Filtering Algorithms Based on the Word-RAM Model

Philippe Van Kessel
Université Laval
Département d'informatique et de génie logiciel
philippe.van-kessel.1@ulaval.ca

Claude-Guy Quimper
Université Laval
Département d'informatique et de génie logiciel
claude-guy.quimper@ift.ulaval.ca

Abstract

The Word-RAM is a model of computation that takes into account the capacity of a computer to manipulate a word of w bits with a single instruction. Many modern constraint solvers use a bitset data structure to encode the values contained in the variable domains. Using the algorithmic techniques developed for the Word-RAM, we propose new filtering algorithms that can prune $O(w)$ values from a domain in a single instruction. Experiments show that on a processor with $w = 64$, the new filtering algorithms that enforce domain consistency on the constraints $A + B = C$, $|A - B| = C$ and ALL-DIFFERENT can offer a speed up of a factor 10.

Introduction

Constraint solvers derive a large part of their efficiency from the constraint propagation process. During this process, the filtering algorithm associated to each constraint excludes partial solutions that cannot be completed into complete solutions. Since these algorithms are called millions of times during the search, it is essential to find the best way to implement them.

The running time analysis is one way to predict the efficiency of an algorithm since it reveals how the computation time grows as the size of the instance increases. This analysis is usually based on a theoretical computer called *random access machine* (RAM) that shares properties with real computers. However, one property that the RAM does not take into account is that modern processors can handle words of $w = 64$ bits with a single instruction. Such a property is considered in the *word random access machine* called Word-RAM (see (Hagerup 1998)). Analyzing an algorithm on the Word-RAM leads to a running time complexity that is a function of w, the size of a word. Changing the way of analyzing algorithms changes the way of designing them. We propose new filtering algorithms designed for the Word-RAM.

The paper is organized as follows. We present the theoretical foundations of the Word-RAM. We then present new filtering algorithms for the constraints $A + B = C$, $|A - B| = C$, $A = f(B)$ where f is any bijective function, and the global constraint ALL-DIFFERENT. We then empirically show that these new algorithms improve the efficiency of the solver.

The Word-RAM

The Theory

A *model of computation* is a set of instructions of an abstract machine and the cost of execution for each of these instructions. Such a model allows to compute the complexity of an algorithm, i.e. the asymptotic cost of an execution when the size of the instance tends to infinity. The *random access machine* (RAM) is the most commonly used model to analyze an algorithm. A RAM is able to perform basic arithmetic operations and comparisons ($+$, $-$, $\times$, $\div$, $\wedge$, $\vee$, $<$, $=$, $>$, etc.) and to randomly access the memory such as changing the value of the i^{th} element of a vector, all in constant time.

A *Word-RAM* is a RAM which is able to perform bitwise operations on a word of w bits. Among the bitwise operations, the operator $a \ll b$ returns the bits in a shifted to the left by b bits. This is equivalent to computing $a2^b$. The operation $a \gg b$ returns the bits in a shifted to the right by b bits. This is equivalent to computing $\lfloor a2^{-b} \rfloor$. The binary operators $\&$, $|$ and $\oplus$ perform a logical conjunction, a logical disjunction, and an exclusive disjunction between each pair of bits. The unary operator $\sim$ negates every bit in a word. The unary operators LSB and MSB return the index of the least significant bit and the most significant bit in a word.

The Word-RAM is a realistic model since modern processors support all of these operations. Even though the operators LSB and MSB are not always integrated to high-level languages such as C, the C compiler *gcc* provides the built-in functions `__builtin_ctz` and `__builtin_clz` that are equivalent to the operators LSB and MSB.

With its bitwise operators, a Word-RAM with words of w bits can manipulate up to w elements of the input in constant time. This leads to a theoretical gain of up to a factor w. The size of a word in modern processors is currently $w = 64$. However, special sets of instructions such as SSE4 in Intel processors allow to perform special operations on registers of 128 bits. In early 2011, Intel released the Sandy Bridge processors with the *Advanced Vector Extensions* (AVX), a set of instructions manipulating 256-bit registers.

Analyzing the running time complexity on the Word-RAM leads to new ways of designing algorithms. We

present existing algorithms based on the Word-RAM model that are already used in constraint solvers and then present new filtering algorithms.

The Word-RAM Model in Constraint Solvers

Constraint solvers exploit the Word-RAM model in different ways. As early as 1979, (McGregor 1979) uses bitsets to encode the domains of the variables in a constraint satisfaction problem. He then uses bitwise operations & and | to compute the intersection and the union of the domains with sets of values. (Haralick & Elliott 1980) use the same strategy to implement an efficient version of forward checking. (Lecoutre & Vion 2008) shows how this data structure can be used to obtain a very efficient implementation of the AC3 algorithm that achieves arc-consistency.

Most of these algorithms share a common approach: the variable domains are encoded with bitsets. A bitset is a sequence of bits $a_{d-1}a_{d-2}\dots a_0$ where each bit a_v is associated to the value a_v. A bit is set to one if its corresponding value belongs to the domain and is set to zero if it does not belong to the domain. We denote by $\mathrm{dom}(X)$ the bitset representing the domain of the variable X. The pruning of the domains can be done by computing a bitset s representing the set of values that can be assigned to a variable while satisfying the constraints. To filter the domain of X, one needs to compute the intersection between the set of values s and the domain $\mathrm{dom}(X)$. The bitwise operator & achieves this computation and processes up to w values in a single operation.

Preliminaries

A set of values A denotes at the same time: the set of values it contains, its bitset representation, and the integer whose binary encoding is the same as the bit set. Thus the set $A = \{1, 5, 7\}$ is equivalent to the bitset $a_7a_6a_5a_4a_3a_2a_1a_0 = 10100010$, where the least significant bit (a_0) is associated to the value 0 and the most significant bit (a_7) is associated to the value 7. The integer 162 represents as well the set A. We consider bitsets of arbitrary length d and assume that any bitwise operator can be implemented with a time complexity $O\left(d/w\right)$ where w is the size of a word.

A support to the constraint $C(X_1, \dots, X_n)$ is a tuple $(t_1, \dots, t_n)$ that satisfies the constraint C such that $t_i \in \mathrm{dom}(X_i)$ for all $i = 1..n$. A support for a value $v \in \mathrm{dom}(X_i)$ is a support such that $t_i = v$. Enforcing domain consistency consists of removing from all domains the values that do not have a support.

A New Filtering Algorithm for the Constraint SUM

The constraint SUM restricts the sum of two variables to be equal to a third variable, i.e. $A + B = C$. We present an algorithm that enforces domain consistency on this constraint. To understand the algorithm, one needs to understand the properties of the shift operators $\ll$ and $\gg$. Consider the bitset 10100010 representing the set $\{1, 5, 7\}$. The bitset set $10100010 \ll 1 = 101000100$ represents the set $\{2, 6, 8\}$. In other words, by shifting the bitset by one unit, all elements in the original set are incremented by one. This relation holds.

$$A \ll k = \{a + k \mid a \in A\} \tag{1}$$

The same result applies for the operator $\gg$, although this operator truncates bits that are shifted to a negative position. For instance, the operation $10100010 \gg 3 = 10100$ modifies the set $\{1, 5, 7\}$ to obtain $\{2, 4\}$. This relation holds.

$$A \gg k = \{a - k \mid a \in A \wedge a \geqslant k\} \tag{2}$$

The shift operators are very convenient to add a constant to all elements in a set. Algorithm 1 extensively uses this property to enforce domain consistency on the constraint $A + B = C$. The *while* loop on line 1 iterates through all the values in the domain of variable A. It extracts the smallest value $v \in \mathrm{dom}(A)$ on line 2, creates a bitset representing the set $\{v\}$ on line 3, and removes v from $\mathrm{dom}(A)$ on line 4 to prevent a second iteration on the value v. This way of iterating through the values of a domain is reused in the next algorithms. It turns out to be the most effective way in practice since it retrieves both the processed value v and the bitset representing the set $\{v\}$.

Algorithm 1: FilterAPlusBEqualsC(A, B, C)

```
   newA ← dom(A), newB ← 0, newC ← 0
1  while dom(A) ≠ 0 do
2      v ← LSB(dom(A))
3      bit ← 1 ≪ v
4      dom(A) ← dom(A) ⊕ bit
5      shiftedB ← dom(B) ≪ v
       if shiftedB & dom(C) ≠ 0 then
6          newC ← newC | shiftedB
7          newB ← newB | (dom(C) ≫ v)
8      else newA ← newA ⊕ bit
       dom(A) ← newA
       dom(B) ← dom(B) & newB
9      dom(C) ← dom(C) & newC
```

This relation prunes the domain of C.

$$\mathrm{dom}'(C) = \mathrm{dom}(C) \cap \bigcup_{v \in \mathrm{dom}(A)} \{b + v \mid b \in \mathrm{dom}(B)\} \tag{3}$$

The computation of the set $\{b + v \mid b \in \mathrm{dom}(B)\}$ is performed on line 5 using the operator $\ll$, the union is performed on line 6 with the operator |. The union is performed only if $\{b+v \mid b \in \mathrm{dom}(B)\}$ intersects the domain $\mathrm{dom}(C)$. Even though this test is not required, it prevents useless computations of the union. The intersection with the domain of C is performed on line 9.

The algorithm filters the domain of A by testing for the existence of a support for each value v. If the set $\{b + v \mid b \in \mathrm{dom}(B)\}$ intersects the domain of the variable C, then there is a value in $\mathrm{dom}(B)$ whose sum with v produces a value in $\mathrm{dom}(C)$ and therefore v has a support in A. If the set $\{b + v \mid b \in \mathrm{dom}(B)\}$ does not intersect the domain of C, then v must be removed from the domain of A (line 8).

This relation prunes the domain of B.

$$\mathrm{dom}'(B) = \mathrm{dom}(B) \cap \bigcup_{v \in \mathrm{dom}(A)} \{c - v \mid c \in \mathrm{dom}(C)\} \quad (4)$$

The union can be computed only over the values v that have a support in A since any value v that does not have a support will produce a set $\{c - v \mid c \in \mathrm{dom}(C)\}$ that is disjoint from $\mathrm{dom}(B)$. Since we assume that all values in the domains are non-negative, the set $\{c - v \mid c \in \mathrm{dom}(C)\}$ can be replaced by $\{c - v \mid c \in \mathrm{dom}(C) \wedge c \geqslant v\}$. This is computed on line 7.

Running Time Analysis

If all domains are contained in an interval $[0, d]$, they can be encoded with bitsets of $\lceil (d+1)/w \rceil$ words. All bitset operations in Algorithm 1 are thus executed in time $O\left(d/w\right)$. The while loop executes exactly $|\mathrm{dom}(A)|$ times. The total running time complexity of the algorithm is therefore $O\left(d|\mathrm{dom}(A)|/w\right)$. Note that this complexity does not depend on the cardinality of $\mathrm{dom}(B)$ nor the cardinality of $\mathrm{dom}(C)$. If $d < w$, each domain fits in a single word. In this special case, we obtain a linear time complexity in $|\mathrm{dom}(A)|$. Finally, the constraint $A + B = C$ is equivalent to $B + A = C$. The algorithm can be adapted to iterate on the domain with the smallest cardinality giving a complexity of $O\left(\min(|\mathrm{dom}(A)|, |\mathrm{dom}(B)|)d/w\right)$. This is particularly significant when A or B is fixed to a value.

Extensions

A linear constraint of the form $\sum_{i=1}^{n} X_i = k$ can be decomposed into a set of constraints $Y_i = Y_{i-1} + X_i$ with $Y_0 = 0$ and $Y_n = k$. Such a decomposition does not hinder propagation since the constraint hypergraph is a hypertree. Algorithm 1 enforces domain consistency on such a decomposition in time $O\left(nd^2/w\right)$ (assuming $|\mathrm{dom}(X_i)| = d$). This is potentially w times faster than the algorithm developed by (Trick 2003) for the same constraint.

One can use the constraint SUM to encode a constraint of difference $A - B = C$. Indeed, rather than posting the constraint $A - B = C$ on the variables, one can simply post the equivalent constraint $B + C = A$ which is a sum.

We assumed that the domains do not contain negative values. One can lift this assumption by associating the least significant bit of all domains to the negative value $m < 0$. Line 2 can be replaced by $v \leftarrow m + \mathrm{LSB}(\mathrm{dom}(A))$ and the algorithm remains correct.

A New Filtering Algorithm for the Absolute Difference Constraint

We consider the constraint $|A - B| = C$, i.e. C is constrained to be the absolute difference between A and B. We assume that the variable domains contain non-negative values. The algorithm exploits this relation.

$$|A - B| = C \iff B + C = A \vee B - C = A \quad (5)$$

Algorithm 2 iterates through all the values v in $\mathrm{dom}(C)$. Line 1 computes the set $B^{+}_{-} = \{b \pm v \mid b \in \mathrm{dom}(B)\}$. If the set B^{+}_{-} does not intersect the domain of A, there is no value $b \in \mathrm{dom}(B)$ and value $a \in \mathrm{dom}(A)$ such that $|a - b| = v$ and therefore v does not have a support in $\mathrm{dom}(C)$. Thus, line 4 removes v from $\mathrm{dom}(C)$.

Algorithm 2: FilterAbsoluteValue(A, B, C)

```
   newA ← 0, newB ← 0, newC ← dom(C)
   while dom(C) ≠ 0 do
       v ← LSB(dom(C))
       bit ← 1 ≪ v
       dom(C) ← dom(C) ⊕ bit
1      B⁺₋ ← (dom(B) ≪ v) | (dom(B) ≫ v)
       if B⁺₋ & dom(A) ≠ 0 then
2          newA ← newA | B⁺₋
3          newB ← newB | (dom(A) ≪ v) | (dom(A) ≫ v)
4      else newC ← newC ⊕ bit
5  dom(A) ← dom(A) & newA
6  dom(B) ← dom(B) & newB
   dom(C) ← newC
```

This relation prunes the domain of variable A.

$$\mathrm{dom}'(A) = \mathrm{dom}(A) \cap \bigcup_{v \in \mathrm{dom}(C)} \{b \pm v \mid b \in \mathrm{dom}(B)\} \quad (6)$$

Line 2 computes the union in Equation (6). The values v that do not have a support in C are omitted in the computation of the union since B^{+}_{-} and $\mathrm{dom}(A)$ are disjoint. Line 5 computes the intersection of Equation (6).

This relation prunes the domain of variable B.

$$\mathrm{dom}'(B) = \mathrm{dom}(B) \cap \bigcup_{v \in \mathrm{dom}(C)} \{a \pm v \mid v \in \mathrm{dom}(C)\} \quad (7)$$

Like the computation of the domain of A, the values v that do not have a support in $\mathrm{dom}(C)$ can be omitted since they produce sets that are disjoint from $\mathrm{dom}(B)$. Line 3 computes the union of the sets $\{a \pm v \mid v \in \mathrm{dom}(C)\}$. Line 6 computes the intersection in Equation (7).

Running Time Analysis

The while loop of Algorithm 2 executes exactly $|\mathrm{dom}(C)|$ times. Let d be the largest value in the domains. All operations execute in time $O\left(d/w\right)$. We therefore obtain a running time complexity of $O\left(|\mathrm{dom}(C)|d/w\right)$. When all domains fit in a single word, we obtain a linear time complexity in $|\mathrm{dom}(C)|$.

A New Filtering Algorithm for Bijective Constraints

We consider the problem of enforcing domain consistency on a constraint $A = f(B)$ where f is a bijective function, i.e. a function with an inverse f' such that $A = f(B) \iff f'(A) = B$. Such a constraint is domain consistent if the bit at position v in $\mathrm{dom}(B)$ is set to one if and only if the bit at position $f(v)$ in $\mathrm{dom}(A)$ is also set to one.

(Knuth 2008) defines a δ-right-shift as an operation that takes for input a bitset $x_{n-1} \dots x_1 x_0$, an integer δ and a mask $\theta_{n-1}, \dots, \theta_0$. The operation returns a bitset $x'_{n-1} \dots x'_1 x'_0$ such that $x'_i = x_{i+\delta}$ if $\theta_i = 1$ and $x'_i = x_i$ otherwise. The δ-right-shift is computed as follows.

$$x' = x \oplus ((x \oplus (x \gg \delta)) \,\&\, \theta) \tag{8}$$

Similarly, the δ-left-shift is computed as follows.

$$x' = x \oplus ((x \oplus (x \ll \delta)) \,\&\, \theta) \tag{9}$$

A *compression* is an operation that takes as input a bitset $a_{n-1}a_{n-2} \dots a_0$ and a mask $m_{n-1}m_{n-2} \dots m_0$ and returns a bitset $0 \dots 0 a_{j_{k-1}} a_{j_{k-2}} \dots a_{j_0}$ such that the indices $j_{k-1} > j_{k-2} > \dots > j_1 > j_0$ are the positions of the k bits in the mask m that are set to one. For instance, consider the bitset $a_7a_6a_5a_4a_3a_2a_1a_0$ and the mask 10011010, the compression operation returns $0000a_7a_4a_3a_1$. The compression of a bitset of 2^p bits is done with $p+1$ consecutive δ-right-shift operations where $\delta = 2^0, 2^1, \dots, 2^p$. For instance, using the masks $\theta^1 = 00000001$, $\theta^2 = 0000110$, $\theta^4 = 00001000$, $\theta^8 = 11110000$ the δ-right-shift operations produce the bitsets $a_7a_6a_5a_4a_3a_2a_1a_0$, $a_7a_6a_5a_4a_3a_2a_1a_1$, $a_7a_6a_5a_4a_3a_4a_3a_1$, $a_7a_6a_5a_4a_7a_4a_3a_1$ and $0000a_7a_4a_3a_1$. (Knuth 2008) shows how to compute the $p+1$ masks $\theta^1, \theta^2, \dots$ in time $O(p^2)$. This computation can take place at compilation time. The inverse of a compression is computed with δ-left-shift operations executed in reverse order, i.e. $\delta = 2^p, 2^{p-1}, \dots, 1$.

If the function f is increasing and defined on all natural numbers, then the compression operation maps the bits in $\mathrm{dom}(A)$ to the bits in $\mathrm{dom}(B)$. To illustrate the idea, consider the constraint $A = 2B$. Suppose that $\mathrm{dom}(A) = \{0, 4, 6\} = 1010001$ and $\mathrm{dom}(B) = \{0, 2, 3\} = 0001101$. The compression of $\mathrm{dom}(A)$ with the mask 1010101 maps the bit in $\mathrm{dom}(A)$ to the bits in $\mathrm{dom}(B)$.

To enforce domain consistency on the constraint $A = f(B)$, one needs to compute (at compilation) the mask m that maps the bit in A to the bits in B. We then set to zero all bits in $\mathrm{dom}(A)$ that are zero in the compression of B.

$$\mathrm{dom}'(A) = \mathrm{dom}(A) \,\&\, \mathit{compression}(\mathrm{dom}(B), m) \tag{10}$$

The domain $\mathrm{dom}(B)$ can be filtered using the inverse of a compression.

$$\mathrm{dom}'(B) = \mathrm{dom}(B) \,\&\, \mathit{compression}^{-1}(\mathrm{dom}(A), m) \tag{11}$$

Let d be the largest value in the domains. A δ-shift is computed in time $O(d/w)$ and there are $O(\log d)$ such shifts to compute. Enforcing domain consistency on an increasing function is done in $O((d \log d)/w)$ steps. When the domains fit on a single word ($d < w$), we obtain an algorithm running in $O(\log d)$. This is particularly efficient when domains are dense.

Finally, we consider the case where the function f is bijective but is not necessarily increasing. In such a case, the function f defines a permutation where each bit b_i in $\mathrm{dom}(B)$ is associated to the bit $a_{f(i)}$ in $\mathrm{dom}(A)$. To filter this constraint, we permute the bits $b_{d-1}b_{d-2} \dots b_0$ representing $\mathrm{dom}(B)$ to obtain $b' = b_{f(d-1)}b_{f(d-2)} \dots b_{f(0)}$. We then compute $\mathrm{dom}'(A) = \mathrm{dom}(A) \,\&\, b'$. We proceed similarly to prune the domain of B using the inverse permutation f^{-1}. Filtering the domains is therefore reduced to permuting the bits in a bitset. (Knuth 2008) defines the operation δ-*swap* with mask θ that swaps the bits x_i with $x_{i+\delta}$ if the bit at position i in θ is set. Permuting the bitset $x_7x_6x_5x_4x_3x_2x_1x_0$ with $\delta = 3$ and $\theta = 00010101$ produces $x_4x_6x_2x_7x_0x_5x_1x_3$. This operation is computed as follows.

$$y \leftarrow (x \oplus (x \gg \delta)) \,\&\, \theta \tag{12}$$

$$x' \leftarrow x \oplus y \oplus (y \ll \delta) \tag{13}$$

(Knuth 2008) credits (Duguid 1959) and (LeCorre 1959) who proved that $O(\log d)$ δ-swap operations are sufficient to obtain any permutation of d bits. Thus enforcing domain consistency on a bijective function can be done in $O((d \log d)/w)$ steps and in $O(\log d)$ steps if each domain entirely fits in a word ($d < w$).

A New Filtering Algorithm for the All-Different Constraint

Global constraints can also have filtering algorithms designed for the Word-RAM. It is the case for the constraint ALL-DIFFERENT$(X_1, \dots, X_n)$ that is satisfied when all variables are pairwise distinct. (Régin 1994) published an algorithm that enforces domain consistency on this constraint. His algorithm processes in three steps: 1) it computes a maximum matching in a graph called the *value graph*, 2) it computes the strongly connected components of the residual graph, and 3) it filters the domains depending on which parts of the residual graph is strongly connected and which nodes lie on a path ending with an unmatched node.

The computation of a maximum matching and the strongly connected components of the graph heavily relies on graph traversal algorithms such as depth-first search. (Cheriyan & Mehlhorn 1996) show how to speed-up the traversal of a graph using a Word-RAM. Two data structures are required to perform a depth-first search: a data structure providing the set of neighbors of a node and a set of nodes that have previously been reached. The neighbors of a node a can be encoded in a bitset $N(a)$. The reached nodes can also be stored in a bitset R. The operations $N(a) \,\&\, \sim R$ return the set of neighbors of node a that have not been reached. If this set is non-empty, the operations LSB$(N(a) \,\&\, \sim R)$ return a neighbor that has not been reached yet. These operations are done in time $O(d/w)$ where d is the number of nodes in the graph. If the graph is dense, this operation is significantly faster than iterating over each element in $N(a)$ to test whether the node has been reached. Using these data structures, (Cheriyan & Mehlhorn 1996) show how to perform a depth-first search, a breadth-first search, and the computation of the strongly connected components of a graph with n nodes all in time $O(n^2/w)$ which is advantageous for dense graphs[1]. (Hopcroft & Karp

[1]They actually reported a complexity of $O(nw + n^2/w)$. We save the term nw by using the operator LSB which Cheriyan and Mehlhorn did not consider in their Word-RAM.

1973) show that performing $O(\sqrt{n})$ breadth-first searches is sufficient to compute a maximum matching. (Cheriyan & Mehlhorn 1996) reuse this result and show how to compute a maximum matching in time $O\left(n^{2.5}/w\right)$.

The *value-graph* $G = \langle V, E\rangle$ as defined by Régin has one vertex per variable and one vertex per value, i.e. $V = \{X_i \mid i = 1..n\} \cup \bigcup_{i=1}^{n} \mathrm{dom}(X_i)$. There is an edge from a variable node X_i to a value node v iff $v \in \mathrm{dom}(X_i)$. Any matching of n edges is a support for the constraint. There is no need to construct the value-graph since all the information is contained in the domains which are already encoded with bitsets. Indeed, the neighbors of the variable node X_i are encoded in the bitset $\mathrm{dom}(X_i)$. This encoding makes it easy to reuse the graph algorithms designed for the Word-RAM. Moreover, value-nodes have at most one neighbor in the residual graph. It is therefore possible to test, in constant time, whether a value-node has a neighbor that has not been reached yet.

The search for an augmenting path in the residual graph usually stops when visiting a free node, i.e. a node that is not adjacent to an edge in the matching. Similarly, when flagging the strongly connected components that can reach a free node, it is sufficient to see whether a variable-node X_i is adjacent to a free node without visiting all the free nodes. This operation can be done by keeping a bitset F of the free value-nodes. It becomes possible to efficiently test whether a variable-node X_i is adjacent to a free node with the operations $\mathrm{dom}(X_i)\,\&\,F \neq 0$.

Using the data structures and algorithms described above, one can obtain an implementation of Régin's algorithm running in time $O\left(n^{2.5}/w\right)$. Following Régin, the algorithm can be further improved by making it incremental. Rather than recomputing the maximum matching from scratch each time the filtering algorithm is called, one can reuse the last computed matching and adapt it to the new value-graph. This leads to an incremental complexity of $O\left(\delta n^2/w\right)$ where δ is the number of values removed from the variable domains since the last execution of the filtering algorithm.

Experiments

We implement the filtering algorithms enforcing domain consistency on the constraints $A+B=C$, $|A-B|=C$ and ALL-DIFFERENT in the constraint solver of the C++ Google or-tools library (Google 2011). This solver uses the bitset data structure to encode the variable domains and is optimized to run in 64-bit mode ($w = 64$). We slightly modify the solver to allow the filtering of multiple values in a single instruction (for instance, see line 9 of Algorithm 1).

We have three different implementations of a filtering algorithm enforcing domain consistency on the constraint $A+B=C$. The first implementation, denoted $\text{SUM}_{WordRam}$, is based on Algorithm 1 with a running time complexity of $O(\min(|\mathrm{dom}(A)|, |\mathrm{dom}(B)|)d/w)$. The second implementation, denoted SUM_{Table}, uses the constraint POSITIVETABLECONSTRAINT available in the or-tools. This constraint takes an enumeration of all triplets (a, b, c) satisfying $a+b=c$ and enforces domain consistency. Finally, we implement a brute force algorithm, denoted $\text{SUM}_{BruteForce}$, that tests, for each pair in $\{(a, b) \mid a \in \mathrm{dom}(A), b \in \mathrm{dom}(B)\}$, whether the triplet $(a, b, a+b)$ is a support for the constraint. Each time a support is found, the values forming this support are flagged. Unflagged values are then removed from the domains. The algorithm runs in time $O\left(|\mathrm{dom}(A)||\mathrm{dom}(B)| + |\mathrm{dom}(C)|\right)$.

We implement the filtering algorithm that enforces domain consistency on the constraint $|A-B|=C$ based on Algorithm 2. We denote this implementation $\text{ABS}_{WordRam}$. The second implementation uses the table constraint and filters the domain based on an enumeration of all triplets (a, b, c) satisfying $|a-b|=c$. We denote this implementation ABS_{Table}.

Since the or-tools do not provide an algorithm that enforces domain consistency on the ALL-DIFFERENT, we implement two versions of the algorithm. All versions reuse the techniques that were tested to be the most efficient according to (Gent, Miguel, & Nightingale 2008). The computation of the maximum matching is incremental. The algorithms are only executed on the components of the graph affected by the removal of a value in the domains. We use a breadth-first search to compute the augmenting paths. The first version we implement, denoted ALL-DIFFERENT, does not use the potential that offers the word-RAM. The second version uses the techniques described in the previous section and is denoted $\text{ALL-DIFFERENT}_{WordRam}$.

All experiments are run on a 64-bit Intel Processor i7 3.4GHz with 4Gb of memory. Since all the filtering algorithms enforce domain consistency, we obtain the same number of backtracks for a given instance. We report the number of backtracks but the computation time is the only point of comparison. We use a timeout of 10 minutes for each instance and report the best time out of three runs.

The n-queen Problem

The n-queen problem consists of placing n queens on a $n \times n$ chessboard such that two queens do not appear on the same row, same column, or same diagonal. Let the queen on column i be on row X_i. We set $A_i = X_i + i$ and $B_i = X_i + (n-i)$ for $i = 1..n$. We impose three constraints ALL-DIFFERENT, one on the variables X_i, one on the variables A_i, and one on the variables B_i.

Table 1 shows that the algorithm $\text{ALL-DIFFERENT}_{WordRam}$ improves the times by 25% over the algorithm ALL-DIFFERENT. Similarly, the algorithm $\text{SUM}_{WordRam}$ is 30% faster than SUM_{Table}. Together, the new algorithms cut the execution times by 45%.

The Magic Square Problem

The magic square consists of filling in a $n \times n$ grid with the integers from 1 to n^2 such that all rows, all columns, and both diagonals sum to the same number. We model this problem with one constraint ALL-DIFFERENT and $2n+2$ linear constraints. Each linear constraint is decomposed into $n-1$ constraints $A+B=C$ for a total of $O(n^2)$ constraints $A+B=C$.

For these problems, the table constraint SUM_{Table} is unable to compete with the constraint $\text{SUM}_{BruteForce}$ because it consumes too much memory. We use a restart strategy in

n-queen					
		ALL-DIFFERENT		ALL-DIFFERENT$_{WordRam}$	
n	bt	SUM$_{Table}$	SUM$_{WordRam}$	SUM$_{Table}$	SUM$_{WordRam}$
9	208	14	11	11	8
10	686	48	38	37	26
11	2940	210	163	157	112
12	13450	972	759	737	526
13	65677	4827	3782	3657	2610
14	344179	25842	20199	19464	13798
15	1948481	149567	116567	111822	80002
Magic Square					
		ALL-DIFFERENT		ALL-DIFFERENT$_{WordRam}$	
n	bt	SUM$_{BruteForce}$	SUM$_{WordRam}$	SUM$_{BruteForce}$	SUM$_{WordRam}$
5	782	613	89	603	61
6	1535	2953	330	2931	238
7	2584	10654	1003	10551	748
8	4336	36301	3501	36220	2844
9	8211	119710	11228	117856	8849
10	23902	596705	46818	587675	37781
11	41857	-	109521	-	90062
Golomb Ruler					
		ALL-DIFFERENT		ALL-DIFFERENT$_{WordRam}$	
n	bt	SUM$_{BruteForce}$	SUM$_{WordRam}$	SUM$_{BruteForce}$	SUM$_{WordRam}$
6	39	8	4	8	2
7	207	44	22	37	16
8	1284	275	175	228	127
9	5980	1823	1286	1538	990
10	33318	12976	10380	11037	8484
11	553793	309715	275332	276345	241827
All-Interval					
		ALL-DIFFERENT		ALL-DIFFERENT$_{WordRam}$	
n	bt	ABS$_{Table}$	ABS$_{WordRam}$	ABS$_{Table}$	ABS$_{WordRam}$
9	855	24	14	18	10
10	2903	93	56	69	37
11	10335	366	216	268	140
12	39270	1555	891	1131	578
13	155792	6823	3838	4857	2480
14	656435	31116	17351	22443	10967
15	2886750	146681	80817	105740	50960
16	13447418	-	402566	522795	251246

Table 1: Execution times, in milliseconds, to solve an instance of size n. The column *bt* reports the number of backtracks.

order to solve larger instance. Most of the computation time is spent in filtering the linear constraints. Table 1 shows that SUM$_{WordRam}$ cuts the computation time by 92% on some instances compared to the algorithm SUM$_{BruteForce}$. The saving offered by ALL-DIFFERENT$_{WordRam}$ is not significant since most of the time is spent in filtering the linear constraints.

The Golomb Ruler Problem

The Golomb ruler problem consists of finding an increasing sequence $X_1, \ldots, X_n$ with minimal X_n such that the differences $D_{ij} = X_j - X_i$ are all distinct. We use one constraint ALL-DIFFERENT over the variables D_{ij} and $O(n^2)$ constraints $X_i + D_{ij} = X_j$. Table 1 shows that ALL-DIFFERENT$_{WordRam}$ offers a gain ranging from 10% to 27% compared to ALL-DIFFERENT. SUM$_{WordRam}$ improves the time up to 35% compared to SUM$_{BruteForce}$. Together the word-RAM algorithms cut the computation time by 75% on the small instances to 22% on the larger instances.

The All-Interval Problem

The all-interval problem consists of finding a sequence $X_1, \ldots, X_n$ such that the differences $D_i = |X_i - X_{i-1}|$ are all distinct. We use a single constraint ALL-DIFFERENT and $n-1$ constraints of absolute difference. Table 1 shows that ALL-DIFFERENT$_{WordRam}$ is from 28% to 37% faster than ALL-DIFFERENT, and that the use of ABS$_{WordRam}$ can cut in half the times compared to ABS$_{Table}$. When combined together, the word-RAM algorithms reduce the computation time by 65%.

Conclusion

Designing filtering algorithms for the Word-RAM leads to significant gains in efficiency. The bitset data structure used

to encode variable domains is one way to exploit the Word-RAM model. It would be interesting to try these algorithms with the Intel AVX instructions that manipulate words of 256 bits.

References

Cheriyan, J., and Mehlhorn, K. 1996. Algorithms for dense graphs and networks on the random access computer. *Algorithmica* 15:521–549.

Duguid, A. M. 1959. Structural properties of switching networks. Technical Report BTL-7, Brown University.

Gent, I. P.; Miguel, I.; and Nightingale, P. 2008. Generalised arc consistency for the alldifferent constraint: An empirical survey. *Artificial Intelligence* 172(18):1973–2000.

Google. 2011. http://code.google.com/p/or-tools/.

Hagerup, T. 1998. Sorting and searching on the word ram. In *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science (STAC'98)*, 366–398.

Haralick, R. M., and Elliott, G. L. 1980. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence* 14(3):263–313.

Hopcroft, J., and Karp, R. 1973. A $n^{\frac{5}{2}}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal of Computing* 2:225–231.

Knuth, D. E. 2008. *The Art of Computer Programming, Pre-Fascicle 1A*, volume 4. Addison-Wesley.

LeCorre, J. 1959. Unpublished work.

Lecoutre, C., and Vion, J. 2008. Enforcing arc consistency using bitwise operations. *Constraint Programming Letters* 2:21–35.

McGregor, J. 1979. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences* 19:229–250.

Régin, J.-C. 1994. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the 11th National Conference on Artificiel Intelligence (AAAI-94)*, 362–367.

Trick, M. A. 2003. A dynamic programming approach for consistency and propagation for knapsack constraints. *Annals of Operations Research* 118:73–84.