Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 253 (2009) 113–128
www.elsevier.com/locate/entcs

Lazy Constraint Imposing for Improving the Path Constraint

Ruben Duarte Viegas[1,2] Francisco Azevedo[3]

CENTRIA - Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Caparica, Portugal

Abstract

In this paper we propose a lazy constraint imposing mechanism for improving the path constraint in *GRASPER*, a state-of-the-art graph constraint solver, having obtained very promising results in terms of both time and space in solving an interesting problem in the Biochemistry subject area, in comparison with *CP(Graph)*, the state-of-the-art solver.

Keywords: constraint programming, graph constraint propagation, path constraint

1 Introduction

Constraint Programming (CP) [16,7,21] has been extensively used for solving combinatorial [18], scheduling [13], allocation [15] problems, among others, in various domains. After the appearance of sets [12] in CP, graph domain variables and corresponding operations were defined [10,9,26,27] allowing users to directly create and manipulate these variables in order to model their actual problem in a much more higher level than before.

One of the definitions proposed for graph domain variables is the one specified in [26,27], which is implemented in *GRASPER (GRaph ConstrAint Satisfaction Problem solvER)*, available in the *CaSPER*[4] platform [5]. As the name indicates, *GRASPER* is a graph constraint satisfaction problem solver. It is directly based upon a finite set solver, *Cardinal* [2] and it provides the means for creating directed

[1] Thanks to the Fundação para a Ciência e Tecnologia for the financial support (SFRH/BD/41362/2007) and a special thanks to Marco Correia for his invaluable help and guidance.

[2] Email:ruben.viegas@di.fct.unl.pt

[3] Email:fa@di.fct.unl.pt

[4] Available at http://proteina.di.fct.unl.pt/casper/

1571-0661
doi:10.1016/j.entcs.2009.10.020

and undirected graph variables, a nucleus of basic constraints upon which more complex and useful constraints can be provided ranging from constraints to impose graph properties (order, size, degree, reachability, connectedness, path, ...) and to impose graph relationships (underlying and oriented, reverse, complementary graph relationships). As demonstrated in [26,27], *GRASPER* appeared as an alternative to *CP(Graph)*, the state-of-the-art graph constraint solver [10,9] in the comparison made between the two for the *Metabolic Pathways Problem* [1,24,17], a problem which can be viewed as a path discovery problem in biochemical networks.

Among all the constraints available on a typical graph solver, one of the most important ones is the *path* constraint. By definition, a *path* between two vertices is a sequence of unique vertices contained in the vertex-set of a graph, starting at an initial vertex and finishing in a terminal vertex and such that for every pair of successive vertices there is an edge linking them (and preserving the direction) in the edge-set of the graph.

In this paper we explain how we can improve, not only in time but also in space, the *path* constraint by employing a lazy constraint imposing mechanism. We start in section 2 by briefly introducing Constraint Programming. In section 3 we briefly describe *GRASPER* and define the *path* constraint and in section 4 we explain how the *path* constraint, as defined in [26,27], was implemented upon *GRASPER*. Subsequently, in section 5 we explain how, using lazy constraint imposing, we can implement a much more efficient *path* constraint and we use these two implementations for solving the *Metabolic Pathways Problem*, presenting results and comparing both implementations, against the state-of-the-art solver, *CP(Graph)*, in section 6. Finally, we end with our closing remarks and future work, in section 7.

2 Constraint Programming

A problem where relations between variables (to which values must be assigned) are restricted by a number of constraints that must be satisfied, is referred to as a Constraint Satisfaction Problem (CSP) [16]. The goal, in Constraint Programming (CP), is to assign values to all the variables without violating any constraint, or to prove this to be impossible. The space formed by all possible combinations of assignments is referred to as the search space.

Search for solutions involve assigning variables with values from their domain and, when a contradiction is found due to some violated constraint, perform some form of backtracking to undo choices made and try other yet unexplored branch of the search tree.

After the appearance of the finite domains [25], where variables could be restricted to range over finite subsets of the universe of values, set constraint solving was proposed in [20] and formalised in [12] with the *ECLiPSe Constraint System* [5] library *Conjunto*, specifying set domains by intervals whose lower and upper bounds are known sets ordered by set inclusion. Such bounds are denoted as *glb* (greatest lower bound) and *lub* (least upper bound). The *glb* of a set variable S can be

[5] Available at http://sourceforge.net/projects/eclipse-clp

seen as the set of elements that are known to belong to set S, while its *lub* is the set of all elements that can belong to S. Local consistency techniques are then applied using interval reasoning to handle set constraints (e.g. equality, disjointedness, containment, together with set operations such as union or intersection). Set domains proved their usefulness in declarativeness and efficiency for $\mathcal{NP}$-complete combinatorial search problems dealing with sets, compared to constraint solving over finite integer domains. Later on, *Cardinal* (also in *ECLiPSe Constraint System*) [2], improved on *Conjunto* by extending propagation on set functions such as cardinality.

Recently, graph domain variables were defined [10,9,26,27] providing new opportunities for modelling graph-related constraint problems. Graph domain variables were defined over two finite sets, one for the set of vertices and one for the set of edges, each edge being a pair of elements in the set of vertices. Examples of graph-related problems are job-shop scheduling [13], planning [19], graph colouring [11], circuit analysis [14], among others. Finite graph variables also allow a direct implementation of constraint graphs, where each vertex represents a variable and each edge a constraint imposed on those variables making it a very intuitive way of understanding and programming constraint networks.

Since the search space of combinatorial problems is usually intractable, a naïve generate-and-test approach, in which each combination of possible assignments is generated and then tested for a solution, until one is found, is unsuitable. Hence, constraint reasoning techniques are usually applied in AI to (often, drastically) reduce search space by discarding impossible solutions [7]. Such local consistency techniques look ahead at logical predicates defined as constraints to discard impossible values from the domain of individual variables.

In general, solvers using such techniques are incomplete, which means that reaching a consistent state for the CSP is not a sufficient condition for its satisfiability. Hence, a search phase must still occur to find a possible assignment of values to all the variables. Consistency techniques are interleaved during this search phase to constantly reduce search space, aiming at saving computation time. Notice that there is a trade-off between the level of consistency applied and the amount of pruning (of the search tree) obtained. Usually, greater levels of consistency produce decreasingly larger improvements on pruning and require increasingly larger amounts of CPU time, thus becoming counter-productive at some stage.

Consistency enforcement is accomplished by means of propagators:

Definition 2.1 A propagator π for a constraint c involving a variable with domain D is a monotonically decreasing function such that $\pi(D) \in D$.

The set of propagators 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 fixpoint operations. In the remaining of the paper, let π_i denote the i'th execution of propagator π.

Constraint solvers thus apply constraint propagation in order to remove redundant (i.e. impossible) values from the domains of variables involved in stored

constraints. If the domain of some variable becomes empty after the application of such techniques, then the CSP is insoluble. Otherwise, the CSP is said to be consistent (with regard to some properties) and there may be a solution, which has to be found to definitely prove one exists.

3 *GRASPER* and the *path* constraint

A graph [3,28,8], is composed by a set of vertices and by a set of edges, where each edge connects a pair of the graph's vertices. Therefore a graph variable can be seen as a pair (V, E) where both V and E are finite set variables. In a directed graph variable each edge is represented by a pair (X, Y) specifying a directed arc from X towards Y.

As for finite integer domains, where variables have a lower bound and an upper bound delimiting the set of possible values that the variable can be instantiated to, we have for finite set and graph domains the same concept.

In finite sets, the domain of each variable is represented by two sets: the *greatest lower bound (glb)* set and the *least upper bound (lub)* set, ordered by set inclusion, which define, respectively, the smallest and the biggest sets to which the variable can be instantiated. In finite graphs, the graph's *glb* is defined as the composition of its vertex-set and edge-set *glb*s and, similarly, the graph's *lub* is defined as the composition of its vertex-set and edge-set *lub*.

We start by defining finite set and finite graph (both directed and undirected) domain variables and then proceed to the description of the functionality we intend to improve.

Definition 3.1 *[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 (*glb*)), $b_X$is the set of elements not excluded from X (its least upper bound (*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 (or, to put it short, *poss*).

Definition 3.2 *[Directed Graph variable]* A directed graph variable X is represented by $dirgraph(V_X, E_X)$ where V_X is a finite set variable representing the vertices of X and E_X another finite set variable representing the edges of X.

Definition 3.3 *[Undirected Graph variable]* An undirected graph variable X is represented by $undirgraph(V_X, E_X)$ where V_X is a finite set variable representing the vertices of X and E_X another finite set variable representing the edges of X.

3.1 Delta domains

CaSPER, the framework where *GRASPER* is built upon provides a very useful structure for use in propagators: *delta domain* variables. A *delta domain* represents the set of updates on a variable domain between two consecutive executions of some propagator. The availability of *delta domain* variables improves the use,

intuitiveness and efficiency of propagators since they provide information about what changed since the last time the propagator was triggered. In the following, let $X \ominus Y = \langle a_Y \setminus a_X, b_X \setminus b_Y \rangle$ be the standard (bounds) difference between two set variables X and Y, with the same type of domain:

Definition 3.4 *[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 a 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 (a singly-linked list is used) 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.

3.2 Core constraints

In order to create and manipulate graph domain variables we provide two constructors (one for directed and one for undirected graph variables) which provide the core constraints of the graph constraint solver.

All the basic operations for accessing and modifying the vertices and edges are supported by finite sets primitives, provided by *Cardinal*, so no additional functionality is needed. Therefore, it is possible to create and manipulate graph variables for use in constraint problems just by providing two simple constraints for graph variable creation and delegating to a set solver the underlying core operations on sets.

These core constraints allow basic manipulation of graph variables, but we also define some other, more complex, constraints based on the core ones thus providing a more powerful, intuitive and declarative set of functions for graph variable manipulation. One of these constraints is the path constraint.

A graph $G = (V, E)$ defines a path between an initial vertex v_s and a final vertex v_f if there is a sequence of vertices $\{v_0, v_1, \dots, v_{n-1}, v_n\} \in V$ such that $\forall i \in \{0, n-1\}, (v_i, v_{i+1}) \in E$ and where $v_0 = v_s$ and $v_n = v_f$. As specified in [26,27] the *path* constraint can be specified as (we only specify the rule for directed graph variables, being the one for undirected ones very similar):

$$path(G_D, v_0, v_f) \equiv quasipath(G_D, v_0, v_f) \wedge weakly\_connected(G_D)$$

which basically says that for ensuring the *path* constraint, one can just ensure a *quasipath* constraint and a *weakly_connected* constraint. This allows the path constraint to be directly decomposed into two more simple constraints.

The *weakly_connected* constraint is a constraint imposing that any two vertices of a graph are reachable from one another, disregarding the orientation of the edges (please consult [27] for details). In turn, the *quasipath* constraint, is a degree constraint, imposing that every vertex of a graph has exactly one predecessor and one successor in the graph. The *quasipath* constraint, for directed graph variables, is specified as:

$$quasipath(G_D, v_0, v_f) \equiv \begin{array}{l} \forall v \in V(G_D) \\ predecessors(G_D, v, P) \wedge \\ successors(G_D, v, S) \wedge \end{array} \begin{cases} \#P = 0 \wedge \\ \#S = 1 & , if\ v = v_0 \\ \\ \#P = 1 \wedge \\ \#S = 0 & , if\ v = v_f \\ \\ \#P = 1 \wedge \\ \#S = 1 & , otherwise \end{cases}$$

which basically dictates that every vertex that belongs to the graph has to have exactly one predecessor and one successor, exceptions being the initial vertex which has no predecessor and the final vertex which has no successor. $predecessors(G_D, v, P)$ and $successors(G_D, v, S)$ represent the constraints for imposing the predecessors and successors of a vertex in a graph.

The $predecessors(G_D, v, P)$ constraint can be expressed as:

$$predecessors(G_D, v, P) \equiv P \subseteq V(G_D) \wedge \forall v' \in V(G_D) : (v' \in P \equiv (v', v) \in E(G_D))$$

Similarly, the $successors(G_D, v, S)$ constraint can be expressed as:

$$successors(G_D, v, S) \equiv S \subseteq V(G_D) \wedge \forall v' \in V(G_D) : (v' \in S \equiv (v, v') \in E(G_D))$$

4 Imposing the *path* constraint

In this section we explain how the *path* constraint was implemented in *GRASPER* and also explain, in general terms, how *CP(Graph)* imposed this constraint, analyzing both solutions. For the next sections, we will denote S' as the new state of a variable S (after propagation) and S as its previous state. The *glb* of S will be represented by $\underline{S}$ and the *lub* of S by $\overline{S}$.

Regarding *GRASPER*'s initial implementation, on imposing the *quasipath* constraint the first task was to iterate over all vertices in the graph variable's *glb* and to impose that their predecessor and successor sets had a cardinality of 1 (exceptions

being the initial and final vertices as explained previously), thus ensuring that every vertex imposed *a priori* to be part of the solution respects the *quasipath* constraint.

The next task was to iterate over all vertices in the graph variable's *poss*. Since they are in the graph variable's *poss* we can not just impose the cardinality of their predecessor and successor sets to have a cardinality of 1 because some of these vertices are still unknown to be part of the solution. A strategy is needed to enforce the *quasipath* constraint on these vertices, such that when one of them is imposed to be part of the solution, the cardinality of its predecessor and successor sets is set to 1, and such that when one of them is imposed not to be part of the solution, the cardinality of its predecessor and successor sets is set to 0 (forcing edge-removal).

In order to tackle this problem the following strategy was adopted: we obtained the predecessor and successor sets for each of these vertices and stored them for future access. After this storage, we could reason upon these sets in the following way (P_v and S_v represent the predecessor and successor sets of a vertex v, respectively):

- If at any time, a vertex is removed from the graph then its predecessor and successor sets cardinality is set to 0

$$\forall v \notin \overline{V} : \#P_v = 0 \wedge \#S_v = 0 \tag{1}$$

- If at any time, a vertex is added to the graph then its predecessor and successor sets cardinality is set to 1

$$\forall v \in \underline{V} : \#P_v = 1 \wedge \#S_v = 1 \tag{2}$$

- If at any time, one vertex has the cardinality of one of its predecessor or successors sets instantiated to 0, then the vertex is removed from the graph

$$\overline{V}' \leftarrow \overline{V} \setminus \{v \in \overline{V} : \#P_v = 0 \vee \#S_v = 0\} \tag{3}$$

- If at any time, one vertex has the cardinality of one of its predecessor or successors sets instantiated to 1, then the vertex is added to the graph

$$\underline{V}' \leftarrow \underline{V} \cup \{v \in \overline{V} : \#P_v = 1 \vee \#S_v = 1\} \tag{4}$$

This implementation is indeed very declarative and intuitive since it is basically a direct translation into constraints of the actual problem. We used this implementation and developed a solution for the *Metabolic Pathways Problem*, whose results we were able to compare against the ones obtained with *CP(Graph)*'s solution and even though not as efficient for the best heuristic we concluded that the results were acceptable and that *GRASPER* was nonetheless an alternative to using *CP(Graph)*.

There were, however, some problems with this implementation regarding both space and time. The problem with space is that basically we are obtaining and storing the predecessor and successor sets of each vertex in the graph variable's *poss* even though we do not know whether a given vertex will become part of the actual solution or not. In a worst case scenario, if we have N vertices and the graph is complete (every vertex is adjacent to every other), then we will have $O(N^2)$ spatial complexity just to store the predecessor and successor sets, which is clearly very expensive.

Regarding time, this solution had several problems. First of all, and applying the same reasoning as before, we were wasting time obtaining the predecessor and

successor sets of each vertex in the graph variable's *poss* not knowing if they would ever become useful. Not only did we wasted time obtaining these sets but we also wasted time in maintaining them, since for each of these sets we had to maintain their consistency with the graph variable. Considering for instance the successor set of a vertex v, we had to maintain consistency in the following way (V and E represent the vertex and edge sets of the graph, respectively):

- If a vertex s is removed from the successor set, the corresponding edge (v, s) must be removed from the graph

$$\overline{E}' \leftarrow \overline{E} \cap \{(x, y) \in \overline{E} : (x \neq v) \vee (x = v \wedge y \in \overline{S})\} \tag{5}$$

- If a vertex s is added to the successor set, the corresponding edge (v, s) must be added to the graph

$$\underline{E}' \leftarrow \underline{E} \cup \{(v, x) \in \overline{E} : x \in \underline{S}\} \tag{6}$$

- If an edge (v, s) is removed from the graph, the vertex s is removed from v's successor set

$$\overline{S}' \leftarrow \overline{S} \cap \{x \in \overline{S} : (v, x) \in \overline{E}\} \tag{7}$$

- If an edge (v, s) is added to the graph, the vertex s is added to v's successor set

$$\underline{S}' \leftarrow \underline{S} \cup \{x \in \overline{S} : (v, x) \in \underline{E}\} \tag{8}$$

A cost linear in the number of vertices and edges of the graph is required, in a worst case scenario, every time one of these operations were performed. Given that each of these operations is performed for every predecessor or successor sets, and that we have a predecessor and successor sets for each vertex in the graph variable's *poss*, many of which may not belong to the solution, it is easy to conclude we were wasting a considerable amount of time.

CP(Graph), in turn, uses a different method for imposing this constraint. It defines a view over the graph variable's domain, more suitable for this problem than a vertex-set and edge-set representation. This view provides an adjacency representation, i.e., it maintains for each vertex a list of its adjacent vertices and it requires some form of consistency maintenance, ensuring that any change in the raw domain representation is reflected into the view and vice-versa.

Upon this view, *CP(Graph)* enforces directly the constraint by enforcing each vertex (except for the initial and final one) to:

- Having exactly one predecessor iff the vertex is in the graph's *glb*
- Having exactly one successor iff the vertex is in the graph's *glb*

which is basically a direct translation of the problem into a network of constraints. The major difference between both methods is the underlying structure that is used to impose these constraints. In the case of *GRASPER* we fetched *a priori* all the predecessor and successor sets for each vertex, which as explained previously, is very time and space consuming, whereas *CP(Graph)* opted for developing a view over the graph raw domain structure which could provide very efficient access to the vertices adjacency sets.

In *GRASPER*, maintaining consistency for the *path* constraint implies sweeping

the graph raw domain entirely, for each predecessor and each successor sets of each vertex, whereas in *CP(Graph)* consistency maintenance requires only sweeping the domain once and updating the view.

However, *CP(Graph)*'s method still has some of the undesirable properties mentioned previously for the *GRASPER* implementation. First of all, albeit in a much smaller scale, there is still some overhead in maintaining consistency between the graph raw domain and the view since a change in the graph raw domain will require an entire sweep over it, in order to update the view. Secondly, using this method implies a duplication of memory usage, since the view is actually another data structure that stores the graph information but in a different way. Last, but not least, the problem of imposing constraints over vertices that may not be part of the solution remains and hence, the feeling of wasting resources needlessly persists.

In the next section we explain how we can use a lazy mechanism for imposing constraints that will both save considerable space and, most importantly, considerable time on imposing the *path* constraint and that can solve the mentioned problems of the methods used by *GRASPER* and *CP(Graph)*.

5 Lazy constraint imposing for the *path* constraint

As explained in the previous section, considerable space was required, in *GRASPER*, to store the predecessor and successor sets of all vertices belonging to the graph variable's *poss* as well as considerable time for obtaining those sets and maintaining their consistency.

While one is able to accept consuming space and time for storing and maintaining those sets for vertices that may become part of the solution one is not, however, able to accept consuming those resources for vertices that present no guarantee of becoming part of the solution.

What we are seeking is basically, a lazy mechanism for delaying, as much as possible, obtaining the predecessor and successor sets (and maintaining their consistency) of a given vertex until it is actually considered part of the solution.

This is easily achieved in the following way. First, and as done in the original implementation, the predecessor and successor sets for all the vertices already in the graph variable's *glb* are obtained and their cardinality is instantiated to 1 (except for the initial and final vertices, as explained previously), by the same reasons we mentioned in the previous section.

Secondly, all vertices in the graph variable's *poss* are iterated upon and an associative table is built, as before, but this time the vertices predecessor and successor sets will not be stored there. This time, only two integer variables are stored: one integer variable for the number of edges having the vertex as out-vertex, i.e., the number of predecessors of the vertex; and another integer variable for the number of edges having the vertex as in-vertex, i.e., the number of successors of the vertex.

This far, a considerable amount of memory has been spared since only two integer variables are stored for each of the vertices in the graph variable's *poss*.

After this initialization phase, one can reason upon the information present in

this table in order to perceive when to impose the actual degree constraint over the vertices. Consistency between the graph raw domain and the degree associative table is done whenever there is a change in the graph's domain (removal of vertex, removal of edge, addition of vertex, addition of edge), in the following way:

- If a vertex is removed from the graph then it is not being considered to make part of the solution and therefore no degree constraint should be posed upon it and thus the information present in the associative table, for that vertex, may be simply disregarded.
- If an edge is removed from the graph, an update of the table is performed. The successor counter for the in-vertex and the predecessor counter for the out-vertex are decremented. If any of these values reaches 0 then, clearly, the corresponding vertex cannot be part of the solution and, therefore can be removed from the graph.
- If a vertex is added to the graph then it may make part of the solution and therefore we are finally in the situation where one is able to accept consuming resources to obtain the vertex predecessor and successor set, to maintain their consistency and to impose that their cardinality is 1.
- If an edge is added to the graph, we are again in the situation where one is able to accept consuming the above mentioned resources and therefore the predecessor and successor sets of both end-vertices are obtained (if they have not already been obtained) and their cardinality is instantiated to 1.

This method is clearly very efficient in terms of memory consumption and it also manages not to waste time imposing heavy constraints on vertices that may not ever be part of a solution and thus, with this mechanism, we solve *GRASPER*'s problem of consuming resources for vertices that present no guarantee of becoming part of the solution.

Additionally, we improve, in space, on *CP(Graph)* since we do not need an actual duplication of the graph. Our associative degree table stores a pair of integers for each vertex, whereas *CP(Graph)* maintains a view over the graph raw domain, which is actually, a complete copy of the graph but with a different representation, more suitable for the operations required by the *path* constraint.

Finally, we also improve in time on *CP(Graph)* since every time a change in the graph domain occurs, we do not need an entire sweep over the domain in order to maintain consistency between the domain and the associative table. Since *GRASPER* has access to delta domain variables and these store information of what changed in a variable's domain we can, in constant time, determine what this change was in our propagators. Hence, whenever a change occurs, we just need to consult the delta domain variable, query what the change was and update the corresponding information in the associative table.

6 Results

In this section we start by describing the Metabolic Pathways problem, giving a general description, explaining how it can be modeled using graph domain variables and finally presenting the results for this problem, for both implementations of *GRASPER* and *CP(Graph)*.

Metabolic networks [17,1] 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. An experiment is performed on DNA cells and these mutated cells (called RNA cells) are placed on DNA chips, which contain specific locations for different strands, so when the cells are placed in the chips, the different strands will fit into their specific locations. Once placed, the DNA strands (which encode specific enzymes) are scanned and catalyse a set of reactions. Given this set of reactions the goal is to know which products were active in the cell, given the initial molecule and the final result. Figure 2 represents a possible pathway between two given molecules regarding the metabolic network of Figure 1.

A recurrent problem in metabolic networks pathway finding is that many paths take shortcuts, in the sense that they traverse highly connected molecules (act as substrates or products of many reactions). However there are some metabolic networks for which some of these highly connected molecules act as main intermediaries. In Figure 1 there are three highly connected molecules, represented by the grid-filled circles.

It is also possible that a path traverses a reaction and its reverse reaction: a reaction from substrates to products and one from products to substrates. Most of the time these reactions are observed in a single direction so we can introduce *exclusive pairs of reactions* to ignore a reaction from the metabolic network when the reverse reaction is known to occur, so that both do not occur simultaneously. Figure 1 shows the presence of five *exclusive pairs of reactions*, represented by 5 pairs of the ticker arrows.

Additionally, it is possible to have various pathways in a given metabolic experiment and often the interest is not to discover one pathway but to discover a pathway which traverses a given set of intermediate products or substrates, thus introducing the concept of *mandatory molecule*. These *mandatory molecules* are useful, for example, if biologists already know some of the products which are in the pathway but do not know the complete pathway. In Figure 1 we imposed the existence of a *mandatory molecule*, represented by a diagonal lined-filled circle.

In fact, the pathway represented in Figure 2 is the shortest pathway obtained from the metabolic network depicted in Figure 1 that complies with all the above constraints.

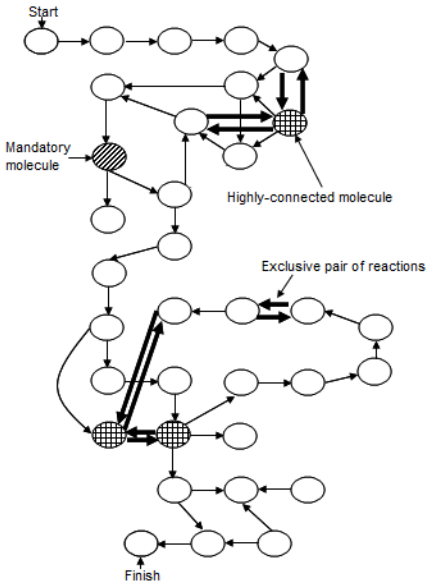

Figure 1 - Metabolic Network

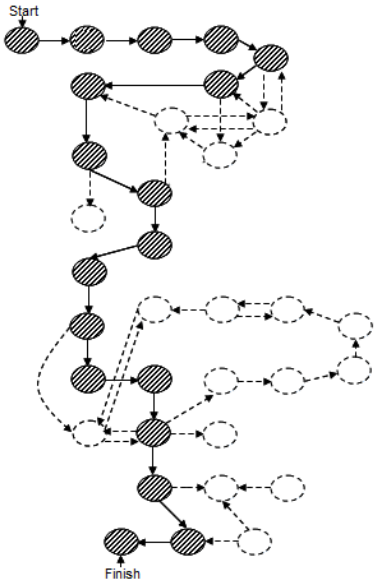

Figure 2 - Metabolic Pathway

Such network can be represented as a directed bi-partite graph, where the compounds, substrates and products represent one of the partition of the vertices and the reactions the other partition. The edges link compounds with the set of reactions and these to the substrates and the products. The search of a pathway between two vertices (the original molecule and a final product or substrate), *without* the mandatory molecules constraint, could be easily performed with a breadth-first [4,22] search algorithm.

Considering the problem of the highly connected molecules, a possible solution is to weight each vertex of the graph, where each vertex's weight is its degree (i.e. the number of edges incident on the vertex) and the solution consists in finding the shortest pathway of the metabolic experiment. This approach allows one to avoid these highly connected molecules whenever it is possible.

The *exclusive pairs of reactions* can also be easily implemented by introducing pairs of *exclusive* vertices, where as soon as it is known that a given vertex belongs to the graph the other one is instantly removed.

Finally, to solve the constraint of *mandatory molecules*, it is sufficient to add the vertices representing these molecules to the graph thus ensuring that any solution must contain all the specified vertices. With this mechanism, however, it is not guaranteed that the intended pathway is the shortest pathway between the given initial and final vertices (e.g. one of the *mandatory* vertices does not belong to the shortest path), so we cannot rely on breadth-first search and must find a different search strategy for solving this problem.

Basically, assuming that $G = dirgraph(V, E)$ is the original graph, composed

of all the vertices and edges of the problem, that v_0 and v_f are the initial and the final vertices, that $Mand = \{v_1, \ldots, v_n\}$ is the set of *mandatory* vertices, that $Excl = \{(v_{e11}, v_{e12}), \ldots, (v_{em1}, v_{em2})\}$ is the set of *exclusive* pairs of vertices and that W_f is a function mapping each vertex to its degree, this problem can be easily modeled in *GRASPER* as:

$$
\begin{gathered}
minimise(W) : \\
subgraph(dirgraph(SubV, SubE), G) \wedge \\
Mand \subseteq SubV \wedge \\
\forall (v_{ei1}, v_{ei2}) \in Excl : (v_{ei1} \notin SubV \vee v_{ei2} \notin SubV) \wedge \\
path(dirgraph(SubV, SubE), v_0, v_f) \\
weight(dirgraph(SubV, SubE), W_f, W)
\end{gathered}
$$

The minimisation function can be found built-in in almost every constraint programming environment. The subgraph relation is directly mapped to our *subgraph* constraint (consult [27] for details on the *subgraph* constraint) and its objective is to allow the extraction of the actual pathway from the original graph containing every vertex and edge from the original problem. The introduction of the *mandatory vertices* is easily achieved by a mere set inclusion operation. The *exclusive pairs of reactions* demand the implementation of a very simple propagator which basically removes one vertex once it is known that another vertex has been added to the graph and they form an *exclusive pair of reactions*. The weighting of the graph is performed using the *weight* constraint (consult [27] for details on the *weight* constraint). These simple operations sketch the basic modelling for this problem, however it is still necessary to perform search so as to trigger the propagators and determine the set of vertices that belong to the pathway and the edges that connect them.

We use a labeling strategy that consists in iteratively extending 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* [23].

Below, we present the results obtained for the problem of solving the shortest metabolic pathways for three metabolic chains (glycosis, heme and lysine) and for increasing graph orders (the order of a graph is the number of vertices that belong to the graph), having one instance per graph order. The instances were obtained from [6] and are the same ones used in [10,9].

We ran both implementations and *CP(Graph)*'s implementation [6] on an Intel Core 2 Duo 2.16 GHz, 4 Mb of L2 Cache, 1.5 Gb of RAM, on graph instances having from 500 to 2000 vertices and using the *shortest-path* heuristic. Table 1 presents the results, in seconds, where G_{old} denotes the original version of *GRASPER*, G_{new} the version with the lazy constraint imposing mechanism and *CP(Graph)*, the *CP(Graph)*'s implementation.

Order	Glycosis			Lysine			Heme		
	G_{old}	G_{new}	$CP(Graph)$	G_{old}	G_{new}	$CP(Graph)$	G_{old}	G_{new}	$CP(Graph)$
500	1.13	0.28	0.21	1.37	0.36	0.41	0.73	0.22	0.10
600	1.75	0.38	0.31	1.74	0.48	0.44	1.05	0.28	0.12
700	2.23	0.45	0.35	2.16	0.47	0.75	1.34	0.36	0.16
800	2.86	0.53	0.50	2.65	0.53	1.00	1.67	0.41	0.19
900	3.69	0.64	0.68	3.23	0.57	1.29	2.12	0.51	0.27
1000	4.85	0.77	0.84	3.57	0.60	1.37	2.62	0.62	0.32
1100	6.10	0.91	1.00	4.66	0.73	1.29	2.98	0.65	0.33
1200	6.60	0.96	1.08	5.76	0.86	2.23	3.73	0.80	0.41
1300	7.47	1.03	1.21	6.95	0.99	2.50	5.06	0.94	0.47
1400	9.12	1.23	1.56	7.99	1.12	2.84	5.12	1.11	0.51
1500	10.60	1.40	1.85	8.98	1.25	2.92	5.46	1.14	0.52
1600	12.50	1.67	2.14	9.80	1.30	2.97	6.60	1.35	0.61
1700	14.70	1.93	2.40	10.40	1.41	3.03	7.61	1.57	0.69
1800	16.70	2.11	2.77	12.00	1.53	3.69	8.69	1.72	0.77
1900	18.70	2.27	3.02	13.60	1.75	3.93	9.75	1.96	0.84
2000	19.50	2.40	3.14	15.30	1.96	2.18	10.80	2.18	0.91

Table 1
Metabolic Pathways Problem results for *GRASPER* versions and *CP(Graph)*

Analyzing the results obtained for both implementations of *GRASPER* we conclude that, for every instance of the problem and for all of the metabolic networks, the lazy constraint imposing mechanism has a major impact on the effectiveness of the application, managing to increase efficiency up to 8 times when comparing to the original version.

We also conclude that *GRASPER* is able to outperform *CP(Graph)* for the *glyco* chain, being able to improve 25% over *CP(Graph)*'s results on the higher instances. Regarding the *lysine* chain, *GRASPER* achieved a speed-up of 2 for some of the larger instances. Conversely, for the *heme* chain, *GRASPER* could not achieve the same results as *CP(Graph)*, taking sometimes twice the time of *CP(Graph)* to solve the problem. The heuristic used can find a solution for the instances of the *heme* chain very efficiently when we directly apply all the constraints, which may explain why the results obtained using the lazy constraint imposing mechanism were not as efficient as the ones obtained with *CP(Graph)*.

[6] We used version 1.3.1 of *GECODE* (available at http://ww.gecode.org) which is the last version upon which *CP(Graph)* runs on. *CP(Graph)* has been discontinued on *GECODE*.

7 Conclusions and future work

In this paper, *GRASPER*'s *path* constraint definition was presented, being specified how its first implementation was performed, showing that it used considerable space and used often too much time for imposing its constraints, although showing acceptable results when comparing to a state-of-the-art solver.

We proposed to use a lazy constraint imposing mechanism for a new implementation that could optimize used space and that would only spend time maintaining consistency on variables that would give some evidence of being part of the solution. We implemented a new version of *GRASPER* with such a mechanism and we compared it against the original one and against *CP(Graph)*, the state-of-the-art solver, and in that comparison, *GRASPER*'s new version was able to outperform the old version (by far) and also *CP(Graph)* for a large set of instances, appearing as a serious alternative to it.

Future work includes investigating where could this mechanism be also applied to, in order to further decrease space consumption and also to further improve the solver's efficiency.

We are also applying the same mechanism for undirected graph variables, which allowed us to discover a possible improvement in the graph variable domain representation that we also believe may substantially improve the solver's efficiency.

Additionally, we intend to implement solutions to other path constraining problems and using larger and more difficult instances in order to determine the limits of our application.

References

[1] Teresa Attwood and Douglas Parry-Smith. *Introduction to Bioinformatics*. Prentice Hall, 1999.

[2] Francisco Azevedo. Cardinal: A Finite Sets Constraint Solver. *Constraints journal*, 12(1):93–129, 2007.

[3] Gary Chartrand. *Introductory Graph Theory*. Dover Publications, 1984.

[4] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2 edition, 2001.

[5] Marco Correia, Pedro Barahona, and Francisco Azevedo. CaSPER: A Programming Environment for Development and Integration of Constraint Solvers. In F. Azevedo, C. Gervet, and E. Pontelli, editors, *Proceedings of the First International Workshop on Constraint Programming Beyond Finite Integer Domains (BeyondFD'05)*, pages 59 – 73, 2005.

[6] Didier Croes. *Recherche de chemins dans le réeseau métabolique et mesure de la distance métabolique entre enzymes*. PhD thesis, ULB, Belgium, 2005.

[7] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[8] Reinhard Diestel. *Graph Theory*. Springer-Verlag, 2005.

[9] Grégoire Dooms. *The CP(Graph) Computation Domain in Constraint Programming*. PhD thesis, Faculté des Sciences Appliquées, Université Catholique de Louvain, 2006.

[10] Grégoire Dooms, Yves Deville, and Pierre Dupont. CP(Graph): Introducing a Graph Computation Domain in Constraint Programming. In *Eleventh International Conference on Principles and Practice of Constraint Programming*, number 3709 in Lecture Notes in Computer Science, pages 211–225. Springer-Verlag, 2005.

[11] Michael Garey, David Johnson, and Larry Stockmeyer. Some simplified np-complete problems. In *STOC '74: Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 47–63. ACM, 1974.

[12] C. Gervet. Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language. *Constraints journal*, 1(3):191–244, 1997.

[13] Jeffrey Herrmann. *Handbook of Production Scheduling*. Springer-Verlag, 2006.

[14] Parag Lala. *Practical Digital Logic Design and Testing*. Prentice Hall, 1996.

[15] R. Lyon. Auctions and alternative procedures for allocating pollution rights. *Land Economics*, 58(1):16–32, 1982.

[16] Kim Marriot and Peter Stuckey. *Programming with Constraints: An introduction*. MIT Press, 1998.

[17] Christopher Mathews and Kensal van Holde. *Biochemistry*. Benjamin Cummings, 2 edition, 1996.

[18] J. McMillan. Selling spectrum rights. *Journal of Economic Perspectives*, 8(3):145–162, 1994.

[19] Michael Pinedo. *Planning and Scheduling in Manufacturing and Services*. Springer Series in Operations Research and Financial Engineering. Springer-Verlag, 2006.

[20] J.-F. Puget. Pecos: A high level constraint programming language. In *Proc. Spicis*, Singapure, 1992.

[21] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Elsevier Science Inc., 2006.

[22] Stuart Russel and Peter Norvig. *Artificial Intelligence: A modern approach*. Prentice Hall, 2002.

[23] Meinolf Sellmann. Cost-based filtering for shorter path constraints. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP)*, volume 2833 of *Lecture Notes in Computer Science*, pages 694–708. Springer-Verlag, 2003.

[24] Jacques van Helden, Lorenz Wernisch, David Gilbert, and Shoshana Wodak. *Graph-based analysis of metabolic networks*, pages 245–274. Springer-Verlag, 2002.

[25] Pascal van Hentenryck and Mehmet Dincbas. Domains in logic programming. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI'86)*, 1986.

[26] Ruben Viegas and Francisco Azevedo. GRASPER: A Framework for Graph CSPs. In Jimmy Lee and Peter Stuckey, editors, *Proceedings of the Sixth International Workshop on Constraint Modelling and Reformulation (ModRef'07)*, Providence, Rhode Island, USA, September 2007.

[27] Ruben Duarte Viegas. GRASPER: Constraint Reasoning with graphs. Master's thesis, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, February 2008.

[28] Junming Xu. *Theory and Application of Graphs*, volume 10 of *Network Theory and Applications*. Kluwer Academic Publishers, 2003.