Flexible Management of Large-Scale Integer Domains in CSPs

Nikolaos Pothitos and Panagiotis Stamatopoulos

Department of Informatics and Telecommunications,
University of Athens,
Panepistimiopolis, 157 84 Athens, Greece
{pothitos,takis}@di.uoa.gr

Abstract. Most research on Constraint Programming concerns the (exponential) search space of *Constraint Satisfaction Problems* (CSPs) and intelligent algorithms that reduce and explore it. This work proposes a different way, not of solving a problem, but of storing the domains of its variables, an important—and less focused—issue especially when they are large. The new data structures that are used are proved theoretically and empirically to adapt better to large domains, than the commonly used ones. The experiments of this work display the contrast between the most popular Constraint Programming systems and a new system that uses the data structures proposed in order to solve CSP instances with wide domains, such as known Bioinformatics problems.

Key words: CSP domain, Bioinformatics, stem-loop detection

1 Introduction

Constraint Programming is an Artificial Intelligence area that focuses on solving CSPs in an efficient way. A CSP is a triplet containing variables, their domains (i.e. set of values) and constraints between variables. The simplicity of this definition makes Constraint Programming attractive to many Computer Science fields, as it makes it easy to express a variety of problems.

When it comes to solving a CSP, the main problem that we face is the exponential time needed, in the general case. The space complexity comes in second place, as it is polynomial in the size (usually denoted d) of the largest domain. But is $O(d)$ the best space—and therefore time—complexity we can achieve when we have to store a domain? Is it possible to define a lower bound for this complexity? Memory management is a crucial factor determining a Constraint Programming system speed, especially when d is too big.

Gent et al. have recently described data structures used to propagate the constraints of a CSP [3]. To the best of our knowledge, the representation of a domain itself has not yet been the *primary* sector of interest of a specific publication in the area. Nevertheless, Schulte and Carlsson in their Constraint Programming systems survey [7] defined formally the two most popular data structures that can represent a finite set of integers:

Bit Vector. Without loss of generality, we suppose that a domain D contains only *positive* integer values. Let a be a bit array. Then the value v belongs to D, if and only if $a[v] = 1$. Bit vector variants are implemented in many Constraint Programming solvers [1, 2].

Range Sequence. Another approach is to use a sequence of *ranges*. Formally, D is 'decomposed' into a set $\{[a_1, b_1], \ldots, [a_n, b_n]\}$, such that $\cup_i [a_i, b_i] = D$. A desired property for this sequence is to be ordered and the shortest possible, i.e. $[a_i, b_i] \cap [a_j, b_j] = \emptyset$, $\forall i \neq j$. In this case δ denotes the number of ranges.

A more simple data structure than the two above, stores only the bounds of D. E.g., for the domain $[1\,..100000]$[1] we store only two numbers in memory: 1 and 100000. Obviously, this is an incomplete representation for the non-continuous domains (e.g. $[1\,..3\ 5\,..9]$). It is therefore incompatible with most algorithms designed for CSPs; only specific methodologies can handle it [11].

On the other hand, for the above domain $[1\,..100000]$, a bit vector would allocate 100,000 bits of memory, although it could be represented by a range sequence using only two memory words. A range sequence can be implemented as a linked list, or as a binary tree, so it is costlier to search for a value in it.

In this work we study the trade-off between memory allocation cost and time consuming operations on domains. A new way of memory management that seeks to reduce the redundant space is proposed. The new algorithms and data structures are shown to perform well, especially on problems which contain large domains. Such problems eminently occur in Bioinformatics, a science that aims at extracting information from large genetic data.

2 Efficient Domain Implementations

While attempting to reduce the *space* complexity, we should not neglect *time* complexity. Except for memory allocation, a constraint programming system is responsible for two other basic operations that are executed many times on a domain:

1. *Search* whether a range of values is included in it.
2. *Removal* of a range of values from a domain.

Note that addition of values is unnecessary; the domain sizes only decrease due to constraint propagation or assignments.

Search or removal of a range of w values costs $O(w)$ time in a *bit vector;* if $w = 1$ this structure is ideal. The same operations in a *range sequence* that has been implemented as a linked list [7] require $O(\delta)$ steps, while the space complexity is much less ($O(\delta)$ too) than the bit vector's one ($O(d)$). A wiser choice would be to implement the range sequence as a binary search tree, with an average search/removal complexity $O(\log \delta)$, and the space complexity left unaffected.

[1] $[a\,..b]$ denotes the integer set $\{a, a+1, \ldots, b\}$.

However, the subtraction of a range of values from the tree is complicated. (It roughly performs two traversals and then joins two subtrees.) This is undesirable, not only for the time it spends, but also for the many modifications that are *done* on the structure. The number of modifications is crucial because they are recorded in order to be *undone* when a Constraint Programming system *backtracks,* that is when it restores a previous (or the initial) state of the domains, in order to restart the process of finding a solution to a CSP (through other paths).

2.1 Gap Intervals Tree Representation

To make things simpler and more efficient, a binary search tree of *gap ranges* was implemented. The advantage of this choice is that the subtraction of a range of values is faster, as it affects only one tree node (i.e. it inserts or modifies only one node).

For example the domain $[9\,..\,17\;44\,..\,101]$ is described by three gaps: $[-\infty\,..\,8]$, $[18\,..\,43]$ and $[102\,..\,+\infty]$. Figure 1 depicts the gaps of a domain that are arranged as a binary search tree. A node of the tree apparently contains the first and the last gap value, and pointers to the left and right 'children.'

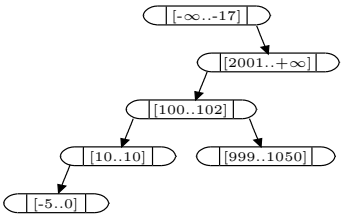


Fig. 1. A tree with the gaps of the domain $[-16\,..\,-6\;\;1\,..\,9\;\;11\,..\,99\;\;103\,..\,998\;\;1051\,..\,2000]$

2.2 Search/Delete Algorithm

Another advantage of this approach is that the two basic operations on a domain are performed by a single algorithm named SearchGap.[2] This function accepts four arguments ($gapNode$, $newStartVal$, $newEndVal$, $removeInterval$).

- If $removeInterval$ is 1, the range $[newStartVal\,..\,newEndVal]$ is deleted from the domain, which is represented by a tree whose root is $gapNode$.
- If $removeInterval$ is 0, the function returns a node of the tree that contains at least one element of $[newStartVal\,..\,newEndVal]$. If there does not exist such a node that meets this criterion, then the function returns an empty node. Thus, in case we want to check whether a range $[a\,..\,b]$ belongs to D, we call SearchGap(root, $a, b, 0$):
  - If the returned node is empty, then $[a\,..\,b] \subseteq D$;
  - otherwise $[a\,..\,b] \nsubseteq D$.

The above procedures manipulate the data structure as a normal binary search tree; the insertions of gaps and the search for specific values is done in logarithmic time as we traverse a path from the root $gapNode$ to an internal node.

While a Constraint Programming system tries to find a solution, it only adds gaps to the tree. During gap insertions the algorithm seeks to merge as many gap nodes as possible in order to keep the tree short.

[2] Available at `http://www.di.uoa.gr/~pothitos/setn2010/algo.pdf`

3 Empirical Results

Although the above domain implementation is compatible with the ordinary CSP formulation, algorithms and constraint propagation methodologies [6], it is recommended especially when we have to solve problems with large non-continuous domains. Such problems naturally occur in Bioinformatics, so we are going to apply the memory management proposed to them.

3.1 A Sequence Problem

Each human cell contains 46 chromosomes; a chromosome is part of our genetic material, since it contains a sequence of DNA nucleotides. There are four types of nucleotides, namely A, T, G and C. (A = adenine, T = thymine, G = guanine, C = cytosine.) A chromosome may include approximately 247.2 million nucleotides.

A Simple Problem Definition. Suppose that we want to 'fit' in a chromosome a sequence of four cytosines C_1, C_2, C_3, C_4 and a sequence of four guanines G_1, G_2, G_3, G_4 too. C_i and G_i designate the *positions* of the corresponding nucleotides in the DNA chain; the initial domain for a position is $[1\,..\,247200000]$. We assume the first sequence grows geometrically with $C_i = \lfloor C_{i+1}/99 \rfloor$ and the second sequence is the arithmetic progression $G_{i+1} = G_i + 99$.

Pitfalls While Solving. This naive CSP, which is limited to only eight constraint variables, may become... difficult, if we do not properly manage the domains that contain millions of values. So, we evolved the data structures of an existing Constraint Programming library and observed their behaviour in comparison with two popular systems.[3]

Naxos. At first, we integrated the gap intervals tree described into Naxos Solver [5]. Naxos is a library for an object-oriented programming environment; it is implemented in C++. It allows the statement of CSPs having constrained variables with finite domains containing integers.

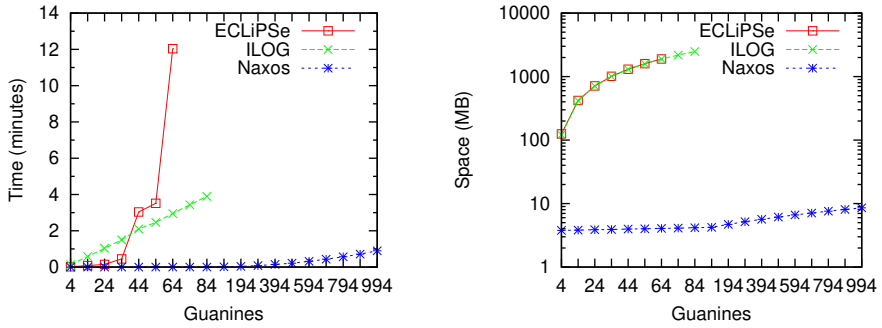
The solution[4] for the naive problem described was found immediately, using 3 MB of memory. All the experiments were carried out on a Sun Blade computer with an 1.5 GHz SPARC processor and 1 GB of memory.

ECL^{i}PS^e. On the same machine, however, it took three seconds for the constraint logic programming system ECL^{i}PS^e version 5.10[5] [2] to find the same solution, using 125 MB of memory, as it implements a bit vector variant to store

[3] The datasets and the experiments source code—for each Constraint Programming system we used—are available at `http://www.di.uoa.gr/~pothitos/setn2010`

[4] The first solution includes the assignments $C_1 = 1$, $C_2 = 99$, $C_3 = 9801$, $C_4 = 970299$, $G_1 = 2$, $G_2 = 101$, $G_3 = 200$ and $G_4 = 299$.

[5] We used the ECL^{i}PS^e library '`ic`' that targets 'Interval Constraints.'



(a) *Time* needed to find a solution (b) Memory *space* allocated

Fig. 2. The resources used by Constraint Programming systems as the problem scales

the domains. If we add one more nucleotide to the problem (i.e. one more constraint variable) the program will be terminated due to stack overflow. This happens because the default stack size is limited, so in order to continue with the following experiments, we increased it manually.

Ilog. ILOG SOLVER version 4.4 [4], a well-known C++ Constraint Programming library, needs treble time (about ten seconds) to find the solution in comparison with ECL^{i}PS^e, but it consumes almost the same memory.

Scaling the Problem. A simple way to scale the problem is to add more guanines in the corresponding sequence. Figure 2 illustrates the time and space that each system spends in order to reach a solution.

Before even adding a hundred nucleotides, ECL^{i}PS^e and ILOG SOLVER ran out of resources, as they had already used all the available physical and virtual memory. On the other hand, NAXOS scales normally, as it benefits from the proposed domain representation, and requires orders of magnitude less memory. The lower price of *allocating* space makes the difference.

3.2 RNA Motifs Detection Problem

In the previous problem we created a nucleotide sequence, but in Bioinformatics it is more important to *search* for specific nucleotide patterns/motifs inside *genomes,* i.e. the nucleotide chains of a specific organism.

We can focus on a specific pattern that describes the way that an RNA molecule folds back on itself, thus formulating helices, also known as *stem-loops* [10]. A stem-loop consists of a helix and a region with specific characters from the RNA alphabet [9].

In contrast to ILOG SOLVER, NAXOS SOLVER extended with the proposed memory management is able to solve this problem for the bacterium *Escherichia coli* genome, which is available through the site of MilPat, a tool dedicated to searching molecular motifs [8].

4 Conclusions and Further Work

In this work, it has been shown that we can achieve a much better lower memory bound for the representation of a domain, than the actual memory consumption of Constraint Programming systems. An improved way of storing a domain, through new data structures and algorithms was proposed. This methodology naturally applies to various problems with wide domains, e.g. Bioinformatics problems that come along with large genome databases.

In future, hybrid data structures can contribute towards the same direction. For example, variable size bit vectors could be integrated into binary tree nodes. Everything should be designed to be as much generic as possible, in order to exploit at any case the plethora of known algorithms for generic CSPs.

Acknowledgements. This work is funded by the Special Account Research Grants of the National and Kapodistrian University of Athens, in the context of the project 'C++ Libraries for Constraint Programming' (project no. 70/4/4639).

We would also like to thank Stavros Anagnostopoulos, a Bioinformatics expert, for his valuable help in our understanding of various biological problems and data.

References

1. Codognet, P., Diaz, D.: Compiling constraints in `clp(FD)`. The Journal of Logic Programming 27(3), 185–226 (1996)
2. ECL^{i}PS^e constraint programming system. `http://eclipse-clp.org` (2008)
3. Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Data structures for generalised arc consistency for extensional constraints. In: AAAI'07: 22$^{\text{nd}}$ National Conf. on Artificial Intelligence, Vancouver. pp. 191–197. AAAI Press, Menlo Park (2007)
4. ILOG S.A.: ILOG Solver 4.4: User's Manual (1999)
5. Pothitos, N.: Naxos Solver. `http://www.di.uoa.gr/~pothitos/naxos` (2009)
6. Sabin, D., Freuder, E.C.: Contradicting conventional wisdom in constraint satisfaction. In: PPCP'94: 2$^{\text{nd}}$ Int'l Workshop on Principles and Practice of Constraint Progr., Washington. LNCS, vol. 874, pp. 125–129. Springer, Heidelberg (1994)
7. Schulte, C., Carlsson, M.: Finite domain constraint programming systems. In: Handbook of Constr. Programming, pp. 495–526. Elsevier Science, Amsterdam (2006)
8. Thébault, P.: MilPat's user manual. `MilPat.pdf` at `http://carlit.toulouse.inra.fr/MilPat` (2006)
9. Thébault, P., de Givry, S., Schiex, T., Gaspin, C.: Searching RNA motifs and their intermolecular contacts with constraint networks. Bioinformatics 22(17), 2074–2080 (2006)
10. Watson, J., Baker, T., Bell, S., Gann, A., Levine, M., Losick, R.: Molecular Biology of the Gene, chap. 6. Pearson/Benjamin Cummings, 5$^{\text{th}}$ edn. (2004)
11. Zytnicki, M., Gaspin, C., Schiex, T.: A new local consistency for weighted CSP dedicated to long domains. In: SAC '06: Proceedings of the 2006 ACM symposium on Applied computing. pp. 394–398. ACM, New York, NY, USA (2006)