Using Indexed Finite Set Variables for Set Bounds Propagation

Ruben Duarte Viegas, Marco Correia, Pedro Barahona, and Francisco Azevedo

CENTRIA
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
{rviegas,mvc,pb,fa}@di.fct.unl.pt

Abstract. Constraint Programming (CP) has been successfully applied to numerous combinatorial problems such as scheduling, graph coloring, circuit analysis, or DNA sequencing. Following the success of CP over traditional domains, set variables were also introduced to more declaratively solve a number of different problems.

Using a bounds representation for a finite set variable allows one to compactly represent the solution set of a set constraint problem. Many consistency mechanisms for maintaining bounds consistency have been proposed and in this paper we propose to use delta domain variable information to speed up constraint propagation. Additionally, we propose the use of indexed set domain variable representations as a better means of improving the use, intuitiveness and efficiency of delta domain variables for propagation tasks.

Keywords: finite set constraint variables, graph constraint variables, constraint propagation, delta domain variables, indexation.

1 Introduction

Set variables and set constraints provide an intuitive and efficient way of modelling Constraint Satisfaction Problems's (CSP) over sets [1,2].

This paper addresses *Cardinal*, a state-of-the-art *bounds consistency* set solver (available in the *ECLiPSe Constraint System* distribution [3] and also with *CaSPER* [4,5]) which has already proved its efficiency in solving real-life set constraint problems [6,2]. More specifically, we improve over the version presented in [2] by lowering its worst-case complexity for most propagators. The improvement is based on the idea of exploring detailed information about past domain updates (delta domains), together with a new domain representation which allows us to take advantage of this information.

We define three new representations of finite set variables, combine them with delta information and compare them against the original *Cardinal* version, in *CaSPER*.

The structure of the paper is the following: in section 2 we present the formal notation we use throughout the paper. In section 3 we explain the usefulness of

H. Geffner et al. (Eds.): IBERAMIA 2008, LNAI 5290, pp. 73–82, 2008.
© Springer-Verlag Berlin Heidelberg 2008

using delta domain variables and how we can use indexation to explore it. We test our indexation mechanisms in section 4 by solving some benchmarks and comparing results. Finally, we conclude in section 5 with our closing remarks and future work.

2 Preliminaries

We start by revising useful definitions while also introducing the notation used throughout the paper.

Definition 1. [CSP] *A constraint network consists of a set of variables $\mathcal{X}$,a set of domains $\mathcal{D}$, and a set of constraints $\mathcal{C}$. Every variable $X \in \mathcal{X}$ has an associated domain $D(X)$ denoting its possible values. The constraint satisfaction problem (CSP) consists in finding an assignment of values to variables such that all constraints are satisfied.*

For CSPs over set variables, each domain $D(X)$ therefore represents a set of possible sets for variable X. In *Cardinal* only the set bounds are explicitly maintained:

Definition 2. [Set variable] *A set variable X is represented by $[a_X, b_X]_{c_X}$ where a_X is the set of elements known to belong to X (its greatest lower bound, or* glb*), b_X is the set of elements not excluded from X (its least upper bound, or* lub*), and c_X its cardinality (a finite domain variable). We define $p_X = b_X \setminus a_X$ to be the set of elements, not yet excluded from X and that can still be added to a_X.*

Solving a CSP typically involves interleaving search with some kind of consistency enforcing which effectively narrows the search space. Consistency enforcement is accomplished by means of propagators:

Definition 3. [Propagator] *A propagator $\pi : \mathcal{D} \rightarrow \mathcal{D}, \pi(\mathcal{D}) \subseteq \mathcal{D}$ is a monotonically decreasing function from domains to domains, preserving the CSP solutions.*

The set of propagators associated with $\mathcal{C}$ are executed repeatedly until fixpoint, i.e., no further domain reduction is possible by an additional execution of any propagator. A propagator may, therefore, be executed several times between consecutive fixpoint operations. In the remaining of the paper, let π_i denote the i'th execution of propagator π.

3 Delta Domains and Indexation

Generally, a *delta domain* represents the set of updates on a variable's domain between two consecutive executions of some propagator at a given fixpoint operation. In the following, let $X \ominus Y = \langle a_X \setminus a_Y, b_X \setminus b_Y \rangle$ be the standard (bounds) difference between two set variables X and Y:

Definition 4. [Delta domain] *Let $D_I(X)$ and $D_F(X)$ denote respectively the initial domain of X (i.e. before any propagator is executed), and final domain of X (i.e. after fixpoint is reached). The delta domain of variable X is $\Delta(X) = D_I(X) \ominus D_F(X)$. Let $D_{\pi_i}(X)$ be the domain of variable X right after the i'th execution of propagator π. The delta domain of variable X with respect to propagator π_i is $\Delta_{\pi_i}(X) = D_{\pi_{i-1}}(X) \ominus D_{\pi_i}(X)$.*

Maintaining delta domains is a complex task. Delta domains must be collected, stored and made available later during a fixpoint operation. Moreover, each propagator has its own (possibly distinct) set of deltas which must be updated independently.

The basic idea is to store $\Delta(X) = \{\delta_1 \dots \delta_n\}$ in each set variable X as the sequence (we use a singly-linked list) of every atomic operation δ_i applied on its domain since the last fixpoint. In this context, δ_i is either a removal or insertion of a range of contiguous elements respectively from the set *lub* or in the set *glb*. A delta domain with respect to some propagator execution $\Delta_{\pi_i}(X)$ is then just a subsequence from the current $\Delta(X)$. Although the full details of this task are out of the scope for this paper, we note that domains may be maintained almost for free on constraint solvers with a smart garbage collection mechanism.

The availability of delta information can speed up constraint propagation for integer domains [7]. As noted in [2], it can also be very useful for set variables.

However, using delta information with *Cardinal* domain representation (a pair of lists for a_X and b_X) does not help reducing complexity in most cases. The reason is that in addition to finding what has changed, propagators need to perform domain pruning as well, and usually these operation may be combined.

As explained, $\Delta(X)$ is a list of contiguous range updates on the domain of X. However, since propagators are not executed in any specific order, the list of ranges is not necessarily sorted. Unfortunately, this means that using delta domains even increases the worst-case complexity. Consider the previous example, but where elements $e_1 \dots e_n$ were inserted in reverse order. The delta domain is now $\Delta_\pi(X) = \{(e_n, e_n), \dots, (e_1, e_1)\}$ instead of $\Delta_\pi(X) = \{(e_1, e_n)\}$ as previously. The complexity for the contained propagator execution is therefore $O(n \times \#b_Y)$[1]. Although in practice $|\Delta_\pi(X)|$ is typically small, we can effectively improve this theoretic worst case bound by indexing the set variable domain.

Since delta domain variables are independent of the data structures used for the domain variables themselves, this enables the development of alternative data structures for storing the domain information and which may make a more efficient use of delta domain information. Note that the same could be achieved with other domain representations (e.g. trees) since in our implementation, delta domain variables are orthogonal to the data structures used for the domain variables themselves.

[1] Note that this could possibly be improved by first sorting $\Delta_\pi(X)$, yielding $O(n \log n + n + \#b_Y)$.

3.1 Indexation

Indexation is a well known mechanism for allowing direct access of elements in constant time. It is not hard to see that with constant access to elements in the domain of X most operations can be done in optimal time. In the running example, the insertion of $e_1 \dots e_n$ in a_X (in any ordering) requires the insertion of $\Delta(X)$ in the domain of Y, which can be done optimally in $O(n)$. Finally, note that indexation *without* delta domains would still require the inspection of $a_X \setminus a_Y$, and thus does not improve over the *Cardinal* domain representation.

In the next sections we introduce three different indexation mechanisms for finite set variables upon which these finite sets domain variables and the *Cardinal* filtering rules can be implemented and which we will evaluate in section 4.

Hash Implementation. The first indexation mechanism uses a list for storing, in sorted order, the elements that constitute the finite set variable (*lub*) and an hash-table to directly access them. At any given time an element may either be in the set's *glb*, *poss* (= *lub* \ *glb*) or may even be removed from the set. For this purpose every node in the hash-table will have both the represented element and an indicator of where the element currently is (*glb*,*poss* or *out*). This implementation will henceforth be called *Hash*.

In Figure 1 we represent the internal structure of a *Cardinal* finite set variable, using the *Hash* implementation, with $glb = \{a\}$ and $poss = \{f, p, z\}$ and having removed the element p. In it, *index* represents our indexing mechanism for efficiently accessing the elements while *elementsHead* and *elementsTail* represent the limits of the list of all elements in sorted order for iteration purposes.

Hash-1 Implementation. This implementation inherits the same structure and functionality as the *Hash* implementation. The only modification is in how the delta domain events are stored. In the *Hash* implementation, deltas are stored as ranges allowing to store in one single delta a complete set of elements that

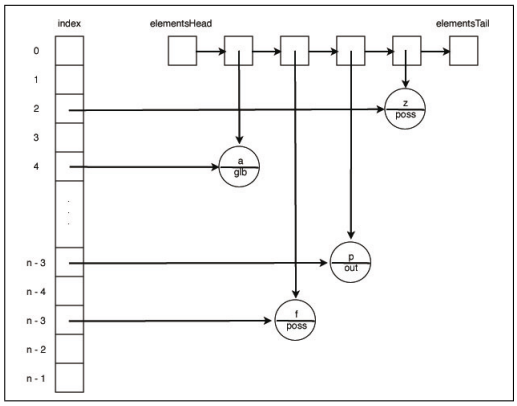

Fig. 1. Hash complete structure

had been either removed from or inserted into the finite set domain variable at a given time. In the *Hash-1* implementation, even though a change in the domain may be caused by a set of elements, these elements are stored individually as deltas, i.e., each delta contains only one of those elements.

Hash+Poss Implementation. The third and final indexation mechanism, in addition to the structures used by both the *Hash* and the *Hash-1* implementations uses a list to store only the elements in the set's *poss*. This is important since many dynamic heuristics reason only about elements which are in the set's *poss* and both the *Hash* and the *Hash-1* implementations do not have a dedicated structure for efficiently accessing it.

If we disregard the indexation used, the main difference between this implementation and the original *Cardinal* implementation is that *Hash+Poss* does not use a structure for the set's *glb* thus theoretically outperforming the original *Cardinal* version for insertion operations since it does not have to sweep the entire set's *glb* in order to find the position where to insert the element, instead it just removes the element from the *poss* structure and updates the indicator to *glb*.

In Figure 2 we represent the internal structure of a *Cardinal* finite set variable, using the *Hash+Poss* implementation, with $glb = \{a\}$ and $poss = \{f, p, z\}$ and having removed the element p. In it, *index* represents our indexing mechanism for efficiently accessing the elements, *elementsHead* and *elementsTail* represent the limits of the list containing all elements in sorted order for iteration purposes and finally, *possHead* and *possTail* represent the limits of the list containing only the elements belonging to the variable's *poss*, in sorted order, for set *poss*'s iteration purposes.

3.2 Complexity Analysis

In order to assess the theoretical efficiency of the different *Cardinal* indexed implementations we introduce the following operations, which represent the most

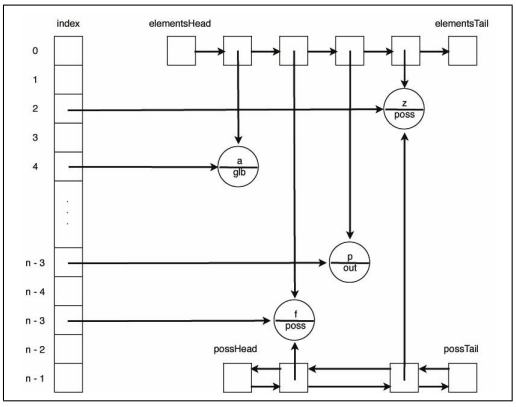

Fig. 2. Hash+Poss complete structure

Table 1. Worst-case temporal complexity of the *hash+poss Cardinal* implementation

Operation	Complexity			
	Cardinal	*Hash*	*Hash-1*	*Hash+Poss*
inGlb(e**):**	$O(\#a_S)$	$O(1)$	$O(1)$	$O(1)$
inPoss(e**):**	$O(\#p_S)$	$O(1)$	$O(1)$	$O(1)$
inLub(e**):**	$O(\#b_S)$	$O(1)$	$O(1)$	$O(1)$
ins(e**):**	$O(\#b_S)$	$O(1)$	$O(1)$	$O(1)$
ins($*e$**):**	$O(1)$	$O(1)$	$O(1)$	$O(1)$
insRg(b**,**e**):**	$O(\#b_S)$	$O(\#(e-b))$	$O(\#(e-b))$	$O(\#(e-b))$
insRg($*b$**,**$*e$**):**	$O(\#a_S)$	$O(\#(*e-*b))$	$O(\#(*e-*b))$	$O(\#(*e-*b))$
rem(e**):**	$O(\#p_S)$	$O(1)$	$O(1)$	$O(1)$
rem($*e$**):**	$O(1)$	$O(1)$	$O(1)$	$O(1)$
remRg(b**,**e**):**	$O(\#p_S)$	$O(\#(e-b))$	$O(\#(e-b))$	$O(\#(e-b))$
remRg($*b$**,**$*e$**):**	$O(1)$	$O(\#(*e-*b))$	$O(\#(*e-*b))$	$O(\#(*e-*b))$
iterateGlb():	$O(\#a_S)$	$O(\#b_S)^2$	$O(\#b_S)^2$	$O(\#b_S)^2$
iteratePoss():	$O(\#p_S)$	$O(\#b_S)^2$	$O(\#b_S)^2$	$O(\#p_S)$
iterateLub():	$O(\#b_S)$	$O(\#b_S)^2$	$O(\#b_S)^2$	$O(\#b_S)^2$

common operations performed on *Cardinal*'s underlying structure, all other set operations being decomposable into these more simple ones.

- **inGlb(**e**):** Consists of verifying if e is in the set's *glb*
- **inPoss(**e**):** Consists of verifying if e is in the set's *poss*
- **inLub(**e**):** Consists of verifying if e is in the set's *lub*
- **ins(**e**):** Consists of moving e from the set's *poss* to the set's *glb*
- **ins(**$*e$**):** Consists of moving the element pointed by e from the set's *poss* to the set's *glb*
- **insRg(**b**,**e**):** Consists of moving the range of elements between b and e from the set's *poss* to the set's *glb*
- **insRg(**$*b$**,**$*e$**):** Consists of moving the range of elements between the one pointed by b and the one pointed by e from the set's *poss* to the set's *glb*
- **rem(**e**):** Consists of removing e from the set's *poss*
- **rem(**$*e$**):** Consists of removing the element pointed by e from the set's *poss*
- **remRg(**b**,**e**):** Consists of removing the range of elements between b and e from the set's *poss*
- **remRg(**$*b$**,**$*e$**):** Consists of removing the range of elements between the element pointed by b and the element pointed by e from the set's *poss*
- **iterateGlb():** Consists of iterating through all the elements in the set's *glb*
- **iteratePoss():** Consists of iterating through all the elements in the set's *poss*
- **iterateLub():** Consists of iterating through all the elements in the set's *lub*

[2] For the indexed versions, elements are not removed from the index, but marked with the *out* indicator and therefore the cost increases for having to pass through these elements.

In Table 1, we present the worst-case temporal complexity analysis for the operations summarised at the beginning of the section for a set variable S and using the original *Cardinal* implementation as well as the 3 indexed implementations.

From Table 1 we conclude that all the indexed versions outperform the original version for access operations, inserting elements and ranges of elements by value and removing elements and ranges of elements by value. On the other hand the original version outperforms the indexed versions for the insertion and removal of ranges of elements by pointer and for the *glb* and *poss* iteration operations, being *Hash+Poss* the exception for this last operation.

At this time, indexed versions provide a more efficient framework than the original version for a considerable part of the operations considered, albeit the original one still manages to present better results then the indexed versions for some of the operations. In the next section and in order to clearly determine the benefit of these indexation mechanisms, we introduce two problems able to be directly encoded, into *CaSPER* [4], as set constraint problems and as graph constraint problems (graphs which in turn are decomposable into sets as explained in [8,9,10,11]) and compare results between all these implementations.

4 Benchmarks

In this section we present the results obtained for our *Cardinal* versions for three different benchmarks. All results were obtained using an Intel Core 2 Duo 2.16 GHz, 1.5 Gb of RAM, 4MB L2 cache.

4.1 Set Covering Problem

In the set-covering problem [12,13,14], we are given a set $U = \{1, 2, \ldots, n-1, n\}$ of elements and a set $X = \{S_1, S_2, \ldots, S_{k-1}, S_k\}$ of sets where each S_i ($1 \leq i \leq k$) is contained in U and the objective is to determine the smallest subset S of X such that $U = \bigcup_{S_i \in S} S_i$.

We tested a solution for the set covering problem for our set implementations with the benchmarks used in [13,15]. A time limit of 300 seconds was imposed. We ran the application for each of the *Cardinal* versions and using each of the problem instances. In Table 2 we present the results, in seconds, obtained for the instances presenting the most variance, where $Size$ is the cardinality of the best solution found.

In Table 2 we can see that for some of the instances the *Hash-1* version could not even manage to obtain a solution as good as the others and when it did it usually was far more inefficient in the task. In turn, *Hash* always obtained the same best solution, being able to outperform the original *Cardinal* version for some of these instances. Additionally, *Hash+Poss* achieved the same results as the original and the *Hash* versions albeit taking slightly more time than the latter to find them. It seems clear that, for this problem, the *Hash Cardinal* version was the best one and there was benefit in using an indexed representation.

Table 2. Comparison between *Cardinal* versions for the set-covering problem

Problem	Size	Cardinal	Hash	Hash-1	Hash+Poss
scp46	42	154.210	124.330	—	131.640
scp56	39	33.900	34.620	133.380	47.020
scp58	38	24.570	20.390	83.620	26.070
scpa3	42	1.230	1.220	5.090	1.930
scpa4	40	93.480	82.890	—	118.770
scpc1	47	0.110	0.130	0.210	0.150
scpd1	26	57.590	33.640	187.720	41.560

4.2 Metabolic Pathways

Metabolic networks [16,17] are biochemical networks which encode information about molecular compounds and reactions which transform these molecules into substrates and products. A pathway in such a network represents a series of reactions which transform a given molecule into others. An application for pathway discovery in metabolic networks is the explanation of DNA experiments.

Constraints in metabolic networks pathway finding include the preferable avoidance of highly connected molecules (since they do not bring additional pathway information), the safeguard that reactions are observed in a single direction (to ignore a reaction from the metabolic network when the reverse reaction is known to occur, so that both do not occur simultaneously) and the obligation to visit a given set of mandatory molecules (when, for instance, biologists already know some of the products which are in the pathway but do not know the complete pathway). In [8,9,10,11] a complete modelling for this problem is presented.

A possible heuristic for search, adapted from [18], is to iteratively extend a path (initially formed only by the starting vertex) until reaching the final vertex. At every step, we determine the next vertex which extends the current path to the final vertex minimizing the overall path cost. Having this vertex we obtain the next edge to label by considering the first edge extending the current path until the determined vertex. The choice step consists in including/excluding the edge from the graph variable. If the edge is included the current path is updated and the last vertex of the path is the out-vertex of the included edge, otherwise the path remains unchanged and we try another extension. The search ends as soon as the final vertex is reached and the path is minimal. This heuristic shall be referred as *shortest-path*.

We present the results obtained for the problem of solving the shortest metabolic pathways for three metabolic chains (*glyco*, *heme* and *lysine*) and for increasing graph orders (the order of a graph is the number of vertices that belong to the graph).

Table 3 presents the results, in seconds, obtained using the *shortest-path* heuristic using instances obtained from [19].

In Table 3 we can observe that the original *Cardinal* version was consistently worse than the three indexed versions. In what regards the three indexed

Table 3. Comparison between *Cardinal* versions for the Metabolic Pathways problem

Order	Cardinal			Hash			Hash-1			Hash+Poss		
	Glyco	Lysine	Heme	Glyco	Lysine	Heme	Glyco	Lysine	Heme	Glyco	Lysine	Heme
200	0.2	0.4	0.2	0.1	0.3	0.1	0.1	0.3	0.1	0.1	0.3	0.1
400	1.2	1.4	0.6	0.8	1.0	0.5	0.8	1.0	0.4	0.8	1.1	0.5
600	1.9	2.2	1.1	1.6	1.6	0.9	1.6	1.7	1.0	1.6	1.7	1.0
800	3.1	2.9	1.8	2.5	2.6	1.6	2.6	2.6	1.5	2.6	2.6	1.6
1000	5.2	4.0	3.1	4.5	3.6	2.6	4.4	3.5	2.5	4.5	3.6	2.5
1200	7.6	6.5	4.4	6.1	5.5	3.5	6.2	5.4	3.5	6.4	5.7	3.6
1400	10.2	9.6	6.0	8.7	7.9	4.9	8.5	7.9	4.9	8.7	7.9	5.1
1600	13.8	10.8	8.0	11.9	9.5	6.2	11.7	9.6	6.3	12.2	9.9	6.5
1800	18.5	13.9	10.7	16.2	11.9	8.4	16.0	11.7	8.2	16.5	12.1	8.6
2000	21.7	17.6	13.5	18.8	15.5	11.0	18.6	15.4	10.6	18.7	15.8	11.3

versions, this benchmark does not present a clear means of distinction in terms of efficiency, since all of them present similar results. Despite this fact, the indexed versions improve the results of the original *Cardinal* version and again bring a benefit in using indexation for finite sets and graph domain variables.

5 Conclusions and Future Work

In this paper we explained the usefulness in combining delta domain variables and indexation for defining more intuitive and efficient propagators. We tested three different indexed finite set representations with two benchmarks which presented us a grounds for comparing these indexed versions with the original *Cardinal* version.

By considering the results obtained, the indexed versions with delta domains appear as a competitive alternative to the original *Cardinal* version. However, the indexed versions did not provide a considerable speed-up. We think that this may be due to the increase of the number of copy operations necessary for storing and restoring the state of the indexed set domains, compared with the original *Cardinal* version. For instance, the removal of a range of n contiguous elements requires $O(n)$ trailing operations in the indexed domains version, while it is $O(1)$ in the original *Cardinal* version (only 2 pointers in the list are modified).

As shown, the indexed representations are better than the original version for some class of constraints problems. Currently we are trying to identify the special features which characterise this class. We also intend to study other finite set representations, of which we highlight the use of delta domain information with the original *Cardinal* version. While the worst-case runtime cannot be improved with this combination (as pointed in section 3), the availability of delta information may be exploited only when it effectively decreases the required number of operations compared to sweeping the variable's domain.

References

1. Gervet, C.: Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language. Constraints journal 1(3), 191–244 (1997)
2. Azevedo, F.: Cardinal: A finite sets constraint solver. Constraints journal 12(1), 93–129 (2007)
3. ECLiPSE Constraint System, `http://eclipse.crosscoreop.com/`
4. Correia, M., Barahona, P., Azevedo, F.: Casper: A programming environment for development and integration of constraint solvers. In: Azevedo, F., Gervet, C., Pontelli, E. (eds.) Proceedings of the First International Workshop on Constraint Programming Beyond Finite Integer Domains (BeyondFD 2005), pp. 59–73 (2005)
5. CaSPER: Constraint Solving Programming Environment for Research, `http://proteina.di.fct.unl.pt/casper/`
6. Azevedo, F.: Constraint Solving over Multi-Valued Logics. Frontiers in Artificial Intelligence and Applications, vol. 91. IOS Press, Amsterdam (2003)
7. Lagerkvist, M., Schulte, C.: Advisors for incremental propagation. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 409–422. Springer, Heidelberg (2007)
8. Viegas, R.D., Azevedo, F.: GRASPER: A Framework for Graph CSPs. In: Lee, J., Stuckey, P. (eds.) Proceedings of the Sixth International Workshop on Constraint Modelling and Reformulation (ModRef 2007), Providence, Rhode Island, USA (September 2007)
9. Viegas, R.D., Azevedo, F.: GRASPER: A Framework for Graph Constraint Satisfaction Problems. In: Analide, C., Novais, P., Henriques, P. (eds.) Simpósio Doutoral em Inteligência Artificial, Guimarães, Portugal (December 2007)
10. Viegas, R.D., Azevedo, F.: GRASPER: A Framework for Graph Constraint Satisfaction Problems. In: Azevedo, F., Lynce, I., Manquinho, V. (eds.) Search Techniques for Constraint Satisfaction, Guimarães, Portugal (December 2007)
11. Viegas, R.D.: Constraint Solving over Finite Graphs. Master's thesis, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa (2008)
12. Padberg, M.W.: Covering, Packing and Knapsack Problems. Annals of Discrete Mathematics, vol. 4 (1979)
13. Beasley, J.: An algorithm for set covering problems. European Journal of Operational Research 31, 85–93 (1987)
14. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press, Cambridge (2001)
15. Beasley, J.: A Lagrangian Heuristic for Set-Covering problems. Naval Research Logistics (NRL) 37(1), 151–164 (1990)
16. Mathews, C., van Holde, K.: Biochemistry, 2nd edn. Benjamin Cummings (1996)
17. Attwood, T., Parry-Smith, D.: Introduction to Bioinformatics. Prentice-Hall, Englewood Cliffs (1999)
18. Sellmann, M.: Cost-based filtering for shorter path constraints. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 694–708. Springer, Heidelberg (2003)
19. Dooms, G.: The CP(Graph) Computation Domain in Constraint Programming. PhD thesis, Faculté des Sciences Appliquées, Université Catholique de Louvain (2006)