;

begin
1  if vertexAttributeOf( $V_i^\alpha$ )  $\neq$  vertexAttributeOf( $V_u^\beta$ ) then return false end if
2  alphaEdgePtr := pointerToFirstRecordInListOfEdgeAttributesOf  $V_i^\alpha$ ;
3  betaEdgePtr := pointerToFirstRecordInListOfEdgeAttributesOf  $V_u^\beta$ ;
4  while alphaEdgePtr  $\neq$  nil do (* iterate over edge attributes *)
5    loop (* to find identical edge attributes *)
6      if (betaEdgePtr = nil) or
7        betaEdgePtr $\uparrow$ .attribute < alphaEdgePtr $\uparrow$ .attribute then return false
8      elsif betaEdgePtr $\uparrow$ .attribute = alphaEdgePtr $\uparrow$ .attribute then exit
9      else betaEdgePtr := betaEdgePtr $\uparrow$ .next
10     end if
11   end loop;
12   alphaVertexPtr := alphaEdgePtr $\uparrow$ .pointerToFirstRecordInListOfVertexAttributes;
13   betaVertexPtr := betaEdgePtr $\uparrow$ .pointerToFirstRecordInListOfVertexAttributes;
14   repeat (* iterate over vertex attributes *)
15     loop (* to find identical vertex attributes *)
16       if (betaVertexPtr = nil) or
17         (betaVertexPtr $\uparrow$ .attribute < alphaVertexPtr $\uparrow$ .attribute) then return false
18       elsif betaVertexPtr $\uparrow$ .attribute = alphaVertexPtr $\uparrow$ .attribute then exit
19       else betaVertexPtr := betaVertexPtr $\uparrow$ .next
20       end if
21     end loop;
22     alphaDegreePtr := alphaVertexPtr $\uparrow$ .pointerToFirstRecordInListOfVertices;
23     betaDegreePtr := betaVertexPtr $\uparrow$ .pointerToFirstRecordInListOfVertices;
24     loop (* comparing degrees of individual adjacent vertices *)
25       if alphaDegreePtr $\uparrow$ .degree > betaDegreePtr $\uparrow$ .degree then return false end if;
26       alphaDegreePtr := alphaDegreePtr $\uparrow$ .next;
27       if alphaDegreePtr = nil then exit end if
28       betaDegreePtr := betaDegreePtr $\uparrow$ .next;
29       if betaDegreePtr = nil then return false end if
30     end loop; (* over individual vertices *)
31     alphaVertexPtr := alphaVertexPtr $\uparrow$ .next;
32   until alphaVertexPtr = nil;
33   alphaEdgePtr := alphaEdgePtr $\uparrow$ .next;
34 end while;
35 return true
end prematches

```

Fig. 11. Predicate *prematches* for labelled undirected graphs.

takes considerable time. Having larger fragments in the alphabet can reduce the length of these lists, but the number of possible fragments increases exponentially with their size [Jiang et al. 2007].

The use of signatures is a well-established alternative to the use of indexes. The simplest *bit-string* method associates with each target, and also with a query, a bit-vector in which there is one bit corresponding to each fragment in the alphabet [Willett 2005]. These bit-vectors typically comprise 1000 bits. In a target's bit-vector, a bit is '1' iff the corresponding fragment is in the target. Similarly, in a query bit-vector a bit is '1' iff the corresponding fragment is in the query. If bit-parallel operations reveal that any '1' in the query bit-vector corresponds to a '0' in a target bit-vector, then isomorphism between the query and this target should not be sought. In this context, a bit-vector is an example of a *signature*, that is, a

```

for each target do
  for each trilabel in the query do
    if the number of occurrences in the query does not exceed the
      number of occurrences of this trilabel in the target then
      if prematching does not empty any domain then
        if preprocessing does not empty any domain then
          seek subgraph isomorphism between query and target

```

Fig. 12. Brief outline of graph retrieval procedure.

collection of information that characterizes a target.

For textual information retrieval, Zobel et al [1998] found that index-based methods substantially out-performed signature methods. However, the number of fragments involved in a conjunctive graph retrieval query is usually not so small. This is one reason why, for graph retrieval, bit-vector methods are well established in practice [Daylight Chemical Information Systems, Inc 2007] whereas index-based methods are still subject to research investigation [Zhang et al. 2009; Zhao et al. 2007; Cheng et al. 2009].

Fragment-based methods require judicious selection of the alphabet. Selection requires substantial effort, which may need to be repeated for each different collection of targets. Moreover, detection of selected fragments within a query may take appreciable time. These disadvantages can be avoided by using a signature method with very simple fragments, and counting their occurrences instead of merely detecting their presence or absence. More specifically, following [Chang and Lee 1991], our experiments work with trilabels.

A *trilabel* is a triple comprising the label of a vertex, the label of an adjacent vertex, and the label of the edge between these two vertices. A trilabel, which is the same thing as a *2-graph* [Chou and Shapiro 1998] and a *distinct edge* [Cheng et al. 2009] is analogous to a textual trigram. Prior to retrieval we store, for each target graph, a signature which consists of the number of occurrences of each and every trilabel in that target. Signatures can subsequently be used in answering any number of queries. At query time we count the number of occurrences of each and every trilabel in the query and then proceed as in Fig 12. For each trilabel in the query signature, this routine requires a search for the same trilabel in the target signature. Appendix Section D describes a fast trilabel matching procedure.

As in Graph Grep [Giugno and Shasha 2002; Shasha et al. 2002], paths of length greater than one could be used instead of trilabels, but of course the number of paths increases rapidly with their length. Instead of using longer paths, we rely on prematching and preprocessing to reduce the number of searches for subgraph isomorphism. Prematching and preprocessing achieve domain reduction that facilitates search, whereas trilabel matching does not. The trilabel test simply prevents further work on comparisons that can certainly not yield any isomorphism. Another comment is that neither prematching nor preprocessing has worst-case exponential time-complexity.

## 7. EXPERIMENTS

### 7.1 Experiments with randomly generated constraint satisfaction problems

**7.1.1 Experimental framework.** Section 7.1 reports experiments with randomly generated instances of the general binary constraint satisfaction problem, not with isomorphism. Parameters that characterize an instance are:

$n$ , the number of variables.

$e$ , the number of scopes. A *scope* is a set of variables that are all subject to the same constraint.

- $\rho$ , the average degree of the variables:  $\rho = 2e/n$ . The degree of a variable  $V_i$  is the number of variables to which  $V_i$  is adjacent.
- $\delta$ , the domain cardinality. Every domain has the same cardinality  $\delta$ .
- $J$ , the number of  $(u, v)$  pairs such that  $P_{ij}(u, v) = \mathbf{true}$ .  $J$  is the same for each pair  $(i, j)$  of variables that are subject to a constraint. In other words, every scope has the same  $J$ .

Exponential time complexity implies that there is a region, within the space defined by these parameters, where algorithms are too slow to be serviceable. A central purpose of our experiments is to explore the extent of this region on a larger scale than has been reported previously. A further purpose is to compare performance of algorithms. With randomly generated instances we can observe and compare algorithm performance when parameters  $n$ ,  $\rho$ ,  $\delta$  and  $J$  are varied systematically, but we cannot explore the whole of this four-dimensional space. Instead we have chosen combinations of parameter values to illustrate their effect on the relative speed of algorithms.

We generate random instances by first choosing  $e = n\rho/2$  scopes randomly, except we make the degree of every variable be at least two, to prevent a short cut that saves time by processing acyclic subgraphs after cyclic parts of the constraint network have been satisfied [Sabin and Freuder 1997]. Moreover, we do not use any randomly chosen set of scopes that does not have a connected constraint graph. This prevents replacement of the constraint satisfaction by a collection of separate smaller problems.

After choosing scopes, our generator randomly chooses  $J$  pairs of values on each scope. No two pairs of values on the same scope are the same: duplicates are precluded. If  $R_{ij}$  is the set of randomly chosen tuples (i.e. pairs of values) on the scope  $(i, j)$  then  $P_{ij}(u, v) \equiv (u, v) \in R_{ij}$  and  $J = |R_{ij}|$ . Except for making the degree of every variable be at least two, our generator is the same as the Model B generator [Gent et al. 2001]. Constraints generated in this way are asymmetric:  $P_{i,j}(u, v) \neq P_{i,j}(v, u)$ .

For each of the randomly chosen scopes, the set of randomly chosen pairs of values on that scope is put directly into fast main memory (not disk) by the random generator. Timings reported below are initialized when this has been done. These timings, using a 3.2GHz Pentium 4 with 2Gbytes of RAM, include time for initialization of the bit matrices. In Sections 7.1.2 and 7.2 the search stops as soon as a solution is found. Constraints are randomly generated completely afresh for each trial. Many trials are required when variance of time taken is high.

In Section 7.1 algorithms are identified as follows, always without preprocessing, always with pop-stack restoration and always with duplicate representation of domains except in Tables II and III:

- BVDR:** Bit vector direct reduction, as in Sections 3.2 and B.2.
- BVCR:** Bit vector cumulative reduction, as in Sections 3.3 and B.2.
- BVFC:** Bit vector forward checking, as in Sections 3.4 and B.2.
- Focus:** Focus search, as in Section 4.
- AC2001:** As in [Bessière et al. 2005].

Tables II and III show that in some cases duplicate (i.e. bit-vector and array) representation<sup>15</sup> makes the search somewhat slower. In Section 7.1.2 we have only used duplicate representation because switching between duplicate and bit-vector-only representation would obscure intercomparison of results.

<sup>15</sup>Duplicate representation is intended to save time by avoiding visiting 0's in bit-vectors. In Table II, duplicate representation is slowest when the average number of 0's in bit-vectors is highest, which is the opposite of what we expected.

Table II. The *ratio* column shows the time taken using duplicate representation of domains divided by the time taken using only bit-vector representation. With duplicate representation, the time is the average over all the trials in Figures 13, 14, 15 and 16. With only bit-vector representation, the time is the average over the same number of trials with the same parameters. The column headed *nb0s* shows the average number of 0's in bit-vectors immediately before invocation of the domain reduction procedure. The *depth* is the average number of electively instantiated variables at this time.

			BVCR			BVDR	BVFC
$\rho$	$n$	$\delta$	ratio	nb0s	depth	ratio	ratio
3.5	100	300	1.294	104.71	2.02	1.373	-
7.0	100	100	0.713	34.58	2.36	0.836	0.915
7.0	100	300	1.336	92.74	1.68	1.431	1.125
7.0	300	100	0.978	22.71	2.38	1.059	1.096
7.0	300	300	1.380	65.52	1.77	1.549	0.986
14.0	300	100	1.003	21.47	2.05	1.088	1.011

Table III. Cumulative reduction with higher  $J$  than in Table II: average times for 1000 trials with duplicate representation of domains and also with only bit-vector representation (BVO). *sd* is the standard deviation of the average, not of the variate. Meanings of *nb0s* and *depth* are the same as in Table II.

				BVCR + Duplicate				BVO	
$\rho$	$n$	$\delta$	$J$	av	sd	nb0s	depth	av	sd
3.5	100	300	4600	4.60	1.91	224.5	13.4	6.65	0.75
7.0	100	100	3225	1.32	0.21	87.6	27.6	3.25	0.56
7.0	100	300	22900	2.12	0.41	280.5	35.9	3.19	0.73
7.0	300	100	3380	2.22	0.74	89.4	92.7	1.44	0.20
7.0	300	300	23600	4.99	1.65	275.4	108.2	4.01	1.10
14.0	300	100	6024	2.53	0.25	95.7	130.3	5.35	0.97

7.1.2 *Examples of time-growth curves.* As before,  $J$  is the number of  $(u, v)$  pairs such that  $P_{ij}(u, v) = \mathbf{true}$ ;  $\delta$  is the domain cardinality. When  $J$  is small the constraints are said to be *tight*. When  $J$  is large in comparison with its maximum value,  $\delta^2$ , the constraints are said to be *loose*. A problem that has very tight constraints is unlikely to have any solution, whereas a problem with very loose constraints is unlikely not to have any solution. Between these extremes there is a well known *crossover point* where the probability of existence of a solution is 0.5 and random problems require the largest search [Prosser 1996; Smith and Dyer 1996].

When  $J$  increases in (a) in Figures 13, 14, 15 and 16 there is exponential growth of time on the tight side of crossover, where there is no solution. When  $J$  increases above crossover, the number of solutions increases, so the time to find one decreases in (b) in these figures. Between (a) and (b) there is a gap, within the range of  $J$ , wherein exponential time complexity makes these algorithms impractical. In Table IV this gap increases with the average degree  $\rho$ , because when there are more constraints it is more difficult to find a solution on the loose side of crossover. The gap increases when *looseEnd* decreases.

Comparing Figure 13(a) with Figure 15(c), also Figure 14(a) with Figure 16(a), we see that, on the tight side of crossover, search becomes much faster when  $\rho$  is doubled and other parameters are unchanged. Search is faster because it is more

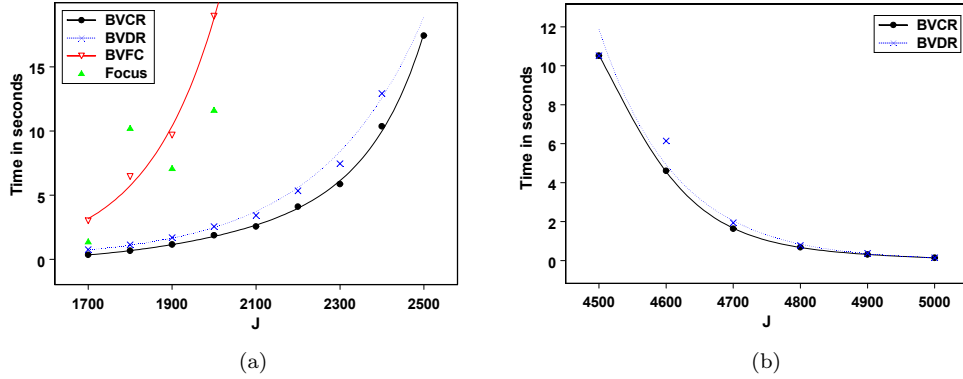


Fig. 13. Average times versus  $J$  in 1000 trials with average degree  $\rho = 3.5$ , number of variables  $n = 100$ , and domain cardinality  $\delta = 300$ . (a) Tight side of crossover. (b) Loose side of crossover.

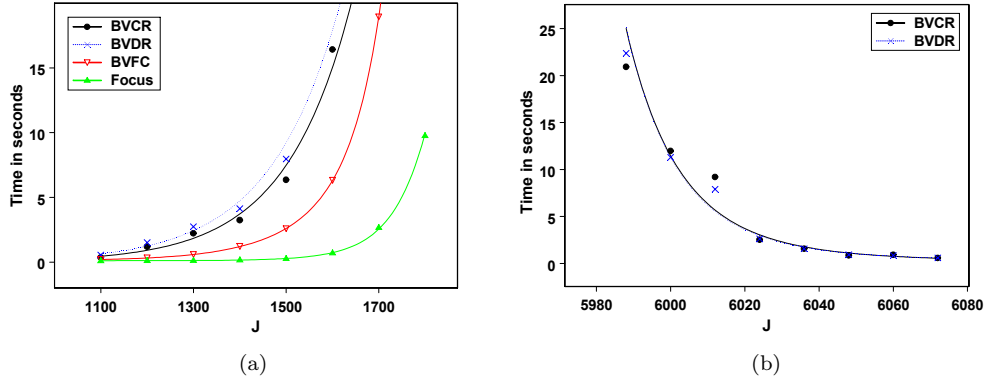


Fig. 14. Average times versus  $J$  with average degree  $\rho = 14$ , number of variables  $n = 300$  and domain cardinality  $\delta = 100$ . (a) 500 trials on the tight side of crossover. (b) 1000 trials on the loose side of crossover, where variance of average time is greater.

constrained<sup>16</sup>.

We also see that cumulative reduction is appreciably faster than direct reduction on the tight side, but there is no significant difference on the loose side of crossover. On the loose side, forward checking, focus search and AC2001 are always very much slower than cumulative reduction: Table V gives examples. On the tight side, AC2001 is always very much slower than direct reduction (BVDR), as would be expected from the wide-ranging large-scale experimental results of Lecoutre and Vion [2008]. Our subsequent experiments exclude AC2001 because there is no reason to expect it to be competitive with BVCR.

Focus search time depends on the number values to which a variable will be instantiated. This is the number of 1's in the result of bit-vector intersection at Line 6 in Fig 6. Focus search becomes more competitive when this number is reduced:

<sup>16</sup>When constraints are not binary, search is faster the greater the number of variables within each constraint [Ullmann 2007, Table 9].

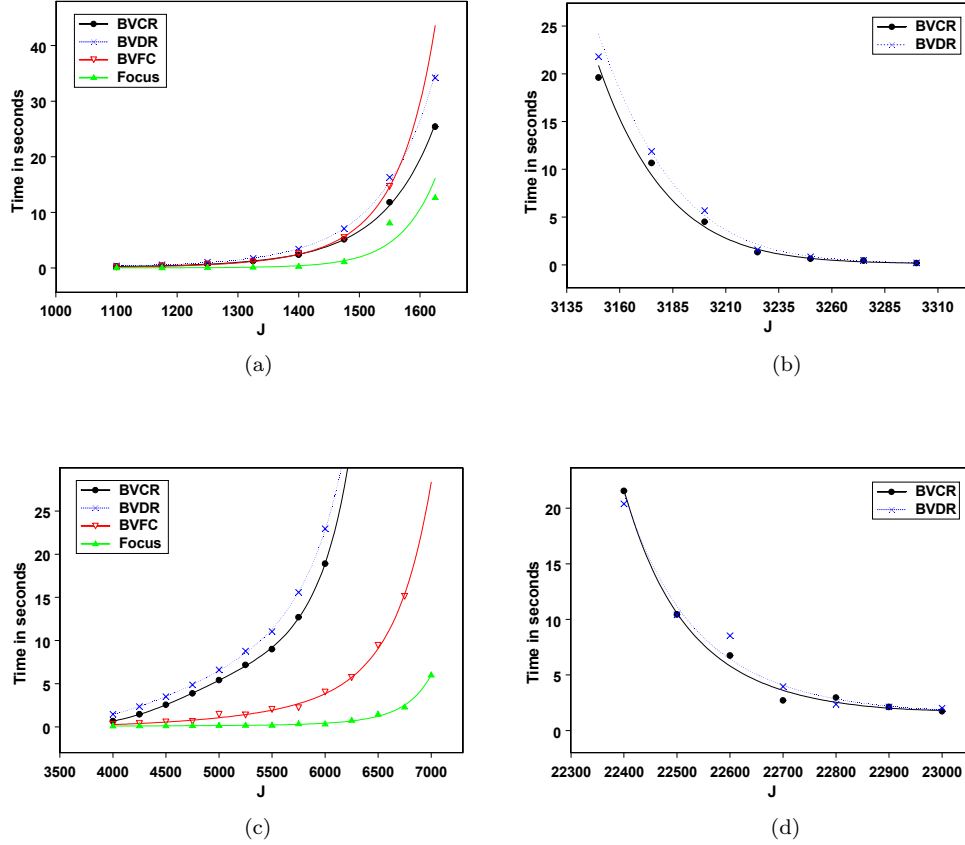


Fig. 15. Average times versus  $J$  with average degree  $\rho = 7$  and number of variables  $n = 100$ . (a)  $\delta = 100$ . 500 trials on the tight side of crossover. (b)  $\delta = 100$ . 1000 trials on the loose side of crossover. (c)  $\delta = 300$ . 500 trials on the tight side of crossover. (d)  $\delta = 300$ . 1000 trials on the loose side of crossover.

Table IV.  $J_T = J$  such that the average BVCR search time on the tight side of crossover is 10 seconds.  $J_L = J$  such that the average BVCR search time on the loose side of crossover is 10 seconds.  $\delta$  is the maximum cardinality of domains;  $\delta^2$  is the greatest possible value of  $J$ .  $\text{Gap} = (J_L - J_T)/\delta^2$ .  $\text{LooseEnd} = (\delta^2 - J_L)/\delta^2$ .  $\rho$  is the average degree;  $n$  is the number of variables.

Figure	$\rho$	$n$	$\delta$	gap	looseEnd
13(a)(b)	3.5	100	300	0.023	0.950
15(a)(b)	7.0	100	100	0.162	0.682
15(c)(d)	7.0	100	300	0.187	0.750
16(a)(b)	7.0	300	100	0.230	0.666
16(c)(d)	7.0	300	300	0.223	0.739
14(a)(b)	14.0	300	100	0.446	0.400

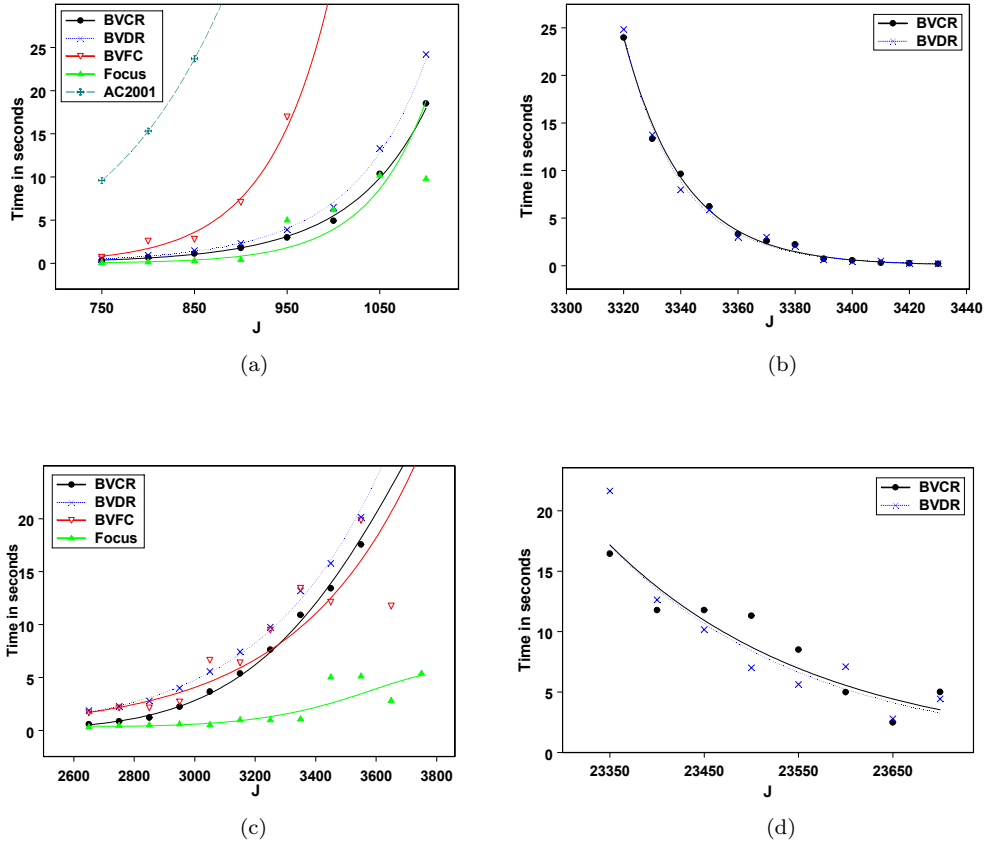


Fig. 16. Average times versus  $J$  with average degree  $\rho = 7$  and number of variables  $n = 300$ . (a) Domain cardinality  $\delta = 100$ . 500 trials on the tight side of crossover. (b)  $\delta = 100$ . 1000 trials on the loose side of crossover. (c)  $\delta = 300$ . 500 trials on the tight side of crossover. (d)  $\delta = 300$ . 1000 trials on the loose side of crossover.

Table V. Average time in seconds, and standard deviation of average time, on the loose side of crossover, with  $\rho = 7$ . Here ‘-’ signifies that the algorithm was too slow to permit sufficient trials. *NbOfTrials* is the number of trials.

$n$	$\delta$	$J$	Algorithm	Time Av	Time Sd	NbOfTrials
100	100	3300	BVCR	0.17	0.02	1000
100	100	3300	BVFC	4.15	0.94	100
100	100	3300	Focus	101.96	32.98	100
100	100	3300	AC2001	12.42	0.52	100
100	300	23000	BVCR	1.72	0.41	1000
100	300	23000	BVFC	8.72	2.19	500
100	300	23000	Focus	71.61	31.76	100
100	300	23000	AC2001	343.10	66.61	100
300	100	3430	BVCR	0.20	0.01	1000
300	100	3430	BVFC	-	-	-
300	100	3430	Focus	-	-	-
300	100	3430	AC2001	188.96	71.13	50
300	300	23700	BVCR	5.01	2.05	1000
300	300	23700	BVFC	767.79	268.51	10
300	300	23700	Focus	-	-	-
300	300	23700	AC2001	135.20	7.74	50

*By increasing the average degree of variables.* This increases the number of bit-vectors that are intersected. Focus search is faster in Figure 15(c), where average degree  $\rho = 7$ , than in Figure 13(a), where  $\rho = 3.5$ . Again, focus search is faster in Figure 14, where  $\rho = 14$ , than in Figure 16(a), where  $\rho = 7.0$ , other parameters being the same.

*By increasing  $\delta$  whilst  $J$  remains unchanged.* This reduces the density of 1's in bit-matrix rows. Focus search is faster in Figure 15(c), where  $\delta = 300$ , than in Figure 15(a), where  $\delta = 100$ . Again, focus search is faster in Figure 16(c), where  $\delta = 300$ , than in Figure 16(a), where  $\delta = 100$ , other parameters being the same.

*By decreasing the average cardinality of domains.* Sections 7.6.4 and 7.7 provide examples.

It is not so easy to explain why focus search is less competitive in Figure 16(a), where  $n = 300$ , than in Figure 15(a), where  $n = 100$ .

## 7.2 Experiments with Frequency assignment problems

Binary constraint satisfaction problems arise in the assignment of frequency and polarization to radio links so as to satisfy constraints such as minimization of interference [Aardal et al. 2007]. A collection of *fapp* (i.e. frequency assignment problem with polarization) problem instances are available at <http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/benchmarks.html> for use as benchmarks in the experimental evaluation of constraint satisfaction algorithms; physical requirements have been translated into clear ready-made constraint satisfaction problem formulations. An XML file includes, for each instance, a specification of the domains of variables, a specification of constraint scopes, and a specification of one or more numeric predicates that must be satisfied on each scope. An example of a numeric predicate is

$$P_{ij}(u, v) = (u \times v < a) \vee (|u - v| \geq b)$$

where  $a$  and  $b$  are prescribed constants.

These *fapp* benchmarks consist of forty sets, *fapp01*, *fapp02*, ..., *fapp40*, each containing eleven instances. Within each set, the eleven instances all have the same number,  $n$ , of variables, the same number,  $e$ , of scopes, and, for each domain, the same *upperBound* – *lowerBound*. In fact *upperBound* – *lowerBound* is so big that bit-vector algorithms are practical only with surrogate values, as in Section 5. For example, for *fapp01*, the greatest range of values in any domain is 5312, whereas  $\delta = 190$ . Another example is that for *fapp20*, the greatest range of values in any domain is 5640, whereas  $\delta = 302$ . Within each set, all of the eleven instances have the same value of  $\delta$ .

In these *fapp* benchmarks there are many cases where, for a given scope, two specified predicates must both be satisfied. If these two predicates for the scope  $(i, j)$  are  $P_{ij}^1$  and  $P_{ij}^2$  then bit-matrices  $M_j^i$  and  $M_i^j$ , initially empty, can be constructed by the routine:

```

for each surrogate value  $u$  of  $V_i$  do
  for each surrogate value  $v$  of  $V_j$  do
    if  $P_{ij}^1(S_i(u), S_j(v)) \wedge P_{ij}^2(S_i(u), S_j(v))$  then
      include  $v$  in  $M_j^i[u]$ ; include  $u$  in  $M_i^j[v]$ ;
    end if
  end for
end for

```

In Table VI, differences in average timings for BVDR, BVCR and IRCR may not be significant since there are only eleven instances in each of the 40 sets, so we have small-sample statistics. Because incremental restoration requires less memory,



Table VI. Parameters and timings on  $40 \times 11$  *fapp* instances, using duplicate representation of domains, not using preprocessing, and stopping when one solution is found. BVDR and BVCR use pop-stack domain restoration; IRCR differs from BVCR only in that it uses incremental restoration. If the search takes more than 1000 seconds it is stopped and this situation is a *time-out*; ‘to’ denotes the number of time-outs; *time* is the average search time. This is the average over only those instances for which there is no time-out. ‘-’ signifies insufficient memory.

fapp	<i>n</i>	<i>e</i>	$\delta$	Set-up time	BVDR		BVCR		IRCR	
					time	to	time	to	time	to
01	200	1108	190	8.71	0.283	0	0.225	0	0.222	0
02	250	1636	210	16.02	0.345	0	0.277	0	0.283	0
03	300	2327	250	35.35	0.244	0	0.254	0	0.254	0
04	300	1799	270	35.49	4.045	0	4.911	0	4.961	0
05	350	2488	270	59.46	0.792	0	0.772	0	0.774	0
06	500	3478	290	78.93	7.142	0	7.618	0	7.618	0
07	600	4777	302	79.49	0.410	0	0.346	0	0.348	0
08	700	3834	282	71.38	1.264	0	1.535	0	1.371	0
09	800	4800	350	61.08	0.634	0	1.097	0	1.078	0
10	900	6071	362	164.21	2.541	0	1.944	0	1.940	0
11	1000	8005	362	319.38	2.335	0	42.780	0	42.700	0
12	1500	13439	310	506.64	19.845	6	59.992	5	59.527	5
13	2000	13699	190	156.47	4.507	4	4.476	4	4.538	4
14	2500	21610	362	942.23	-	-	-	-	143.328	6
15	3000	17754	182	218.08	13.309	1	1.603	2	1.595	2
16	260	2088	302	29.96	0.007	0	0.003	0	0.012	0
17	300	2056	302	40.24	0.014	0	0.010	0	0.017	0
18	350	2387	302	34.93	0.012	0	0.011	0	0.021	0
19	350	3114	802	515.94	0.149	0	0.127	0	0.114	0
20	420	2487	302	51.03	0.009	0	0.009	0	0.011	0
21	500	1589	242	30.31	0.056	0	0.046	0	0.108	0
22	1750	16924	802	-	-	-	-	-	-	-
23	1800	33337	302	859.99	0.048	0	0.059	0	0.049	0
24	2000	14301	302	453.17	0.219	0	0.189	0	0.184	0
25	2230	11974	302	266.76	0.193	0	0.154	0	0.155	0
26	2300	12761	302	386.77	0.335	0	0.334	0	0.255	0
27	2550	6231	242	117.77	0.753	0	0.636	0	0.632	0
28	2800	12046	998	-	-	-	-	-	-	-
29	2900	41781	998	-	-	-	-	-	-	-
30	3000	33301	778	-	-	-	-	-	-	-
31	400	1644	700	280.97	-	-	-	-	1.060	0
32	550	5017	998	-	-	-	-	-	-	-
33	650	4631	498	288.29	-	-	-	-	0.142	0
34	750	4623	998	-	-	-	-	-	-	-
35	1500	11723	698	-	-	-	-	-	-	-
36	2000	10067	454	517.94	-	-	-	-	0.445	0
37	2250	22553	998	-	-	-	-	-	-	-
38	2500	32622	698	-	-	-	-	-	-	-
39	2750	12605	502	1,115.05	-	-	-	-	2.829	0
40	3000	28313	698	-	-	-	-	-	-	-

IRCR processes 55 more *fapp* instances than BVCR and BVDR. Another comment is that *Set-up time* is almost entirely the average time to construct bit-matrices. Although in most cases the set-up time very greatly exceeds the search-time, the use of bit-matrices may nevertheless be sensible because set-up time is  $O(e\delta^2)$  whereas a search algorithm has worst-case exponential time complexity. For the *fapp* benchmarks, timings obtained by other workers can be seen at <http://www.cril.univ-artois.fr/CPAI06/round2/results/globalbybench.php?idev=6&idcat=38&idSubCat=56>.

Table VII shows average search time and numbers of time-outs for forward checking (without preprocessing) on *fapp*01-10. Here forward checking is certainly very much slower than BVCR. With focus search (without preprocessing) there is time-out on all of these 110 instances, so focus search is certainly very much slower than forward checking. Very poor timings for focus search and forward checking can

Table VII. Average forward checking search times on *fapp01-10*.  $K$  is the average number of 1's in bit-matrices; higher  $K$  signifies looser constraints.

fapp	time	to	$K$
01	18.73	4	11,163
02	5.20	8	14,005
03	9.71	3	22,734
04	3.72	8	28,876
05	108.76	6	34,655
06	0.78	7	34,166
07	101.72	3	24,998
08	0.01	8	27,992
09	186.14	7	18,795
10	5.22	5	40,803

probably be attributed to the looseness of the constraints, which was unhelpful to these procedures on the loose side of crossover in Section 7.1.

### 7.3 Graph versions of algorithms

The remainder of Section 7 deals with isomorphism, for which search is always preceded by preprocessing. Algorithms are henceforward identified as follows:

- BVCR:** Bit vector cumulative reduction, as in Section 6.3.1,
- BVFC:** Bit vector forward checking, as in Section 6.3.2,
- Focus:** Focus search, as in Section 6.3.3,
- Preprocessing:** As in Section 6.3.1.
- ADNP:** This is a variant of BVCR in which the *allDifferent* constraint is not propagated<sup>17</sup>. ADNP stands for *allDifferent not propagated*.

Duplicate (i.e. bit-vector and array) representation of domains is used because in Table VIII this is usually faster than representing domains only by bit-vectors. More importantly, duplicate representation allows incremental restoration, which enables experimentation on a larger scale than with pop-stack restoration. For this reason, incremental restoration is always used in Sections 7.4.1, 7.5 and 7.6. Pop-stack restoration would have been practicable in some cases, but switching between pop-stack and incremental restoration would have obscured the inter-comparison of results.

### 7.4 Experiments with graph automorphism

**7.4.1 Random graphs.** An *automorphism* is an isomorphism of a graph with itself. Although determination of automorphism of random graphs is nowadays a too-easy test of an isomorphism algorithm, Figure 17 shows some timings for comparison with [Foggia et al. 2001; Cordella et al. 2004]. Here  $G^\beta$  is the same graph as  $G^\alpha$ , which is connected and randomly-generated except that no vertex has degree one, for reasons briefly indicated in Section 7.1.1. Figure 17 shows results obtained using three different vertex invariants:

<sup>17</sup>ADNP uses a set *available* of values that are currently absent from single-valued domains. ADNP instantiates a variable  $i$  to a value  $u$  only if  $u \in \text{available}$ . When a variable  $i$  is instantiated to a value  $u$ , then  $u$  is removed from *available*, following Ullmann [1976]. Moreover, when a domain is reduced to a single value,  $v$ , then  $v$  is removed from *available* unless  $v$  is already absent from *available*, in which case procedure *reduce* returns *consistent* = **false**. If, when at least one domain is multivalued, procedure *choose* finds no multivalued domain that includes at least one value in *available* then the effect is the same as when procedure *reduce* returns *consistent* = **false**. The set *available* is subject to pop-stack restoration.

Table VIII. Average times for finding all subgraph isomorphisms in 10,000 trials with duplicate representation of domains and with only bit-vector representation (BVO). In these experiments with graphs randomly generated as in Section 7.6.1,  $n_\alpha = 500$ ,  $\eta$  is the number of distinct vertex and edge labels, and the column headed *directed* indicates whether graphs are directed. The column headed *isos* shows average numbers of subgraph isomorphisms. Here *sd* is the standard deviation of the average, not of the variate.

$n_\beta$	$\rho$	$\eta$	directed	isos	Duplicate		BVO	
					av	sd	av	sd
550	3	1	no	41,272.8	0.6776	0.0784	1.3267	0.1616
625	6	1	no	57,759.8	1.1025	0.1060	1.2193	0.1427
625	9	1	no	6.4	0.8873	0.0107	0.6801	0.0791
625	3	2	no	29,963.7	0.5616	0.0570	0.7015	0.0846
725	3	3	no	17,909.3	0.3292	0.0596	0.5619	0.0879
750	3	3	no	47,807.1	1.0445	0.0866	1.4307	0.1247
1000	3	4	no	16,691.4	0.4915	0.0515	0.6346	0.0622
575	3	1	yes	16,378.8	0.2701	0.0401	0.4066	0.0500
625	6	1	yes	255.9	0.0226	0.0002	0.0327	0.0003
675	6	1	yes	100,591.8	1.5456	0.1058	1.9650	0.2131
800	9	1	yes	69,377.0	1.1395	0.1650	1.3486	0.1984

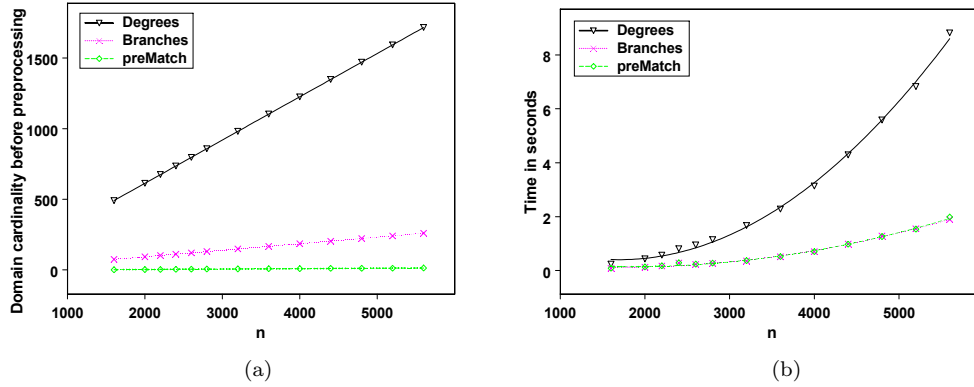


Fig. 17. Results of 100 trials, with algorithm BVCR with preprocessing and  $\rho = 3$  which is the average degree. Here  $n$  is the number of vertices in the graph. (a) Average cardinality of domains before commencement of preprocessing. (b) Average time to find all automorphisms.

- degrees* a value  $u$  is initially included in domain  $D_i$  if  $\text{degreeOf}(V_u^\beta) = \text{degreeOf}(V_i^\alpha)$ .
- branches* a value  $u$  is initially included in domain  $D_i$  if the sum of degrees of vertices adjacent to  $V_u^\beta$  equals the sum of degrees of vertices adjacent to  $V_i^\alpha$ . Results obtained using *branches* do not differ significantly from results obtained using distance-2-degrees instead of branches.
- prematches* a value  $u$  is initially included in domain  $D_i$  if  $\text{prematches}(i, u)$  returns **true**, where *prematches* is as in Figure 8 except that the two instances of ' $>$ ' are both changed to ' $\neq$ '.

For each different number of vertices in Figure 17:

- The average number of automorphisms is less than 1.41.
- The average cardinality of domains after preprocessing is less than 1.0003.
- The average of the total number of elective instantiations during the search is less than 1.53

Table IX. Details of ten sets of strongly regular graphs. *nbGraphs* is the number of graphs in a set; *nbAutomorphisms* is the average number of automorphisms of graphs in this set.

Row	$n$	$\rho$	$\lambda$	$\mu$	nbGraphs	nbAutomorphisms
1	25	8	3	2	1	28,800
2	25	12	5	6	15	52.27
3	26	10	3	4	10	18.9
4	27	10	1	5	1	51,840
5	28	12	6	4	4	10,290
6	29	14	6	7	41	11.85
7	35	16	6	8	3854	14.05
8	35	18	9	9	227	189.95
9	36	14	4	6	180	78.92
10	40	12	2	4	28	3,782.1

Table X. Experimental results in Row  $x$  are obtained with the set of graphs in Row  $x$  of Table IX. *time* is the average total time in seconds to find all automorphisms; *nodes* is the average number of search-tree nodes visited, which is the same thing as the average number of elective instantiations during the search. BVCR and BVFC here use pop-stack domain restoration.

Row	BVCR		BVFC		Focus	
	time	nodes	time	nodes	time	nodes
1	0.210	50,025	0.300	99,225	0.060	358,425
2	0.014	2,412	0.027	5,947	0.002	7,694
3	0.006	701	0.011	3,036	0.001	4,298
4	0.440	88,857	0.950	298,647	0.200	875,367
5	0.255	24,645	0.360	99,511	0.077	267,004
6	0.032	3,213	0.056	11,188	0.004	12,872
7	0.086	4,765	1.012	14,385	0.007	16,413
8	1.685	94,862	1.352	197,079	0.069	184,510
9	0.034	1,655	0.068	13,126	0.007	15,657
10	0.134	7,391	0.269	40,288	0.060	134,611

This means that there is almost no search, and times in Figure 17(b) are almost entirely times taken by preprocessing. The curve-fit curves in Figure 17(a) are exactly linear; in (b) they are gently quadratic. Because there is almost no search, there is no point in comparing the performance of different search algorithms such as BVCR, forward checking and focus search on randomly generated graphs.

**7.4.2 Strongly regular graphs.** A graph  $G = (V, E)$  that has  $n$  vertices, all of these having the same degree  $\rho$ , is said to be *strongly regular* if there are integers  $\lambda$  and  $\nu$  such that:

- Every two adjacent vertices have  $\lambda$  common neighbors, and
- Every two non-adjacent vertices have  $\nu$  common neighbors.

Finding automorphisms of strongly regular graphs can be a hard test for a vertex partitioning algorithm. In the present context, strongly regular graphs are of interest because preprocessing, as in Section 6.1, does not reduce any domain and is therefore useless; there is no point in comparing performance using unary constraints derived from degrees or branches or prematching, because none of these reduce any domain. Every domain is initialized to contain exactly  $n$  values.

Each row in Table IX provides information about one of ten sets of strongly regular graphs obtained from <http://cs.anu.edu.au/~bdm/data/graphs.html>. Within a set, all the graphs belong to the same class, denoted by  $srg(n, \rho, \lambda, \nu)$ . Focus search is fastest in all rows in Table X, which shows timings. In Section 7.1, forward checking and focus search work best with high  $\rho$ . Forward checking is faster than

Table XI. Average time in seconds for BVCR, ADNP, BVFC and focus search on benchmark instances of Solnon [2010].  $\rho_\alpha$  and  $\rho_\beta$  are average degrees of  $G^\alpha$  and  $G^\beta$ ;  $n_\beta$  is the number of vertices in  $G^\beta$ . Each line in this table shows averages over 90 instances, except for the random instances, where averages are over 30 instances, and *scalefree*, where averages are over 20 instances. Moreover each line shows average times for finding one isomorphism, except for the bottom three lines, which show average times for finding all isomorphisms. Algorithms are implemented in Modula-2. There are no time-outs except for two with focus search with *random* 0.10 and  $n_\beta = 600$ . A hyphen signifies that an algorithm is excessively slow.

benchmark	$n_\beta$	$\rho_\alpha$	$\rho_\beta$	BVCR	ADNP	BVFC	Focus
fixed valence	200	5.67	6.00	0.050	0.047	0.008	0.130
	400	5.83	6.00	0.412	0.689	0.056	0.173
	800	5.91	6.00	4.795	8.603	0.390	1.723
bounded valence	200	4.12	5.98	0.002	0.002	0.002	0.005
	400	4.18	5.90	0.006	0.009	0.128	0.037
	800	4.27	5.99	0.026	0.036	0.105	0.045
4D irregular meshes	256	4.11	6.04	0.030	0.044	0.026	0.016
	625	4.28	6.23	0.083	0.267	1.104	0.133
	1296	4.36	6.35	2.699	3.521	2.566	6.028
random 0.01	200	2.82	4.45	0.018	0.037	0.022	0.096
	400	4.03	8.18	10.000	10.450	63.291	4.010
	600	5.42	12.08	72.380	184.339	63.891	2.795
random 0.05	200	7.97	19.47	6.039	30.117	0.841	0.263
	400	15.60	38.94	-	-	14.706	1.976
	600	23.41	58.43	-	-	89.447	10.333
random 0.10	200	15.08	37.85	-	-	8.862	1.435
	400	25.17	50.55	-	-	-	89.278
	600	45.48	113.82	-	-	-	209.295
scale free	200	5.43	6.03	0.015	0.018	0.016	0.017
	600	5.43	6.02	0.268	0.268	0.267	0.238
	1000	5.44	6.02	1.129	1.078	1.103	1.027

BVCR only in Row 8 of Table X, where  $\rho = 18$ . In Row 8, the time taken by focus search is less than the time taken by BVCR by a factor of 24; this is the only row where the number of nodes visited by focus search is less than the number of nodes visited by forward checking. Although focus search visits the largest number of nodes in all other rows, it is fast because it does less work after each elective instantiation.

## 7.5 Experiments with subgraph isomorphism benchmarks

Solnon [2010] has reported experimental comparison of recent subgraph isomorphism algorithms using benchmark pairs of graphs  $(G^\alpha, G^\beta)$  available at <http://www710.univ-lyon1.fr/~csolnon/benchmarks.tgz>. Timings in Table XI can be compared with Solnon's timings for other algorithms on the same benchmarks. Comparison is of course hazardous when algorithms are implemented in different programming languages by different programmers and when timings are obtained using different processors. It would be more meaningful to compare time-growth curves, as we have done for BVCR and focus search in Section 7.1.2, but appropriate time-growth curves have not yet been published for competitor algorithms.

In twenty trials with each of the *scalefree* benchmarks with  $n_\beta = 200, 600, 1000$ , an isomorphism was found respectively in 9, 10 and 12 trials by preprocessing without any search. This is one reason why timings with these benchmarks are similar for the four algorithms. As in Section 7.1.2, focus search becomes more competitive the higher the average degree of  $G^\alpha$ . This is particularly clear with the random benchmarks.

ADNP is a variant of BVCR that was introduced in Section 7.3. Results for ADNP are included to help assess the extent to which propagation of the *allDifferent* constraint in BVCR is actually beneficial. In Table XI the differences in times

for BVCR and ADNP are not very significant, except where the average time for BVCR exceeds two seconds, but BVCR is very much faster than ADNP in Figure 18. Results of small-scale experimentation, as in Table XI, may be misleading.

## 7.6 Experiments with subgraph isomorphism of randomly generated graphs

**7.6.1 Experimental framework.** We aim to show how search time and number of isomorphisms depend on the sizes of the two graphs. We work with randomly generated graphs to facilitate systematic variation of problem parameters. Our procedure is as follows.

A graph  $G^\beta$ , having  $n_\beta$  vertices, is generated randomly, except that no vertex has degree less than two. A subset of  $n_\alpha$  vertices of  $G^\beta$  are selected randomly to be the vertices of a new smaller graph  $G^\alpha$ . Every pair of these vertices that are adjacent in  $G^\beta$  are also adjacent in  $G^\alpha$ . There are no other edges in  $G^\alpha$ , except that if  $G^\alpha$  is found not to be connected then, until  $G^\alpha$  becomes connected, a succession of new edges are inserted into  $G^\alpha$  and  $G^\beta$  so as to preserve subgraph isomorphism. In most cases very few new edges are inserted. In Section 7.6,  $\rho$  denotes the average degree of graph  $G^\beta$  before insertion of extra edges to ensure that  $G^\alpha$  is connected.

In experiments with randomly generated labelled graphs, each vertex in  $G^\beta$  is labelled with an ordinal chosen randomly within the range  $1, \dots, \eta$ . Each edge in  $G^\beta$  is also labelled with an ordinal chosen randomly within the same range. Of course different ranges could be used for vertices and edges, but for simplicity we use equal ranges. Labels of vertices and edges in  $G^\alpha$  are the same as those of the vertices and edges in  $G^\beta$  to which they are mapped by the isomorphism inherent in the construction.

In all experiments reported in Section 7.6, the search is preceded by BVCR pre-processing, which is preceded by prematching. Prematching means applying unary constraints by invoking the appropriate version of *prematches*.

**7.6.2 Unlabelled undirected subgraph isomorphism.** As before,  $n_\alpha$  and  $n_\beta$  are the numbers of vertices in graphs  $G^\alpha$  and  $G^\beta$ , respectively. When  $n_\beta = n_\alpha$ , finding graph (not sub-graph) isomorphism is not NP-Complete; instead, we have relatively gentle time-growth, as in Figure 17(b). If we start with  $n_\alpha$  much less than  $n_\beta$  and increase  $n_\alpha$  progressively until  $n_\beta = n_\alpha$ , the time to find one isomorphism at first increases exponentially, as in Section 7.1.2 and then decreases as we approach  $n_\beta = n_\alpha$ . The maximum in Figure 18 has not, so far as we know, been reported previously. Figure 18 also shows that ADNP, a variant of BVCR introduced in Section 7.3, is usually much slower than BVCR. This suggests that propagation of *allDifferent*, as in Section 6.3.1, substantially increases the speed of BVCR.

If we fix  $n_\alpha$  and progressively increase  $n_\beta$ , starting near  $n_\alpha = n_\beta$ , the time to find one isomorphism at first increases exponentially, in Figure 19, until it reaches a further maximum, which again, so far as we know, has not been reported previously. When  $n_\beta$  is increased beyond this maximum there are more edges in  $G^\beta$  that could correspond to a given edge in  $G^\alpha$ . Because there are more solutions, finding one of these takes less time. Near the maximum the variance of timings is particularly high; the search is quick in most cases and very slow in others<sup>18</sup>. Despite the upward slope of the average-time plot in Figure 19(b), the number of time-outs tends to decrease as  $n_\beta$  increases. Moreover, the average time when  $n_\beta > 150$  in Figure 19(a) exceeds all average times in Figure 19(b).

**7.6.3 Unlabelled directed subgraph isomorphism.** With unlabelled directed graphs, and with the same parameters as in Figure 19, we see no maximum in Figure 20(a).

<sup>18</sup>which is *heavy-tailed* behavior [Gomes et al. 2000; Hulubei and O’Sullivan 2006]. This also occurs at crossover in Section 7.1.2.

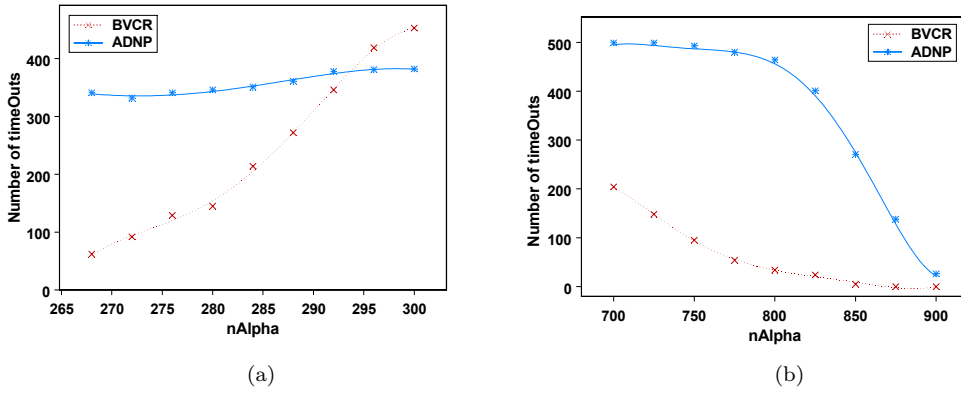


Fig. 18. The number of timeOuts when seeking *one* isomorphism in 500 trials with  $n_\beta = 1000$  and average degree  $\rho = 3$ . Here a *timeOut* is a trial in which the search is stopped at 10 seconds. (a) Tight side of maximum. (b) Loose side of maximum.

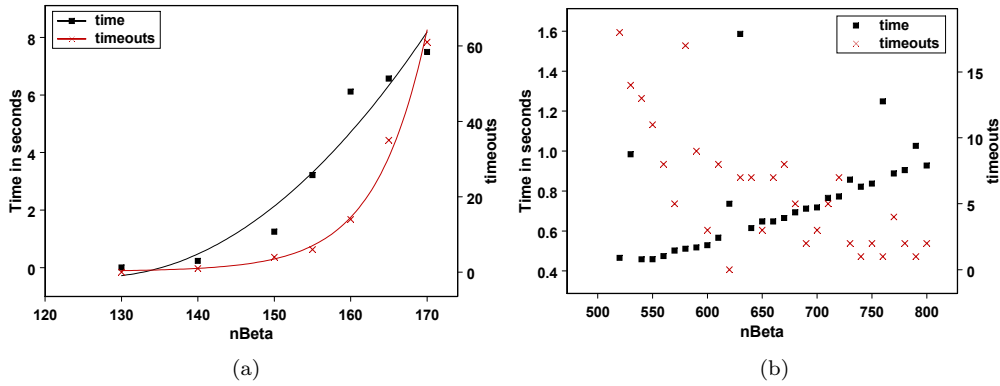


Fig. 19. Average time to find *one* isomorphism (Left-hand axis) and number of time-outs (Right-hand axis) with BVCR,  $n_\alpha = 100$ , average degree  $\rho = 3$  and 1000 trials. Here a *timeOut* is a trial in which the search is stopped at 1000 seconds; *time* is the average over trials where there is no time-out. (a) Tight side of maximum. (b) Loose side of maximum.

The increasing difference between total time<sup>19</sup> and search time is almost entirely due to preprocessing, which removes an increasing number of values from domains, as can be seen in Figure 20(b).

**7.6.4 Labelled undirected subgraph isomorphism.** When *all* isomorphisms are found, the sharp peak in Figure 21(a) is unexpected and unexplained. Again with the same parameters as in Figures 19 and 20, Figure 21(b) shows average total time to find one isomorphism. Most of this time is taken by preprocessing<sup>20</sup>, which is the same for Figures 21(a) and (b), and this is why the difference in total times is slight.

<sup>19</sup>In 1000 trials there is exactly one time-out at 1000 seconds when  $n_\alpha = 1000, 1200, 1400$  and this is not included in the average over the remaining 999 trials in Figure 20(a). There are no time-outs with other values of  $n_\alpha$ .

<sup>20</sup>When  $n_\beta = 5, 600$ , the average cardinalities of domains before and after preprocessing are 653.2 and 1.632 respectively.

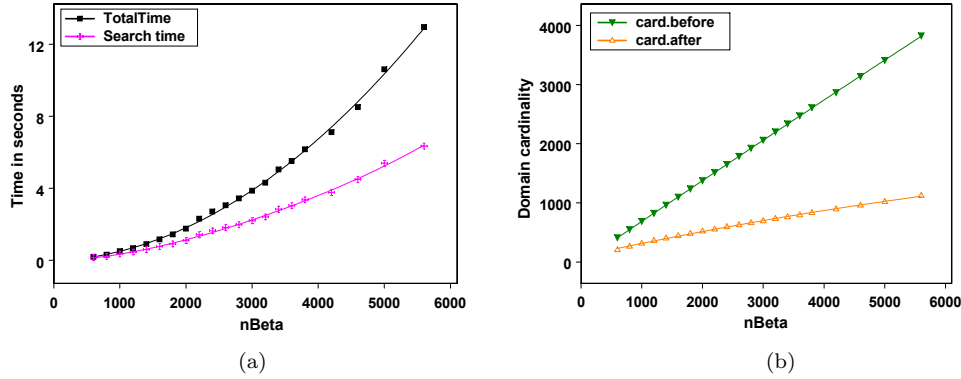


Fig. 20. Results of 1000 trials with unlabelled directed subgraph isomorphism, BVCR,  $n_\alpha = 100$ , and average degree  $\rho = 3$ . (a) Average total time and search time to find *one* isomorphism. (b) Domain cardinalities before and after preprocessing.

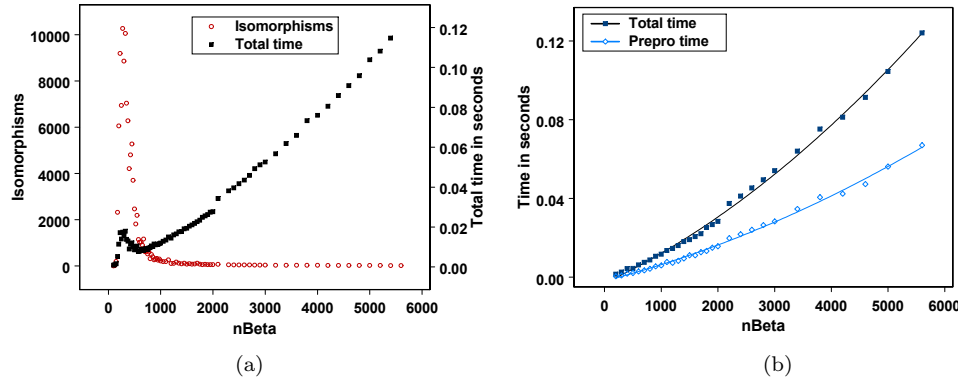


Fig. 21. Results of 500 trials with labelled undirected subgraph isomorphism with BVCR,  $n_\alpha = 100$ , average degree  $\rho = 3$  and with  $\eta = 2$  which means that there are two possible labels for each vertex and for each edge. (a) Number of isomorphisms (left-hand axis) and total time to find all of them (right-hand axis). (b) Average total time and preprocessing time to find only *one* isomorphism.

When we fix  $n_\alpha = 200$  and increase  $n_\beta$ , the number of isomorphisms increases so rapidly that it is not practical to observe the top of the peak. Instead, Figure 22(a) shows the number of isomorphisms and the time to find all of them, up to  $n_\beta = 380$ . Figure 22(b) continues these results for  $n_\beta \geq 1000$ . Focus search is competitive in these experiments because of the very low cardinality<sup>21</sup> of domains after preprocessing, as can be seen in Figure 23(a), right-hand axis. We see no peak in total time to find one isomorphism, left-hand axis. When  $\eta = 4$  and all isomorphisms are found, the peak is so small that we can see the whole of it in Figure 23(b).

With  $n_\alpha = 1000$  and  $\eta = 8$  labels, the slight peak in the number of isomorphisms

<sup>21</sup>Very low cardinality means that the amount of data moved by incremental restoration is exceptionally low, so incremental restoration is substantially faster than pop-stack restoration in this case, although in most other cases incremental restoration is appreciably slower.



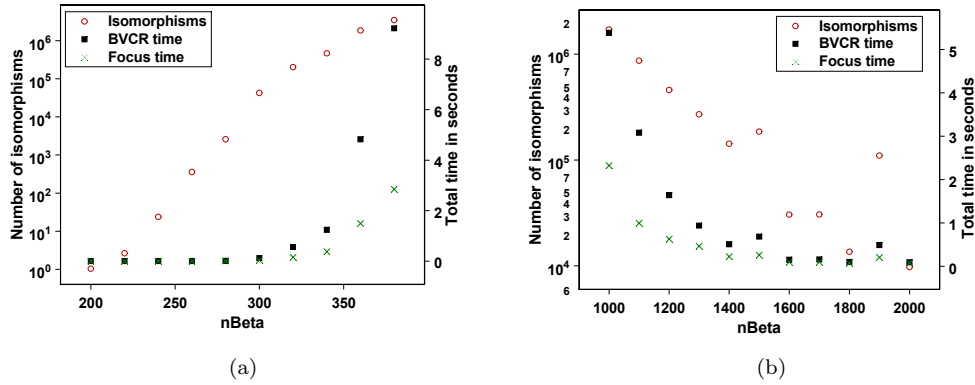


Fig. 22. Five hundred trials with labelled undirected subgraph isomorphism, BVCR and focus search,  $n_{\alpha} = 200$ , average degree  $\rho = 3$  and  $\eta = 2$  labels. The number of isomorphisms is plotted against the left axis, which is logarithmic. The total time, in seconds, to find all isomorphisms is plotted against the right axis, which is linear. (a)  $n_{\beta} \leq 380$ . (b)  $n_{\beta} \geq 1000$ .

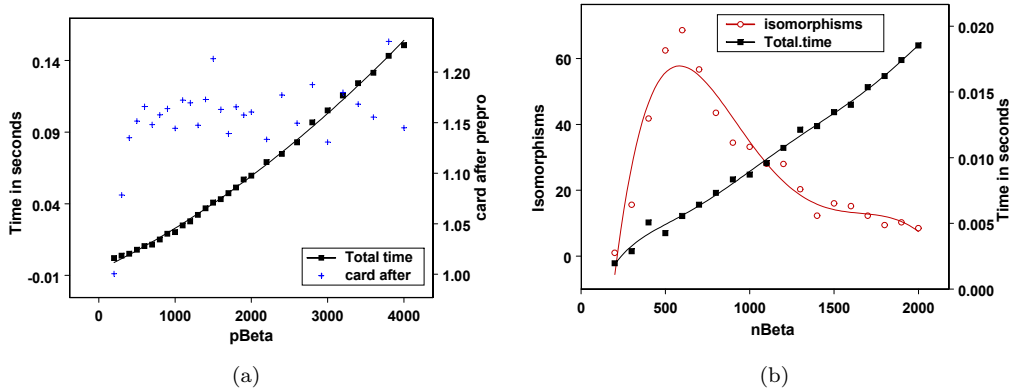


Fig. 23. Labelled undirected subgraph isomorphism with BVCR,  $n_{\alpha} = 200$  and average degree  $\rho = 3$ . (a) With  $\eta = 2$  labels, one hundred trials wherein the search stops as soon as the first isomorphism is found. (b) With  $\eta = 4$  labels, five hundred trials wherein all isomorphisms are found.

in Figure 24(a) may perhaps be due to the slight peak in average domain cardinality after preprocessing. This cardinality, which is always less than 1.01, is plotted against the right hand axis in Figure 24(b). Total time to find all isomorphisms, in Figure 24(a), is only very slightly greater than the time to find just one isomorphism in Figure 24(b). This is because when  $n_{\alpha} = 1000$  the total time is predominantly prematching time, as can be seen in Figure 24(b). Prematching checks  $n_{\alpha} \times n_{\beta}$  possible correspondences, taking significant time when values of  $n_{\alpha}$  and  $n_{\beta}$  are high. The discontinuity in total-time curves at  $n_{\beta} = 2000$  is presumably due to the visible discontinuity in the prematching time curve, for which we have no explanation.

## 7.7 Molecular graph retrieval experiments

The structural formula of a molecule can be regarded as a graph, in which vertices are labelled with atom-identifiers, such as C, O, N, and edges are labelled with

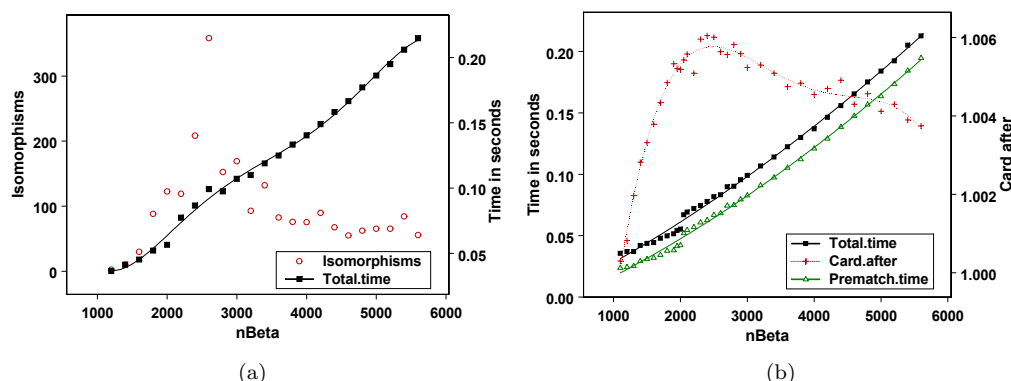


Fig. 24. Results of trials with labelled undirected subgraph isomorphism with BVCR,  $n_{\alpha} = 1000$ , average degree  $\rho = 3$  and  $\eta = 8$  labels. (a) Five hundred trials wherein all isomorphisms are found. (b) One hundred trials wherein the search stops as soon as the first isomorphism is found.

Table XII. Numbers of atoms in intervals 41 through 52. Within Interval 52, the number of atoms is distributed decreasingly across the range 84..382; the lower end of this range is very much more densely populated than the higher end.

Interval	Min number of atoms	Max number of atoms	Average number of atoms	Average number of bonds
41	41	41	41.00	44.82
42	42	42	42.00	45.66
43	43	44	43.51	47.09
44	45	46	45.50	49.52
45	47	48	47.50	51.52
46	49	51	50.06	54.53
47	52	54	52.92	57.70
48	55	58	56.46	61.37
49	59	63	60.97	66.26
50	64	70	66.62	71.88
51	71	83	76.22	81.80
52	84	382	118.33	133.77

bond-types<sup>22</sup>, which may be 1,2,3 or 4. In the following experiments, queries and targets are graphs representing molecules; we seek all targets to which a query is exactly subgraph isomorphic. These experiments are intended to show how the effectiveness of trilabel matching, prematching and preprocessing depend on the numbers of atoms in query and target molecules, using molecular data freely available to the public<sup>23</sup>. This data is of a kind for which there are well-established practical requirements for determination of subgraph isomorphism.

We have sorted this data-set into intervals, each containing at least 1000 molecules. For  $6 < n < 43$ , every molecule in interval  $n$  has exactly  $n$  atoms. For  $n > 41$ , Table XII shows the range of numbers of atoms in molecules in interval  $n$ . Note that this range increases with  $n$  and that interval 52 includes molecules containing up to 382 atoms. Molecules within each interval have not been sorted and remain in the same sequence as in the original data-set.

When comparing millions of pairs of molecular structures, we aim to spend minimal time using trilabels and prematching to rule out non-isomorphic comparisons.

<sup>22</sup>Here 4 denotes an aromatic bond as in a benzene ring. In all cases we have converted 121212 benzene rings to 444444.

<sup>23</sup>From the US Developmental Therapeutics Program via the bottom link on the page [http://dtp.cancer.gov/docs/3d\\_database/Structural\\_information/structural\\_data.html](http://dtp.cancer.gov/docs/3d_database/Structural_information/structural_data.html).

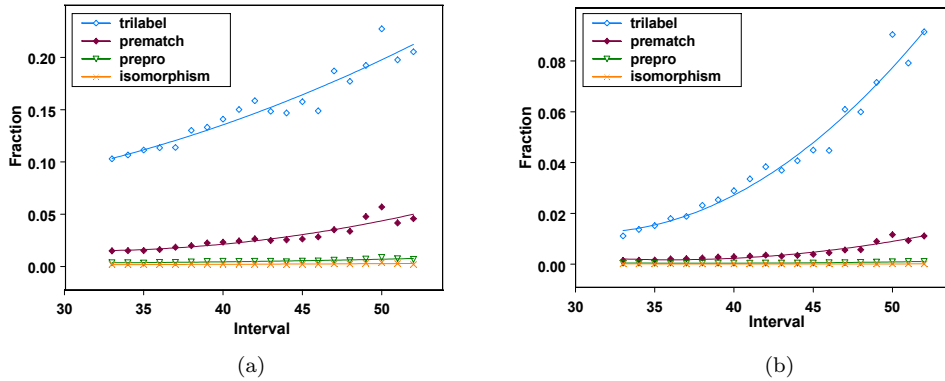


Fig. 25. Fraction admitted versus interval that contains target atoms. (a) Query has 12 atoms;  $n_\alpha = 12$ . (b) Query has 24 atoms;  $n_\alpha = 24$ .

In Figure 25, which explores effectiveness of methods for *avoiding* search for isomorphism,  $n_\alpha$  is the number of atoms in a query. These experiments use the first 3000 molecules<sup>24</sup> having  $n_\alpha$  atoms as queries, and use the first 1000 molecules in each interval 33 through 52 as targets. For  $3000 \times 1000$  comparisons in each of these twenty intervals, Figure 25 shows four fractions of the three million comparisons:

- trilateral* is the fraction wherein no trilateral occurs more times in the query than in the target.
- prematch* is the fraction wherein prematching leaves no domain empty.
- prepro* is the fraction wherein preprocessing leaves no domain empty.
- isomorphism* is the fraction wherein an isomorphism is found. The search terminates when an isomorphism is found.

Figure 25 shows that prematching greatly reduces the number of invocations of preprocessing, particularly when  $n_\alpha = 24$ . Although the number of invocations of prematching increases when *interval* increases, the number of invocations of preprocessing increases more slowly.

Table XIII shows the average total time, including disk access time, for comparing a query with a target. Cases where there is no preprocessing or no search<sup>25</sup> are included in average preprocessing and search times. For example, when there is no search, zero search time is included in the average search time. One reason why total time is less when  $n_\alpha = 24$  is that trilateral and prematch filtering are more effective than when  $n_\alpha = 12$ , as can be seen in Figure 25. Although the search time for focus search is an order of magnitude less than for BVCR when  $n_\alpha = 24$ , there is not much difference in total time, because of the time taken by disk access, trilateral matching and prematching; also because search is avoided in many cases. The time growth curves in Figure 26 are not simply exponential<sup>26</sup>: Timings are similar for BVCR and focus search when  $n_\alpha = 12$ , but focus search is faster<sup>27</sup> when  $n_\alpha = 24$ .

<sup>24</sup>Excluding molecules that are not connected. A molecule is *connected* iff there is a path from any one of its atoms to all of its atoms via bonds. We also exclude any molecule that includes any atom other than C, O, N, H, Cl or S, which occur most frequently in this data-set. Inclusion of unusual atoms would make a query too easy.

<sup>25</sup>For focus the number of searches is exactly the same as it is for BVCR.

<sup>26</sup>Negative slope is attributable to decreasing cardinalities of domains.

<sup>27</sup>Low cardinality of domains improves the speed of focus search. For example, with  $n_\alpha = 24$  there were 917 searches in interval 35; average domain cardinality and average search time were

Table XIII. Average time in microseconds for comparing one query with one target. These averages are over the  $3000 \times 20 \times 1000$  query-target comparisons in Figure 25. In these experiments, BVCR has pop-stack restoration.

Number of atoms in query	BVCR			Focus		
	Total time	Preprocess time	Search time	Total time	Preprocess time	Search time
12	22.455	0.409	0.314	21.536	0.416	0.359
24	11.266	0.141	1.396	10.349	0.126	0.123

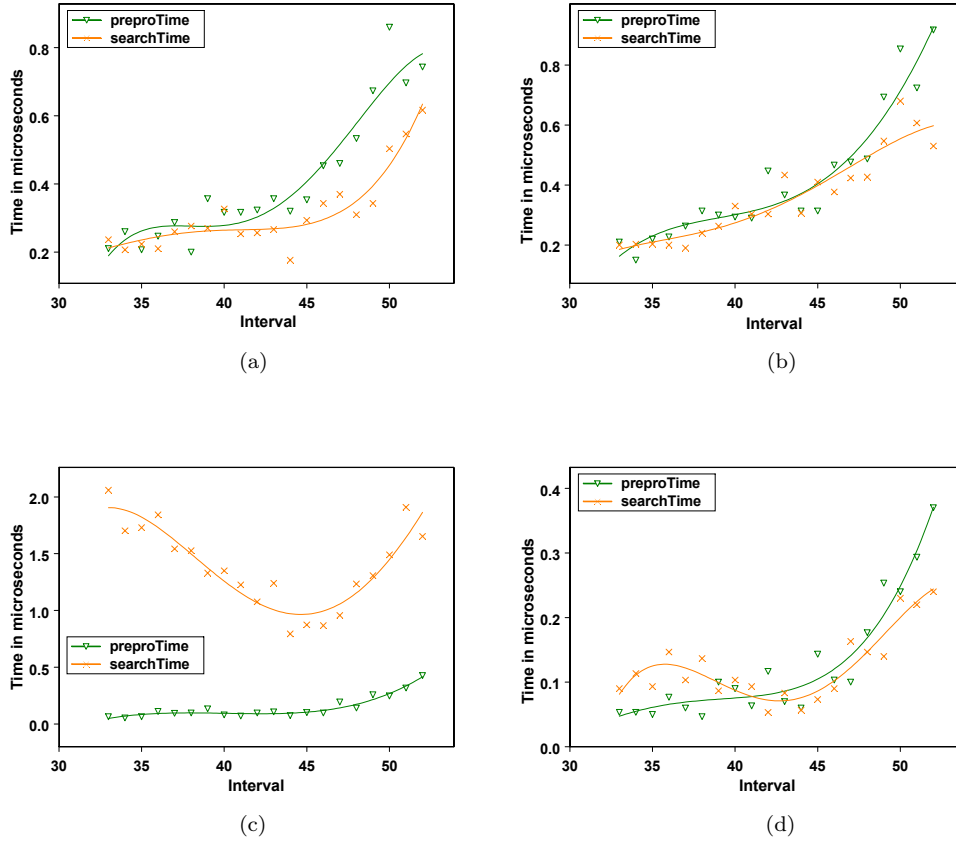


Fig. 26. Preprocess time and search time versus interval. (a) BVCR with  $n_\alpha = 12$ . (b) Focus search with  $n_\alpha = 12$ . (c) BVCR with  $n_\alpha = 24$ . (d) Focus search with  $n_\alpha = 24$ .

Whereas Figure 25 plots fraction against target interval, Figure 27(a) plots fraction against the number,  $n_\alpha$ , of atoms in each query. Total time decreases as  $n_\alpha$  increases because trilateral testing and prematching become more effective. Although focus search time decreases remarkably in Figure 27(d), the effect on total time is negligible.

Figures 25 and 27 show results of a four stage process: trilateral signature matching, prematching, preprocessing and search. Preprocessing, as in Section 6.3.1, applies and propagates the *allDifferent* constraint, which is essential, but trilateral matching and prematching can be omitted. In Table XIV, which illustrates the effect such omissions, there are three cases:

5.465 and 305  $\mu\text{secs}$ , respectively. With  $n_\alpha = 24$  there were 1214 searches in interval 42; average domain cardinality and average search time were 5.100 and 131  $\mu\text{secs}$ , respectively.

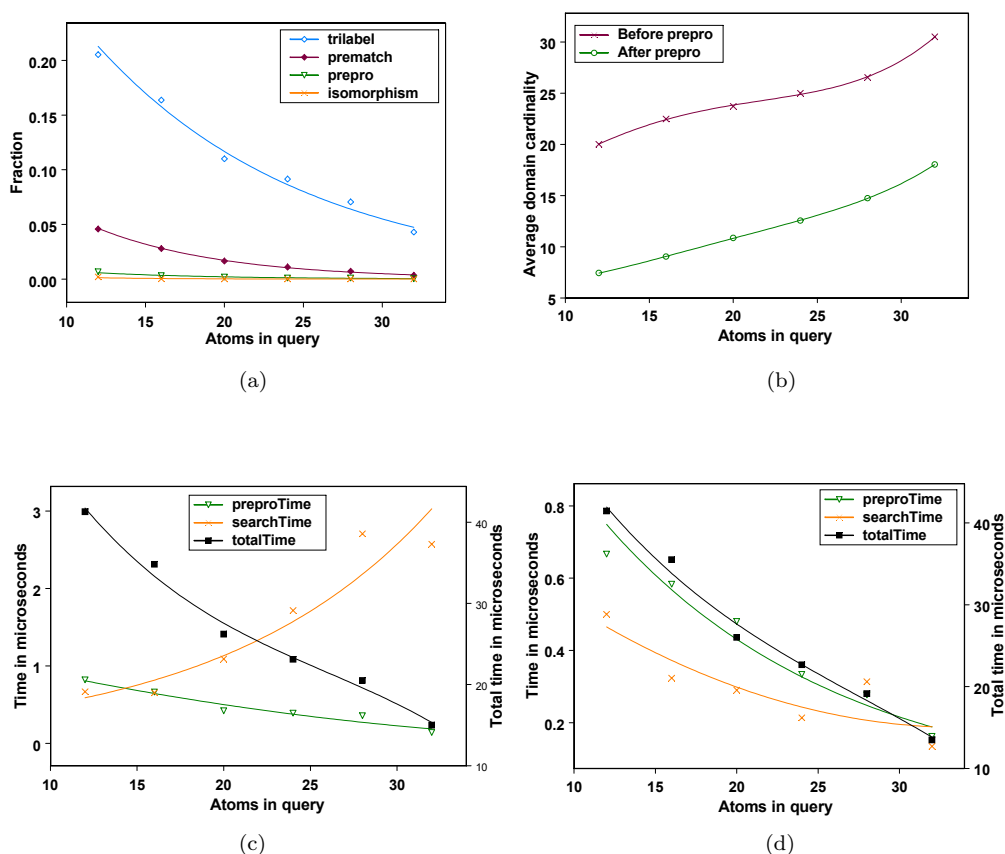


Fig. 27. Results of experiments with 1000 target molecules in Interval 52 and 3000 query molecules in Interval  $n_\alpha$ . The meaning of average times is the same as for Table XIII, except that these are averages over  $3000 \times 1000$  query-target comparisons. (a) Fraction admitted. (b) Average domain cardinality before and after preprocessing. (c) Average search time and preprocessing time (left axis) and total time (right axis) for BVCR. (d) Average search time and preprocessing time (left axis) and total time (right axis) for focus search.

- (1) Trilabel matching and prematching are both applied.
- (2) Trilabel matching is omitted and prematching is applied.
- (3) Trilabel matching is applied and prematching is replaced by  $hk$ -attribute degree matching as in Section 6.6.

These results have been obtained with 60 queries<sup>28</sup> which are not complete (hydrogen-suppressed) molecules, but instead are molecular substructures that are typical of practical queries. In these 60 queries the minimum number of atoms is five, the maximum number is 23 and the average is 10.82. Table XIV confirms that omission of stages increases average overall time. Comparing Case 1 with Case 3, we see that replacing prematching by  $hk$ -attribute degree doubles the total time.

## 8. CONCLUSION

Domain reduction is not an antidote for worst-case exponential time-complexity, but it does allow some real practical problems to be solved, e.g. [Kohler et al. 2002;

<sup>28</sup>Kindly provided by Dr John Holliday, Department of Information Studies, The University of Sheffield, UK.

Table XIV. Results of comparison of 60 queries with 1000 molecules in interval 52. *Number of prepros* is the number of invocations of preprocessing within 60,000 query/target comparisons. *Domain cardinality* is the cardinality after preprocessing, averaged over the number of searches. *Total time* is the average in microseconds over 60,000 comparisons including cases where there is no prematch or no preprocessing or no search. The search algorithm is BVCR with pop-stack restoration.

Case	1	2	3
Number of prepros	4,706	5,208	5,779
Number of searches	1,711	1,724	1,817
Domain cardinality	6.872	7.024	7.016
Total time	45.98	138.25	90.35

Chisholm and Motherwell 2004; Willett 2008; Bandyopadhyay et al. 2009; Hassan 2009]. For binary constraints, bit-vector domain reduction gains speed by using bit-parallel operations that are available in conventional processors. However, there may never be just one bit-vector domain reduction procedure that is always the best one to choose.

Incremental restoration allows us to solve bigger problems than pop-stack restoration, but is usually somewhat slower. Cumulative reduction is sometimes more than twice as fast as direct reduction, but is usually not much more than thirty percent faster. Compared with direct reduction, cumulative reduction does not incur any additional memory cost, but it does make the reduction procedure rather more complicated.

There is a well-known trade-off between, on the one hand, light-weight domain reduction that requires many invocations because each achieves only a little reduction, and on the other hand, heavy-weight reduction that requires fewer invocations because each achieves more reduction [Bessière et al. 2002; Chmeiss and Saïs 2004]. Hitherto, forward checking has been the usual example of a light-weight domain reduction procedure. The main original contribution of this paper is the introduction of focus search, which is lighter-weight than forward checking. Focus search is an ultra light-weight process that does not have save/restore, nor dynamic variable ordering, nor does it update duplicate representations of domains during the search. Implementation of the *allDifferent* constraint is simpler in focus search than in forward checking.

Although heavy-weight reduction, e.g. BVCR, is fastest for many constraint satisfaction problems, light-weight reduction is sometimes an order of magnitude faster. Two examples are when the average degree of randomly-generated constraint graphs is high (Figs 14(a), 15(c), 16(c))), and when constraint symmetry hampers domain reduction in the enumeration of automorphisms of strongly regular graphs (TableX). In these examples, forward checking is faster than BVCR, and focus search is even faster.

Besides introducing focus search, this article has also contributed an up-to-date algorithm for subgraph isomorphism, which differs from an earlier algorithm [Ullmann 1976] as follows:

- (1) Straightforward implementation of a requirement that edges which correspond in an isomorphism must have matching labels.
- (2) Dynamic variable ordering is applied with the weighting heuristic of Boussemart et al [2004].
- (3) Propagational domain reduction procedures employ a queue, following Mackworth [1977].
- (4) Domain reduction is not applied to domains that are single-valued at the time

of invocation of the domain reduction procedure. Linked lists serve to prevent the domain reduction procedure from visiting single-valued domains.

- (5) The *allDifferent* constraint is applied within the domain reduction procedure. When a domain becomes single-valued, this value is removed from all other domains, and in this sense the *allDifferent* constraint is propagated, thus substantially reducing search time.
- (6) Prematching is a non-propagational domain reduction procedure that can be helpful before preprocessing.
- (7) To avoid serial search for 1's in bit vectors, domains can be represented by arrays as well as by bit-vectors, using Briggs-Torczon swapping to facilitate save/restore of array contents. An advantage of this duplicate representation is that it allows efficient incremental save/restore of domains when pop-stack restoration would require too much memory.

Technical developments reported in this article have allowed systematic experimentation on an unusually large scale, although it is only possible to explore an extremely small part of the multidimensional space of possible problems. With randomly-generated binary constraint satisfaction (not isomorphism) problems, we have found that if the average degree  $\rho$  is increased whilst other parameters are unchanged, the search becomes faster on the tight side of crossover but takes longer on the loose side because there are fewer solutions, so the average time to find one is greater. Between the two sides, there is a region where algorithms are intolerably slow. The size of this region increases when  $\rho$  increases in Table IV.

For unlabelled undirected subgraph isomorphism there are *two* separate regions where search may be intolerably slow. As before,  $n_\alpha$  and  $n_\beta$  are the numbers of vertices in the smaller and in the larger graph, respectively. If  $n_\beta$  is fixed while  $n_\alpha$  increases, search time at first increases exponentially, and then decreases as  $n_\alpha$  approaches  $n_\beta$ , in Figure 18, because graph (not subgraph) isomorphism is not NP-Complete. If  $n_\alpha$  is fixed while  $n_\beta$  is increased above  $n_\alpha$ , search time at first increases exponentially and then decreases, in Figure 19, because there are more solutions. This second maximum is analogous to crossover.

NP-Completeness [Cook 1971] has hitherto discouraged development of algorithms for subgraph isomorphism. For directed unlabelled and undirected labelled subgraph isomorphism, which are the most important practical cases, our results suggest that average-time complexity is *not* exponential when  $n_\beta < 5600$ . The time to find a single isomorphism increases quadratically with  $n_\beta$  in Figures 20(a), 21(b), 23(a) and 24(b). It is therefore not surprising that algorithms for subgraph isomorphism have well-established practical applications in chemoinformatics.

Prematching is a valuable addition to the pre-existing armory of techniques for avoiding fruitless searches for molecular subgraph isomorphism. Searches are avoided more successfully as the molecular query size,  $n_\alpha$ , increases in Figures 25 and 27(a). This is why in Figure 27(c)(d) the total time per molecular comparison decreases as  $n_\alpha$  increases, contradicting an assertion [Shang et al. 2008] that the larger the query graphs “the higher the cost for subgraph isomorphism testing.”

We intend that future experimentation will include equitable comparison with Solnon’s [2010] bipartite matching algorithm. We hope that future theoretical progress will explain peaks in numbers of isomorphisms in Figures 21(a), 22, 23(b) and 24(a).

## APPENDIX

## A. REPRESENTATION OF THE SET OF VARIABLES THAT HAVE MULTIVALUED DOMAINS

## A.1 Data structure

While solving a constraint satisfaction problem, much time is usually spent within domain reduction procedures, which therefore require efficient access to adjacent variables whose domains are currently multivalued<sup>29</sup> without wasting time visiting other adjacent variables. To allow rapid access, we provide each variable with a doubly-linked list of adjacent variables. This is doubly-linked so that adjacent variables can easily be unlinked when their domains cease to be multivalued, and can easily be re-linked when multiple values have been restored to their domains.

More specifically, we provide a one-dimensional array of pointers, one for each variable. The pointer for  $V_i$  points to a dummy record in a circular doubly-linked list of multivalued variables adjacent to  $V_i$ . This list is circular and includes a dummy record to prevent the list from becoming empty, thus avoiding the need to check for this. These doubly-linked lists are part of a multilist structure that is accessible via an index, which is another one-dimensional array of pointers, one for each variable, as in Figure 28. For a variable  $V_i$ , the element of the index points to a record for  $V_i$  in the doubly-linked list for an adjacent variable  $V_j$ . This record includes a pointer to the record for  $V_i$  in the doubly-linked list for another variable  $V_k$  that is adjacent to  $V_i$ . This record includes a pointer to the record for  $V_i$  in the doubly-linked list for yet another variable  $V_h$  that is adjacent to  $V_i$ , and so on, until all records for  $V_i$  are included in the list pointed to by the index element for  $V_i$ . The index serves to identify records that should be unlinked/re-linked from/into doubly linked lists, as will be explained.

In the doubly-linked list of variables adjacent to  $V_g$ , the record for an adjacent variable  $V_i$  includes three pointers:

<b>toNextAdjacent</b>	points to the next variable that is adjacent to $V_g$ ;
<b>toPreviousAdjacent</b>	points to the previous variable that is adjacent to $V_g$ ;
<b>toNextForSameVariable</b>	points to the record for $V_i$ within the doubly-linked list for another variable that is adjacent to $V_i$

If the record for adjacent variable  $V_i$  is the last in the list, then **toNextAdjacent** points to the dummy record for  $V_g$ . If the record for adjacent variable  $V_i$  is the first in the list, then **toPreviousAdjacent** points to the dummy record for  $V_g$ . If  $V_i$  is not adjacent to any further variable then its record has **toNextForSameVariable** = **nil**. As well as these three pointers, each record in a doubly-linked list also includes a pointer to the relevant bit-matrix and, for adjacent variable  $V_i$ , the value of  $i$ .

During the search, when procedure *choose* finds that domain  $D_i$  has become single-valued, records for  $V_i$  in all doubly-linked lists are located via the index and are unlinked. However, these records for  $V_i$  remain linked into the list pointed to by the index element for  $V_i$ . When, because of backtrack,  $D_i$  is no longer single-valued, then  $V_i$  is re-linked into the doubly-linked lists for all variables adjacent to  $V_i$ . Relinking is achieved by updating pointers that point to the record for  $V_i$ . Pointers within this record remain unchanged because the sequence in which records are re-linked is exactly the reverse of the sequence in which they were previously unlinked [Knuth 2000].

When used with direct or cumulative domain reduction, procedure *choose* selects, for elective instantiation, a variable whose domain is not already single-valued. We

<sup>29</sup>In this section, *multivalued* means multivalued at the time of invocation of the domain reduction procedure.



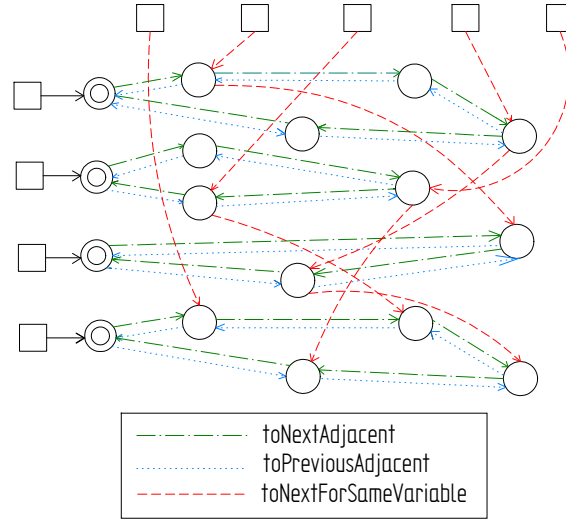


Fig. 28. An incomplete example of doubly linked circular lists threaded by index lists. At the top, rectangles in the horizontal row represent elements of the index. At the left, each rectangle corresponds to a variable and points to a circular list of adjacent variables. Concentric circles represent dummy records.

arrange that procedure *choose* proceeds:

**for** each variable that has a multivalued domain **do** evaluate heuristic score ...

without visiting variables that have already been found to have single-valued domains. For this we maintain an array *varSequence* so that elements *varSequence*[0], *varSequence*[1], ..., *varSequence*[*endSubscript*] identify variables that currently have multivalued domains and elements *varSequence*[*endSubscript*+1], ..., *varSequence*[*n*−1] identify variables that currently have single-valued domains. The main advantage of maintaining this array is that it readily shows which variables should be removed from, or inserted into, doubly-linked lists.

Suppose, as a toy example, that there are only ten variables,  $V_0, \dots, V_9$ , and that before commencement of search, array *varSequence* is initialized to contain:

```

0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
    
```

where the upper row of digits are array subscripts and the lower row are array elements that identify variables. If, for example, all variables are multivalued before procedure *choose* initially selects a variable  $V_3$  then this procedure rearranges the array contents:

```

0 1 2 3 4 5 6 7 8 9
0 1 2 9 4 5 6 7 8 3
    
```

and assigns *endSubscript* := 8, signifying that variables identified by *varSequence*[0], ..., *varSequence*[8] are multivalued. Domain  $D_3$  will be made single-valued by elective instantiation; so, via the index,  $V_3$  will be detached from the doubly-linked lists associated with all variables that are adjacent to  $V_3$ .

Continuing this example, suppose that the next invocation of procedure *reduce* renders  $D_1$  and  $D_5$  single-valued and that procedure *choose* selects  $V_2$  to be instantiated next. In this case, procedure *choose* swaps<sup>30</sup> the array contents:

<sup>30</sup>This idea of swapping has come from Briggs and Torczon [1993] via Cheng and Yap [2008].

```

0 1 2 3 4 5 6 7 8 9
0 8 6 9 4 7 2 5 1 3

```

so that domains identified by  $varSequence[0], \dots, varSequence[endSubscript]$ , where now  $endSubscript = 5$ , are multivalued. If the next invocation of the domain-reduction procedure does not yield an empty domain, then the next invocation of procedure *choose* will only process domains identified by  $varSequence[0], \dots, varSequence[endSubscript]$  so some unnecessary work will be avoided. Far more importantly, variables identified by  $varSequence[endSubscript + 1], \dots, varSequence[previousEndSubscript]$ , where  $previousEndSubscript = 8$ , are detached from the doubly-linked lists of adjacent variables.

Before updating  $endSubscript$ , procedure *choose* pushes onto a stack the value that  $endSubscript$  had at the time of this call of procedure *choose*. To illustrate the use of this, suppose now that in the last example shown above, where finally  $endSubscript = 5$ , the next invocation of the domain reduction procedure does yield an empty domain, and that this is the case after each available value has been assigned to  $V_2$  by elective instantiation. The search must now backtrack to the situation that existed just before  $V_2$  was selected for instantiation. The assignment  $endSubscript := \text{value popped off the stack}$ , which in this example means  $endSubscript := 8$ , signifies that domains of variables  $varSequence[0], \dots, varSequence[endSubscript]$  are again multivalued. This efficient restoration does not put the contents of array  $varSequence$  back into their original sequence, but that does not matter for our purposes. What is again important is that variables identified by  $varSequence[6], \dots, varSequence[8]$  are re-linked into doubly linked lists of adjacent variables.

## A.2 Implementation of procedure *choose*

Figure 29 shows the version of procedure *choose* that is used with bit vector cumulative reduction (BVCR) and direct reduction (BVDR), which continue domain reduction until convergence. This procedure selects for elective instantiation a variable  $V_i$  whose domain is not already single-valued due to implied instantiation. This procedure also identifies further domains that have become single valued, and decrements  $endSubs$  after swapping these out. The call  $endPush(previousEndSubs)$  at Line 22 simply pushes  $previousEndSubs$  onto a stack, which is named  $endStack$ . Although, for introductory simplicity, this is not shown, a call  $relink(endSubscript)$  at the end of Line 21 in Figure 1 achieves:

```

for  $i := endSubscript + 1$  to value now at top of  $endStack$  do
  for  $V_j :=$  each variable adjacent to  $V_i$  do
    re-link record for  $V_i$  into doubly linked list that is associated with  $V_j$ 
  end for
end for

```

This call of *relink* is followed immediately by the call  $endPop(endSubscript)$  which assigns to  $endSubscript$  the value popped off  $endStack$ .

With forward checking we use a version of procedure *choose*, Figure 30, which may select for elective instantiation a variable whose domain is already single valued. Unlike domain reduction procedures that iterate until convergence, forward checking does not require efficient access to adjacent variables that had multivalued domains at the time of invocation of this procedure. Instead it requires efficient access to all adjacent variables that are not already instantiated electively.

The loop at Lines 4 through 10 in Figure 30 does not swap out variables whose domains are found to be single-valued; such variables are candidates for elective instantiation. Therefore the value of  $endSubscript$  is not changed by this loop. Records for the single selected variable are the only ones unlinked from doubly-linked lists by the call of *unlink* at Line 15. These lists enable efficient access to all adjacent variables except those that have been instantiated electively.

```

procedure choose(out  $i$  : integer; out terminal: boolean);
input (via global variables): cardinalities of domains;
output:  $terminal = (\text{all domains are single-valued})$ ;
          $i$  such that  $V_i$  will be instantiated next (if  $terminal = \text{false}$ );
input and output: The following are accessed and updated via global variables:
         endSubscript; (* see Section A.1 *)
         varSequence, endStack; (* of previous endSubscripts *)
         and doubly linked lists; (* see Section A.1 *)
begin
1  if endSubscript < 0 then (* all domains are single-valued *)
    terminal := true; return
  end if;
2  min := greatestAvailableValue;  $i := 0$ ;  $j := 0$ ; previousEndSubs := endSubscript;
3  loop (* over variables already known to be multivalued *)
4     $k := \text{varSequence}[j]$ ;
5    if  $D_k$  is now multivalued then (*  $V_k$  is a candidate for instantiation *)
6      compute heuristic score for the variable  $V_k$ ;
7      if score < min then min := score;  $i := k$ ; iSubscript :=  $j$  end if;
8      if  $j = \text{endSubscript}$  then exit end if;
9       $j := j + 1$ ;
10   else (* swap  $k$  out of the range of multivalued variables in varSequence *)
11     varSequence[ $j$ ] := varSequence[endSubscript];
12     varSequence[endSubscript] :=  $k$ ; endSubscript := endSubscript - 1;
13     if  $j > \text{endSubscript}$  then exit end if;
14   end if;
15 end loop;
16 if endSubscript < 0 then (* no unlinking is required *)
    terminal := true; endSubscript := previousEndSubs; (* to prevent unlink/relink *)
    return
17 else terminal := false
18 end if;
19 if iSubscript < endSubscript then (* swap  $i$  out of multivalued range *)
20   varSequence[iSubscript] := varSequence[endSubscript]; varSequence[endSubscript] :=  $i$ ;
21 end if; (* this swap is not required if iSubscript = endSubscript *)
22 endPush(previousEndSubs); (* to allow subsequent restoration *)
23 unlink(endSubscript, previousEndSubs); (* unlink  $V_i$  from lists identified by
    varSequence[endSubscript]..varSequence[previousEndSubs] *)
24 endSubscript := endSubscript - 1; (* instantiation will make  $D_i$  single-valued *)
end choose;

```

Fig. 29. Procedure *choose* for use with BVDR and BVCR.

## B. ARRAY REPRESENTATION OF DOMAINS

### B.1 Swapping in the array representation of domains

Bitwise parallel **and** and **or** operations in Sections 3.2 and 3.3 necessarily work with bit-vector representation of domains. This representation does not allow efficient search for untried values in Fig 1, nor does it allow efficient enumeration of values within domain reduction procedures (Line 6 in Figures 2 and 3). As well as representing domain  $D_i$  by a bit-vector  $Dsets[i]$ , we can also store values currently in  $D_i$  in  $Darrays[i, 1], \dots, Darrays[i, Dcards[i]]$ , where  $Dcards[i]$  is the current cardinality of  $D_i$ . Values currently in  $D_i$  can be enumerated simply by

```
for  $k := 1$  to Dcards[ $i$ ] do nextValue := Darrays[ $i, k$ ] ...
```

```

procedure choose(out i: integer; out terminal: boolean);
input (via global variables): cardinalities of domains;
output: terminal = (all domains are single-valued);
         i such that  $V_i$  will be instantiated next (if terminal = false);
input and output: The following are accessed and updated via global variables:
         endSubscript; (* see Section A.1 *)
         varSequence, endStack; (* of previous endSubscripts *)
         and doubly linked lists; (* see Section A.1 *)
begin
1  terminal:= true;
2  if endSubscript < 0 then return end if; (* all domains are single-valued *)
3  min:= greatestAvailableValue; i := 0; j := 0;
4  loop (* over variables that have not been electively instantiated *)
5    k := varSequence[j];
6    if  $D_k$  is multivalued then terminal:= false end if ;
7    compute heuristic score for the variable  $V_k$ ;
8    if score < min then min:= score; i := k; iSubscript:= j end if;
9    if j = endSubscript then exit else j := j + 1 end if;
10 end loop;
11 if terminal then return end if;
12 if iSubscript < endSubscript then (* swap i out of multivalued range *)
13   varSequence[iSubscript]:= varSequence[endSubscript]; varSequence[endSubscript]:= i;
14 end if; (* this swap is not required if iSubscript = endSubscript *)
15 endPush(endSubscript); unlink(endSubscript, endSubscript);
16 endSubscript:= endSubscript - 1; (* instantiation will make  $D_i$  single-valued*)
end choose;

```

Fig. 30. Procedure *choose* for use with forward checking.

without searching for 1's in Dsets[*i*]. Following Briggs and Torczon [1993], as in Section A.1, a value  $v$  can be removed from the *Darrays* representation of  $D_i$  by swapping it out of the subarray *Darrays*[*i*, 1], ..., *Darrays*[*i*, Dcards[*i*]]. Values that are not currently in domain  $D_i$  reside in *Darrays*[*i*, Dcards[*i*] + 1], ..., *Darrays*[*i*,  $\delta$ ], where  $\delta$  is the greatest cardinality of any domain. Given the subscript that locates a value  $v$  in *Darrays*[*i*], value  $v$  can be swapped out by:

```

procedure swapOut(in i, subscript, v: integer; in out dCard: integer);
input: i identifies a domain; v is a value that is to be removed from  $D_i$ ;
        subscript is such that  $v$  resides at Darrays[i, subscript];
input and output: dCard is the current cardinality of  $D_i$ ;
        Darrays[i, 1]..Darrays[i, dCard] are the values currently in  $D_i$ ;
begin
  Darrays[i, subscript]:= Darrays[i, dCard]; Darrays[i, dCard]:= v;
  dCard:= dCard-1
end swapOut

```

Values can be restored to the array representation of  $D_i$  by appropriately incrementing Dcards[*i*]. As a toy example, in which the top row shows subscripts and the second row shows values in *Darrays*[*i*],

```

0 1 2 3 4 5 6 7 8 9
0 8 6 9 4 7 2 5 1 3

```

with Dcards[*i*] = 4 we have  $D_i = \{0, 8, 6, 9\}$ . Suppose for example that {4, 7, 2} have been removed from  $D_i$  by domain reduction. It is important that {4, 7, 2} can be restored to domain  $D_i$  simply by Dcards[*i*] := Dcards[*i*] + 3, without moving

any value within *Darrays*. Values may not be restored into their original sequence within *Darrays*, but that does not matter here.

Many domains must be saved and restored during the search. Although this is not shown in Figure 1, the search routine saves the whole of the two-dimensional array *Darrays* at Line 6 by pushing the whole of the one dimensional array *Dcards* onto a stack. It is important that the two-dimensional array *Darrays* is not pushed onto a stack. At Lines 14, 18 and 21 in Fig 1, which does not show this detail, the two-dimensional array *Darrays* is restored by popping the one-dimensional array *Dcards* off the stack [Cheng and Yap 2008], without moving any value within *Darrays*.

Pop-stack save/restore of the entire array *Dsets* can be avoided by using incremental restoration, which is easy when *Darrays* and *Dcards* are up-to-date and consistent with *Dsets* at the time of restoration:

```

oldCards:= Dcards; Dcards:= DcardStack[top]; top:= top - 1;
(* Dcards has been popped off the stack *)
for i := 0 to n - 1 do (* for each variable *)
  for k := oldCards[i] + 1 to Dcards[i] do include Darrays[i, k] in Dsets[i]
  end for
end for

```

In many cases we find experimentally that the use of double (i.e. *Dsets* and *Darrays*) representation does not significantly improve the speed of search, and in some cases makes it slower. However, the availability of incremental restoration is a clear advantage of double representation.

## B.2 Direct and cumulative reduction with array representation

The direct reduction procedure in Figure 31 uses bit-vector and array representation of domains. After removing a value  $u$  from *Dsets*[ $i$ ] this procedure swaps  $u$  out of *Darrays*[ $i$ ] only if  $D_i$  is not now empty. If  $D_i$  would be emptied by removing the last value, this value is not removed from *Dsets*, nor is it swapped out of *Darrays*. In this case the assignment *Dcards*[ $i$ ] := 1 makes *Dsets*, *Darrays* and *Dcards* mutually consistent, so incremental restoration can be used straightforwardly.

Array representation can also be used with cumulative reduction, as in Figure 32. Unlike direct reduction, cumulative reduction does not remove values from a domain  $D_i$  sequentially, and therefore cannot update *Darrays*[ $i$ ] as soon as a value is removed. However, if  $D_i$  is reduced, then  $i$  is put into the queue, and will be  $j$  in a subsequent iteration. After removing  $j$  from the queue, cumulative reduction processes values in  $D_j$  sequentially, aiming to reduce adjacent domains. During this sequential process for the first variable adjacent to  $V_j$ , the procedure updates *Darrays*[ $j$ ] and *Dcards*[ $j$ ]

The cumulative reduction procedure may return *consistent* = **false** when the queue contains variables for which *Darrays* is inconsistent with *Dsets*. This procedure does not waste time enforcing consistency that will anyway be over-ridden by pop-stack restoration. With pop-stack restoration, this is an advantage of cumulative reduction, but not with incremental restoration, which requires *Darrays*, *Dsets* and *Dcards* to be mutually consistent. With incremental restoration, just before the cumulative reduction procedure returns *consistent* = **false**, we restore consistency by:

```

for g:= each variable in the queue do
  for x := 1 to Dcards[g] do include Darrays[g, x] in Dsets[g] end for
end for

```

With forward checking, *Dcards* is required for reference by procedure *choose*. The

```

procedure reduce(in h: integer; out consistent: boolean);
input:  h identifies the variable that has just been instantiated;
        Bit matrices are accessed via global variables;
input and output: Dsets is an array of bit-vectors representing domains;
        Dcards is an array of domain cardinalities;
        Darrays is an array of arrays that represent domains;
output:  consistent = (no domain is empty);
begin
1  initialize queue to be empty; insert h into queue;
2  repeat
3    j := variable removed from queue;
4    for each variable  $V_i$  that is adjacent to  $V_j$  such that  $D_i$  is multivalued do
      (* uSubscript is the subscript of the next value in Darrays[i] *)
5    uSubscript := 1; dCard := Dcards[i]; changed := false;
6    while uSubscript ≤ dCard do (* for each value now in  $D_i$  do *)
7      u := Darrays[i, uSubscript];
8      if  $M_j^i[u] * Dsets[j] = 0$  then (* there is no support in  $D_j$  for u in  $D_i$  *)
9        if dCard = 1 then (*  $D_i$  would be empty after removing u *)
           $w_{ij} := w_{ij} + 1$ ; consistent := false; Dcards[i] := 1; return
        end if;
10       remove u from Dsets[i]; changed := true;
11       swapOut(i, uSubscript, u, dCard); (* swap u out of Darrays[i] *)
12       else uSubscript := uSubscript + 1 (* to access next value in  $D_i$  *)
13       end if;
14     end while;
15     if changed and ( $i \notin$  queue) then insert i into queue end if;
16     Dcards[i] := dCard;
17   end for;
18 until queue is empty;
19 consistent := true;
end reduce;

```

Fig. 31. A bit-vector direct reduction implementation of procedure *reduce* with array representation of domains.

version of forward checking in Figure 33 counts 1's in bit-vectors while updating *Darrays* if *consistent* = **true**. If *consistent* = **false** then, with pop-stack restoration, this procedure does not waste time updating *Darrays* and *Dcards*, which will be over-written by restoration. With incremental restoration, if *consistent* = **false**, we make *Darrays* consistent with *Dsets* in the same way as for cumulative reduction. That is, we refer to *Darrays* to restore all values that have been removed from *Dsets*.

By referring to *Darrays*, the loop at Line 8 in Figure 33 avoids checking every bit in *Dsets*[*i*]. If we do not have duplicate representation of domains, we need to check every bit in *Dsets*[*i*] to update *Dcards*[*i*]. This is particularly unwelcome in forward checking, in which the domain reduction procedure is intended to be maximally simple. Another comment is that array representation enables the search routine to find untried values without visiting 0's in bit vectors. This is important for forward checking, which usually involves many more elective instantiations than direct or cumulative reduction.

### B.3 Enforcing the *allDifferent* constraint using the array representation of domains

With direct reduction we enforce *allDifferent* by inserting the fragment shown in Fig 34 between Lines 3 and 4 in Figure 31. This propagates the *allDifferent* constraint via the queue. Moreover, when a value is removed from a domain, *Dsets*, *Darrays* and *Dcards* remain mutually consistent, so this fragment works correctly with pop-stack or incremental restoration.

When *j* is removed from the queue during cumulative reduction, *Dcards*[*j*] is out of date. We therefore wait until *Dcards*[*j*] has been updated before checking whether

```

procedure reduce(in h: integer; out consistent: boolean);
input:  h identifies the variable that has just been instantiated;
        Bit matrices are accessed via global variables;
input and output: Dsets is an array of bit-vectors representing domains;
        Dcards is an array of domain cardinalities;
        Darrays is an array of arrays that represent domains;
output:  consistent = (no domain is empty);
begin
1  initialize queue to be empty; insert h into queue;
2  repeat
3      j:= variable removed from queue; dCard:= Dcards[j];
4      if there is no variable adjacent to  $V_j$  that has a multivalued domain then
5          vSubscript:= 1;
6          while vSubscript ≤ dCard do (* without attempting domain reduction
           update Darrays[j] by removing values that are not in Dsets[j] *)
7              v:= Darrays[j, vSubscript];
8              if v in Dsets[j] then vSubscript:= vSubscript + 1 (* to access next v *)
9              else swapOut(j, vSubscript, v, dCard)
10             end if;
11         end while;
12      else
13          firstAdjacent:= true;
14          for each variable  $V_i$  that is adjacent to  $V_j$  such that  $D_i$  is multivalued do
15              B:= empty; vSubscript:= 1;
16              if firstAdjacent then (* update Darrays[j] and accumulate B *)
17                  while vSubscript ≤ dCard do
18                      v:= Darrays[j, vSubscript];
19                      if v in Dsets[j] then  $B := B + M_i^j[v]$ ; vSubscript:= vSubscript + 1
20                      else swapOut(j, vSubscript, v, dCard)
21                      end if;
22                  end while;
23                  firstAdjacent:= false;
24              else (* Darrays[j] is up-to-date, so only accumulate B *)
25                  while vSubscript ≤ dCard do
26                       $B := B + M_i^j[Darrays[j, vSubscript]]$ ; vSubscript:= vSubscript + 1
27                  end while;
28              end if;
29               $B := Dsets[i] * B$ ; (*  $B := \{u \in D_i | u \text{ is supported in } D_j\}$  *)
30              if Dsets[i] ≠ B then (*  $D_i$  will be reduced *)
31                  if B = 0 then  $w_{ij} := w_{ij} + 1$ ; consistent:= false; return end if
32                  Dsets[i]:= B; (* reducing domain  $D_i$  *)
33                  if ( $i \notin \text{queue}$ ) then insert i into queue end if;
34              end if;
35          end for;
36      end if;
37      Dcards[j]:= dCard;
38  until queue is empty;
39  consistent:= true;
end reduce
    
```

Fig. 32. A bit-vector cumulative reduction implementation of procedure *reduce* with array representation of domains.

$Dcards[j] = 1$ . With pop-stack restoration, we can simply insert the fragment shown in Figure 34, with its Line 8 deleted, between Lines 37 and 38 in Figure 32. Line 8 is deleted because  $Darrays[k]$  will be updated when  $k$  is removed from the queue.

With cumulative reduction and incremental restoration, if removing  $u$  would empty  $D_k$ , then for each variable,  $i$ , in the queue, we insert into  $Dsets[i]$  all the values now in  $Darrays[i]$ , to make these mutually consistent. This correction is a disadvantage of cumulative reduction.

With forward checking, Line 2 in Figure 33 becomes

```

procedure forwardCheck(in  $j, v$  : integer; in out  $Dsets$ : array of bit-vectors;
                        out  $consistent$ : boolean);
input:  $j$  identifies the variable that has just been instantiated;
        $v$  is the value that has been assigned to  $j$  by elective instantiation;
       Bit matrices are accessed via global variables;
input and output:  $Dsets$  is an array of bit-vectors representing domains;
                   $Dcards$  is an array of domain cardinalities;
                   $Darrays$  is an array of arrays that represent domains;
output:  $consistent = (\text{no domain is empty})$ ;
begin
1  for each variable  $V_i$  adjacent to  $V_j$  that has not been electively instantiated do
2     $Dsets[i] := Dsets[i] * M_i^j[v]$ ;  (*  $D_i := D_i \cap \{u \in D_i | u \text{ is supported by } v \in D_j\}$  *)
3    if  $Dsets[i] = 0$  then  $w_{ij} := w_{ij} + 1$ ;  $consistent := \text{false}$ ; return end if;
4  end for
5   $consistent := \text{true}$ ;
6  for each variable  $V_i$  adjacent to  $V_j$  that has not been electively instantiated do
    (* update  $Darrays$  and  $Dcards$  *)
7     $uSubscript := 1$ ;  $dCard := Dcards[i]$ ;
8    while  $uSubscript \leq dCard$  do (* for each value in  $Darrays[i]$  *)
9       $u := Darrays[i, uSubscript]$ ;
10     if  $u$  in  $Dsets[i]$  then  $uSubscript := uSubscript + 1$ 
11     else  $swapOut(i, uSubscript, u, dCard)$ 
12     end if
13   end while;
14    $Dcards[i] := dCard$ ;
15 end for
end forwardCheck;

```

Fig. 33. Forward checking with array representation of domains.

```

1  if  $Dcards[j] = 1$  then (*  $D_j$  is single-valued *)
2    let  $u$  be the single value in  $D_j$ ;
3    for each  $k$  such that  $D_k$  is multivalued do
4      if ( $k \neq j$ ) and ( $u \in Dsets[k]$ ) then
5        if  $Dcards[k] = 1$  then (* removing  $u$  would empty  $D_k$  *)
           $consistent := \text{false}$ ; return
        end if;
6        remove  $u$  from  $Dsets[k]$ ;  $Dcards[k] := Dcards[k] - 1$ ;
7        if  $k \notin \text{queue}$  then insert  $k$  into queue end if;
        (* subsequently  $D_k$  may be found single-valued when  $k$  is removed from queue *)
8        find  $u$  in  $Darrays[k]$ ; swap  $u$  out of  $Darrays[k]$ ;
9      end if
10    end for
11  end if

```

Fig. 34. Fragment inserted between Lines 3 and 4 in Fig 31 to enforce the *allDifferent* constraint when pop-stack or incremental domain restoration is used with direct reduction.

$$Dsets[i] := Dsets[i] * M_i^j[v] - \{v\};$$

so the value  $v$  of  $V_j$  is removed from domains of all uninstantiated variables that are adjacent to  $V_j$ . Furthermore, with pop-stack restoration, the amended forward checking procedure ends by removing  $v$  from all  $D_k$  such that  $k \neq j$ ,  $V_k$  is not electively instantiated, and  $V_k$  is not adjacent to  $V_j$  (because adjacent domains have already been processed). If this empties  $D_k$  then the procedure returns  $consistent = \text{false}$ . Otherwise,  $v$  is found in, and swapped out of,  $Darrays[k]$ . As in the previous section, serial search for  $v$  is unwelcome in forward checking, which is intended to be a quick-and-cheap procedure. With incremental restoration, further work is required, as before, to make  $Dsets$ ,  $Darrays$  and  $Dcards$  mutually consistent when



the procedure returns *consistent* = **false**.

### C. RELATIONSHIP BETWEEN FOCUS SEARCH AND PARTITION SEARCH

This article is primarily concerned with constraints between pairs of variables. In problems where there are constraints between more than two variables, the constraints are said to be *non-binary*. A set of variables that are together subject to a constraint constitute the *scope* of that constraint; the *arity* of the constraint is the number of variables in its scope.

Focus search belongs to a family of algorithms that also includes partition search for non-binary constraint satisfaction [Ullmann 2007]. Like focus search, partition search employs a static instantiation sequence<sup>31</sup> and avoids save/restore. However, partition search uses a tabular representation of constraints: this means that the constraint on any given scope is represented by an explicit relational table of tuples. At the time when a set  $Y_i$  of variables have already been instantiated, partition search works with a scope  $S_j$  that maximally overlaps  $Y_i$ . Let  $S_j^b$  be the set of variables in  $S_j$  that are not in  $Y_i$ . Let  $T_j^b$  be the set of tuples on  $S_j^b$  such that each tuple in  $T_j^b$  is part of a tuple on  $S_j$

—that satisfies the constraint on  $S_j$  and

—matches instantiated values of variables in  $Y_i$  at this time.

Partition search instantiates  $S_j^b$  to successive tuples in  $T_j^b$ , which are obtained by accessing a data structure such as a hash table or a trie.

Now suppose that, at this same time, there is another scope  $S_k$  that maximally overlaps  $Y_i$ , such that  $S_k^b = S_j^b$ . In this case,  $S_j^b$  can in principle be instantiated to successive tuples in  $T_j^b \cap T_k^b$ ; but if  $|S_j^b| > 1$  then enumeration of this intersection would take an unwelcome amount of time. Actually partition search would successively instantiate  $S_j^b$  to each tuple in  $T_j^b$ , and then check whether the instantiated tuple was also in  $T_k^b$ .

Again suppose that a set  $Y_i$  of variables have already been instantiated, and now suppose also that every scope comprises exactly two variables. In this case  $T_j^b$  is a set of values of a single variable; this set can be represented by a bit-vector. An intersection such as  $T_j^b \cap T_k^b$  is easily implemented by bitwise **and**. Focus search employs this intersection, whereas partition search does not, which is why partition search is useless with binary constraints. As explained previously, focus search accesses a bit-vector such as  $M_j^i[u]$  in a one-dimensional array using subscript  $u$ . A one-dimensional array is inadequate when  $|S_j \cap Y_i| > 1$ .

Lines 1 through 4 of the focus search procedure *reduce* check the support of values in domains of variables that have not yet been instantiated. Partition search only checks consistency of values of instantiated variables.

Partition search takes account of the sequence of variables within each scope. To avoid the cost of constant redirection during the search, partition search re-numbers the variables and also the scopes. Re-numbering is unnecessary in focus search, which imposes the chosen static sequence via arrays *predecessor* and *successor*.

### D. TRILABEL SIGNATURE MATCHING

When a query graph is compared with hundreds of thousands of target graphs, the first step in each of these comparisons is an attempt to match the query graph trilabel signature with the target trilabel signature. The trilabel signature comparison process should be maximally efficient because it is usually executed a very large number of times. For each query trilabel, minimal time should be spent searching for a matching target trilabel.

<sup>31</sup>Also based on the maximum cardinality algorithm [Tarjan and Yannakakis 1984].

Table XV. Examples of query and target trilabels.

Query				Target			
first vertex	second vertex	edge label	count	first vertex	second vertex	edge label	count
60	62	2	3	55	65	4	2
65	65	1	2	58	60	1	1
				58	62	2	4
				58	62	3	4
				60	62	2	3
				60	63	4	1
				60	65	3	3
				60	69	5	4
				64	64	1	1
				65	65	1	1
				65	66	4	3
				68	72	3	3

Let  $n_\lambda$  be the number of distinct vertex labels that may occur anywhere in the set of target graphs. We represent each distinct vertex label by a unique ordinal in the range  $1, \dots, n_\lambda$ . These representative ordinals are chosen so that if vertex label  $L_i$  is more frequent than vertex label  $L_j$ , then the ordinal that represents  $L_i$  is greater than the ordinal that represents  $L_j$ . The frequency of a vertex label is proportional to the total number of its occurrences in the entire set of target graphs. We take the liberty of not distinguishing between a vertex label and the ordinal that represents it. Sorting vertex labels into increasing order means sorting vertex labels into increasing order of their representative ordinal.

Every trilabel is a triple  $\langle \text{vertexOneLabel}, \text{vertexTwoLabel}, \text{edgeLabel} \rangle$  sorted so that  $\text{vertexOneLabel} \leq \text{vertexTwoLabel}$ , which means that  $\text{vertexOneLabel}$  is not more frequent than  $\text{vertexTwoLabel}$ . We process trilabels in sequence of increasing frequency, because failure is more likely earlier than later in this sequence. More specifically, trilabels are always processed in the lexicographic sequence illustrated in Table XV.

In the example in Table XV, as soon as  $\text{vertexOneLabels}$  are compared, we see that target trilabel  $\langle 55, 65, 4 \rangle$  does not match query trilabel  $\langle 60, 62, 2 \rangle$ . Again, when the  $\text{vertexOneLabels}$  are compared we see that target trilabel  $\langle 58, 60, 1 \rangle$  does not match query trilabel  $\langle 60, 62, 2 \rangle$ . After discovering this mismatch of  $\text{vertexOneLabels}$ , we *skip* futile comparisons between query trilabel  $\langle 60, 62, 2 \rangle$  and subsequent target trilabels  $\langle 58, 62, 2 \rangle$  and  $\langle 58, 62, 3 \rangle$  that have  $\text{vertexOneLabel} = 58$ . Subsequently, after discovering the match between query trilabel  $\langle 60, 62, 2 \rangle$  and a target trilabel, we *skip* futile comparisons of query trilabel  $\langle 65, 65, 1 \rangle$  with further target trilabels that have  $\text{vertexOneLabel} = 60$ .

We skip futile comparisons by storing each signature in its own individual two-level structure that is similar to a de la Briandais [1959] trie. For each signature, the first level of the trie is the *head*, and the second level is the *body*. The head is an array of records that have two fields:

- $\text{vertexOneLabel}$ ,
- the number of trilabels in this signature that have this  $\text{vertexOneLabel}$  for the first vertex.

The body is an array of records that have three fields:

- $\text{vertexTwoLabel}$
- $\text{edgeLabel}$
- number of occurrences of this trilabel in this signature.

In the body there is one record corresponding 1:1 to each trilabel in the signature. For the query in Table XV, the head is

Table XVI. Head and body of target signature.

head		-	body		
first vertex	nbOf trilabels	-	second vertex	edge label	count
55	1	-	65	4	2
58	3	-	60	1	1
60	4	-	62	2	4
64	1	-	62	3	4
65	2	-	62	2	3
68	1	-	63	4	1
		-	65	3	3
		-	69	5	4
		-	64	1	1
		-	65	1	1
		-	66	4	3
		-	72	3	3

60 1  
65 1

and the body is

62 2 3  
65 1 2

The number 3 at the right end of the first line in the body signifies that the trilabel  $\langle 60, 62, 2 \rangle$  occurs three times in the query graph. Similarly, the number 2 in the next line signifies that the trilabel  $\langle 65, 65, 1 \rangle$  occurs twice in the query graph. Table XVI shows the head and body of the target signature in Table XV.

Figure 35 is a transcription of our signature matching procedure. For our example this procedure visits the first three records in the target head, seeking *firstVertexLabel* = 60. The procedure skips the first four records in the target body, because these are associated with *firstVertexLabel* < 60. After finding the matching trilabel  $\langle 60, 62, 2 \rangle$ , the procedure visits the next two records in the target head, seeking *firstVertexLabel* = 65, and skips four records in the target body that are associated with *firstVertexLabel* < 65. The procedure terminates as soon as it finds the matching trilabel  $\langle 65, 65, 1 \rangle$ .

If the example is amended so that the first query trilabel is now  $\langle 54, 62, 2 \rangle$ , the first iteration of the loop at Lines 7 through 19 returns **false** at Line 14 in Figure 35, when no body records have been visited. If, instead, the first query trilabel is now changed to  $\langle 56, 62, 2 \rangle$ , the second iteration of the loop at Lines 7 through 19 returns **false** at Line 14, when no body records have been visited. If, instead, the first query trilabel is changed to  $\langle 60, 61, 2 \rangle$ , the procedure returns **false** at Line 28 when just one query body record and one target body record have been visited. For signatures that do not match, the procedure is intended to return **false** as soon as possible. For our initial example where the query signature is  $(\langle 60, 62, 2 \rangle, \langle 65, 65, 1 \rangle)$ , the procedure returns **true** without visiting the last two target trilabels.

Hundreds of thousands of target signatures cannot all reside in fast memory. Reading successive target signatures in turn individually from disk would take too long because of latency. Instead, we store target signatures on disk within big pages, each holding many target signatures. We ensure that each target signature is wholly within a single page. Here a page is a fixed-size block of, for example, 20,000 bytes. When a page is read from disk it looks like an array of bytes. We do not copy successive target signatures from this into the array-of-records structure described above. Instead, our signature matching procedure works with target signatures directly in array-of-bytes form in the page buffer, to save time. Figure 35 is a transcription in the sense that it has the target signature in array-of-records form, because this is easier to understand. This procedure does, however, work with the

```

procedure signatureMatches(): boolean;
input: Query and target signatures in array-of-records form;
output: Return true iff every query trilateral matches a different target trilateral;
begin
1  targetHeadSubscript:= 1; targetBodySubscript:= 1; queryBodySubscript:= 1;
2  for queryHeadSubscript:= 1 to qNbFirstLabels do
3    with queryHead[queryHeadSubscript] do
4      qFirst:= firstVertexLabel; qNbTuples:= nbTuples;
5    end with;
6    queryBodyEndSubscript:= queryBodySubscript + qNbTuples;
7    loop (* seeking target firstVertexLabel = qFirst *)
8      if target[targetHeadSubscript].firstVertexLabel = qFirst then (* found *)
9        tNbTuples:= target[targetHeadSubscript].nbTuples;
10       if tNbTuples < qNbTuples then return false end if;
11       targetHeadSubscript:= targetHeadSubscript + 1; exit;
12     end if;
13     if (target[targetHeadSubscript].firstVertexLabel > qFirst) or
14       (targetHeadSubscript = tNbFirstLabels) then return false
15     else
16       targetBodySubscript:= targetBodySubscript + tNbTuples;
17       targetHeadSubscript:= targetHeadSubscript + 1
18     end if
19   end loop;
20   loop (* for each entry in query body that has this qFirst *)
21     with queryBody[queryBodySubscript] do
22       qSecond:= secondVertexLabel; qEdge:= edgeLabel; qCount:= count;
23     end with;
24     loop (* seek matching entry in target body *)
25       with targetBody[targetBodySubscript] do
26         tSecond:= secondVertexLabel; tEdge:= edgeLabel; tCount:= count;
27       end with;
28       if tSecond > qSecond then return false end if;
29       if (tSecond < qSecond) or (tEdge < qEdge) then
30         if tNbTuples = 1 then return false (* because no more tSeconds for tFirst *)
31         else tNbTuples:= tNbTuples - 1; targetBodySubscript:= targetBodySubscript + 1;
32         end if
33       elseif (tEdge > qEdge) or (qCount > tCount) then return false
34       else (* match has been found in target body *)
35         targetBodySubscript:= targetBodySubscript + 1;
36         tNbTuples:= tNbTuples - 1; exit
37       end if;
38     end loop; (*loop over target body *)
39     queryBodySubscript:= queryBodySubscript + 1;
40     if queryBodySubscript = queryBodyEndSubscript then exit end if;
41     if tNbTuples = 0 then return false end if;
42   end loop;
43   targetBodySubscript:= targetBodySubscript + tNbTuples;
44 end for; (* for query head subscript *)
45 return true
end signatureMatches

```

Fig. 35. Transcription of a signature matching procedure.

query signature in array-of-records form.

#### ACKNOWLEDGMENTS

The author wishes to thank Peter Willett for providing molecular advice, John Holliday for providing sample queries, and the reviewers for valuable comments, suggestions and guidance.

## REFERENCES

- AARDAL, K. I., VAN HOESSEL, S. P. M., KOSTER, A. M. C. A., MANNINO, C., AND SASSANO, A. 2007. Models and solution techniques for frequency assignment problems. *Annals of Operations Research* 153, 1, 79–129.
- ARTYMIUK, P. J., SPRIGGS, R. V., AND WILLETT, P. 2005. Graph theoretic methods for the analysis of structural relationships in biological macromolecules. *J. Am. Soc. Inf. Sci. Technol.* 56, 5, 518–528.
- BANDYOPADHYAY, D., HUAN, J., PRINS, J., SNOEYINK, J., WANG, W., AND TROPISHA, A. 2009. Identification of family-specific residue packing motifs and their use for structure-based protein function prediction: I. method development. *Journal of Computer-Aided Molecular Design* 23, 11, 773–784.
- BESSIÈRE, C. 2006. Constraint Propagation. *Handbook of Constraint Propagation*. Elsevier, New York, NY, USA, Chapter 3. Eds: Rossi, F., van Beek, P., and Walsh, T. <http://www.lirmm.fr/~bessiere/stock/TR06020.pdf>.
- BESSIÈRE, C., MESEGUER, P., FREUDER, E. C., AND LARROSA, J. 2002. On forward checking for non-binary constraint satisfaction. *Artif. Intell.* 141, 1, 205–224.
- BESSIÈRE, C., RÉGIN, J.-C., YAP, R. H. C., AND ZHANG, Y. 2005. An optimal coarse-grained arc consistency algorithm. *Artif. Intell.* 165, 2, 165–185.
- BESSIÈRE, C., STERGIOU, K., AND WALSH, T. 2008. Domain filtering consistencies for non-binary constraints. *Artif. Intell.* 172, 6-7, 800–822.
- BOUSSEMARY, F., HEMERY, F., AND LECOUTRE, C. 2004. Revision ordering heuristics for the constraint satisfaction problem. In *Proc of the First International Workshop on Constraint Propagation and Implementation (CPAI'2004) held with CP'2004*. 29–43. <http://www.cril.univ-artois.fr/~lecoutre/research/publications/2004/CPW2004.pdf>.
- BOUSSEMARY, F., HEMERY, F., LECOUTRE, C., AND SAÏS, L. 2004. Boosting systematic search by weighting constraints. In *16th European Conference on Artificial Intelligence (ECAI'2004)*. 146–150. <http://www.cril.univ-artois.fr/~lecoutre/research/publications/2004/ECAI2004.pdf>.
- BOUTSELAKIS, H., DIMITROPOULOS, D., FILLON, J., GOLOVIN, A., HENRICK, K., HUSSAIN, A., IONIDES, J., JOHN, M., KELLER, P. A., KRISSINEL, E., MCNEIL, P., NAIM, A., NEWMAN, R., OLDFIELD, T., PINEDA, J., RACHEDI, A., COPELAND, J., SITNOV, A., SOBHANY, S., SUAREZ-URUENA, A., SWAMINATHAN, J., TAGARI, M., TATE, J., TROMM, S., VELANKAR, S., AND VRANKEN, W. 2003. MSD: the European Bioinformatics Institute Macromolecular Structure Database. *Nucleic Acids Res.* 31, 1, 458–462.
- BRIGGS, P. AND TORCZON, L. 1993. An efficient representation for sparse sets. *ACM Lett. Program. Lang. Syst.* 2, 1-4, 59–69.
- BROWN, N. 2009. Chemoinformatics—an introduction for computer scientists. *ACM Comput. Surv.* 41, 2, 1–38.
- CHANG, C. C. AND LEE, S. Y. 1991. Retrieval of similar pictures on pictorial databases. *Pattern Recogn.* 24, 7, 675–681.
- CHENG, J., KE, Y., AND NG, W. 2009. Efficient query processing on graph databases. *ACM Trans. Database Syst.* 34, 1, 1–48.
- CHENG, K. C. AND YAP, R. H. 2008. Maintaining arc consistency on ad-hoc r-ary constraints. In *14th International Conference on Principles and Practice of Constraint Programming (CP'2008)*. 509–523.
- CHENG, K. C. AND YAP, R. H. 2010. An mdd-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints* 15, 2, 265–304.
- CHISHOLM, J. A. AND MOTHERWELL, S. 2004. A new algorithm for performing three-dimensional searches of the cambridge structural database. *Journal of Applied Crystallography* 37, 331–334.
- CHMEISS, A. AND SAÏS, L. 2004. Constraint satisfaction problems: Backtrack search revisited. In *ICTAI '04: Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence*. IEEE Computer Society, Washington, DC, USA, 252–257.
- CHOU, Y.-Y. AND SHAPIRO, L. G. 1998. Probabilistic relational indexing. In *Proceedings of Fourteenth International Conference on Pattern Recognition*. 1331–1335. Volume 2.
- CONTE, D., FOGGIA, P., SANSONE, C., AND VENTO, M. 2004. Thirty years of graph matching in pattern recognition. *International Journal of Pattern Recognition and Artificial Intelligence* 18, 3, 265–298.
- COOK, S. A. 1971. The complexity of theorem-proving procedures. In *Proc. 3rd ACM Symposium on Theory of Computing*. 151–158.
- CORDELLA, L. P., FOGGIA, P., SANSONE, C., AND VENTO, M. 2004. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26, 10, 1367–1372.

- CORNEIL, D. AND KIRKPATRICK, D. 1980. A theoretical analysis of various heuristics for the graph isomorphism problem. *SIAM Journal on Computing* 9, 2, 281–297.
- DAYLIGHT CHEMICAL INFORMATION SYSTEMS, INC. 2007. <http://www.daylight.com/dayhtml/doc/theory/theory.finger.html>. see Section 6.1.2.
- DE LA BRIANDAIS, R. 1959. File searching using variable length keys. In *Proc 1959 Western Joint Computer Conference*. 295–298.
- DURAND, P., LABARRE, L., MEIL, A., DIVO, J.-L., VANDENBROUCK, Y., VIARI, A., AND WOJCIK, J. 2006. Genolink: a graph-based querying and browsing system for investigating the function of genes and proteins. Open access at <http://www.biomedcentral.com/1471-2105/7/21>.
- FOGGIA, P., SANSONE, C., AND VENTO, M. 2001. A performance comparison of five algorithms for graph isomorphism. In *Proceedings of the 3rd IAPR-TC15 Workshop on Graph based Representation (Gbr2001)*. <http://www.amalfi.dis.unina.it/people/vento/lavori/gbr01bm.pdf>.
- FOWLER, G., HARALICK, R. M., GRAY, F. G., FEUSTEL, C., AND GRINSTEAD, C. 1983. Efficient graph automorphism by vertex partitioning. *Artif. Intell.* 21, 1-2, 245–269.
- GENT, I. P., MACINTYRE, E., PROSSER, P., SMITH, B. M., AND WALSH, T. 2001. Random constraint satisfaction: Flaws and structure. *Constraints* 6, 4, 345–372.
- GIUGNO, R. AND SHASHA, D. 2002. Graphgrep: A fast and universal method for querying graphs. *Pattern Recognition, 16th International Conference on* 2, 112–115.
- GOLOMB, S. W. AND BAUMERT, L. D. 1965. Backtrack programming. *J. ACM* 12, 4, 516–524.
- GOMES, C. P., SELMAN, B., CRATO, N., AND KAUTZ, H. 2000. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. Autom. Reason.* 24, 1-2, 67–100.
- HARALICK, R. M. AND ELLIOTT, G. 1980. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence* 14, 263–313.
- HASSAN, T. 2009. User-guided wrapping of pdf documents using graph matching techniques. In *10th International Conference on Document Analysis and Recognition*. IEEE Computer Society, Los Alamitos, CA, USA, 631–635.
- HOPCROFT, J. E. AND KARP, R. M. 1973. An  $n^{5/2}$  algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing* 2, 4, 225–231.
- HULUBEI, T. AND O’SULLIVAN, B. 2006. The impact of search heuristics on heavy-tailed behaviour. *Constraints* 11, 2-3, 159–178.
- JIANG, H., WANG, H., YU, P. S., AND ZHOU, S. 2007. GString: A novel approach for efficient search in graph databases. In *International Conference on Data Engineering*. IEEE Computer Society, Los Alamitos, CA, USA, 566–575.
- KLUKAS, C., KOSCHÜTZKI, D., AND SCHREIBER, F. 2005. Graph pattern analysis with pattern-gravisto. *J. Graph Algorithms Appl.* 9, 1, 19–29.
- KNUTH, D. E. 2000. Dancing links. in *Millennial Perspectives in Computer Science*, Eds: Davies, J., Roscoe, W. and Jim Woodcock, J. Palgrave, Houndmills, Basingstoke, Hampshire, UK, 187–214.
- KOHLER, E., MORRIS, R., AND CHEN, B. 2002. Programming language optimizations for modular router configurations. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. ACM, New York, NY, USA, 251–263.
- LARROSA, J. AND VALIENTE, G. 2002. Constraint satisfaction algorithms for graph pattern matching. *Mathematical Structures in Comp. Sci.* 12, 4, 403–422.
- LEACH, A. R. AND GILLET, V. J. 2003. *An Introduction to Chemoinformatics*. Kluwer Academic Publishers, Dordrecht, The Netherlands.
- LECOUTRE, C. 2008. Optimization of simple tabular reduction for table constraints. In *CP ’08: Proceedings of the 14th international conference on Principles and Practice of Constraint Programming*. Springer-Verlag, Berlin, Heidelberg, 128–143.
- LECOUTRE, C. 2009. *Constraint Networks: techniques and algorithms*. John Wiley and Sons, Hoboken, NJ, USA.
- LECOUTRE, C. AND VION, J. 2008. Enforcing arc consistency using bitwise operations. *Constraint Programming Letters* 2, 21–35.
- LYNCE, I. AND MARQUES-SILVA, J. P. 2003. An overview of backtrack search satisfiability algorithms. *Annals of Mathematics and Artificial Intelligence* 37, 3, 307–326.
- MACKWORTH, A. K. 1977. Consistency in networks of relations. *Artificial Intelligence* 8, 1, 99–118.
- MCGREGOR, J. J. 1979. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences* 19, 229–250.
- MCKAY, B. 2009. Nauty user’s guide (version 2.4). <http://cs.anu.edu.au/~bdm/nauty/nug.pdf>.
- MESSMER, B. T. AND BUNKE, H. 2000. Efficient subgraph isomorphism detection: A decomposition approach. *IEEE Trans. on Knowl. and Data Eng.* 12, 2, 307–323.
- ACM Journal Name, Vol. 15, No. 1, January 2011.

- NAANAA, W. 2009. A domain decomposition algorithm for constraint satisfaction. *J. Exp. Algorithmics* 13, 1.13–1.23.
- PROSCHAK, E., WEGNER, J. K., SCHÜLLER, A., SCHNEIDER, G., AND FECHNER, U. 2007. Molecular query language (mql)—a context-free grammar for substructure matching. *J. Chem. Inf. Model* 47, 2, 295–301.
- PROSSER, P. 1996. An empirical study of phase transitions in binary constraint satisfaction problems. *Artif. Intell.* 81, 1-2, 81–109.
- SABIN, D. AND FREUDER, E. 1994. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, PPCP'94*, A. Borning, Ed. Lecture Notes in Computer Science, Volume 874, Springer Verlag, 10–20.
- SABIN, D. AND FREUDER, E. 1997. Understanding and improving the MAC algorithm. In *Proceedings of CP'97, published as Principles and Practice of Constraint Programming, Lecture Notes in Computer Science, Volume 1330/1997*, G. Smolka, Ed. Springer Verlag, 167–181.
- SCHULTE, C. 1999. Comparing trailing and copying for constraint programming. In *Proceedings of the 1999 international conference on Logic programming (ICLP'99)*. Massachusetts Institute of Technology, Cambridge, MA, USA, 275–289.
- SHANG, H., ZHANG, Y., LIN, X., AND YU, J. X. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. In *VLDB '08: Proceedings of the 34th international conference on very large data bases*. VLDB Endowment, 364–375.
- SHASHA, D., WANG, J. T. L., AND GIUGNO, R. 2002. Algorithmics and applications of tree and graph searching. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, New York, NY, USA, 39–52.
- SMITH, B. M. AND DYER, M. 1996. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence* 81, 155–181.
- SOLNON, C. 2010. AllDifferent-based filtering for subgraph isomorphism. *Artif. Intell.* 174, 12-13, 850–864.
- TARJAN, R. AND YANNAKAKIS, M. 1984. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. on Computing* 13, 566–579.
- ULLMANN, J. R. 1965. Parallel recognition of idealized line characters. *Kybernetik* 2, 5, 221–226. <http://www.visionbib.com/papers/1965/Kybernetik65.pdf>; Short introduction: <http://www.visionbib.com/papers/1965/Kybintro.pdf>.
- ULLMANN, J. R. 1976. An algorithm for subgraph isomorphism. *J. ACM* 23, 1, 31–42.
- ULLMANN, J. R. 1977. A binary n-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words. *Computer Journal* 20, 2, 141–147.
- ULLMANN, J. R. 2007. Partition search for non-binary constraint satisfaction. *Inf. Sci.* 177, 18, 3639–3678.
- WALLACE, R. J. AND FREUDER, E. C. 1992. Ordering heuristics for arc consistency algorithms. In *AI/GI/VI 92: Proceedings of the Ninth Canadian Conference on Artificial Intelligence*. 163–169.
- WILLETT, P. 1999. *Matching of chemical and biological structures using subgraph and maximal common subgraph isomorphism algorithms*. Springer Verlag, New York, USA, 11–38. in *Rational Drug Design*, Eds: Truhlar, D.G., Howe, W.J., Hopfinger, A.J., Blaney, J.D. and Dammkoehler, R.
- WILLETT, P. 2005. Chemoinformatics techniques for data mining in files of two-dimensional and three-dimensional chemical molecules. In *Proceedings of the Third Conference on the Foundations of Information Science*, M. Petitjean, Ed. MDPI, Basel. <http://www.mdpi.org/fis2005/proceedings.html>.
- WILLETT, P. 2008. From chemical documentation to chemoinformatics: 50 years of chemical information science. *J. Inf. Sci.* 34, 4, 477–499.
- WILLETT, P., BARNARD, J. M., AND DOWNS, G. M. 1998. Chemical similarity searching. *Journal of Chemical Information and Computer Sciences* 38, 6, 983–996.
- YAN, X., YU, P. S., AND HAN, J. 2005. Graph indexing based on discriminative frequent structure analysis. *ACM Trans. Database Syst.* 30, 4, 960–993.
- YAN, X., ZHU, F., YU, P. S., AND HAN, J. 2006. Feature-based similarity search in graph structures. *ACM Trans. Database Syst.* 31, 4, 1418–1453.
- ZAMPELLI, S., DEVILLE, Y., AND SOLNON, C. 2010. Solving subgraph isomorphism problems with constraint programming. *Constraints* 15, 3, 327–353.
- ZHANG, S., LI, S., AND YANG, J. 2009. Gaddi: distance index based subgraph matching in biological networks. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*. ACM, New York, NY, USA, 192–203.

- ZHAO, P., YU, J. X., AND YU, P. S. 2007. Graph indexing: tree + delta  $\geq$  graph. In *Vldb '07: Proceedings of the 33rd international conference on Very Large Data Bases*. VLDB Endowment, 938–949.
- ZOBEL, J., MOFFAT, A., AND RAMAMOCHANARAO, K. 1998. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst.* 23, 4, 453–490.
- ZOU, L., CHEN, L., ZHANG, H., LU, Y., AND LOU, Q. 2008. Summarization graph indexing: Beyond frequent structure-based approach. In *Database Systems for Advanced Applications (DASFAA)*. 141–155.