A Mozart implementation of CP(BioNet)

Grégoire Dooms, Yves Deville, Pierre Dupont

Department of Computing Science and Engineering
Université catholique de Louvain
B-1348 Louvain-la-Neuve - Belgium
{dooms,yde,pdupont}@info.ucl.ac.be

Abstract

Abstract
Keywords: Mozart, Oz, Constraint Programming, Graph Domain Variables, Constrained Path Finding, Path Constraint

1 Introduction

Post genomic era blah blah

In [3, 4], we proposed a constraint programming approach to biochemical network analysis The goal is to be able to cover a broad range of analyses (including very computationally complex ones) by using a declarative query language and at the same time still be able to perform these analyses in reasonable time.

A first evaluation [3, 4] of the CP(BioNet) framework consisted in implementing a prototype and testing it against a complex problem: constrained path finding. This was done using Mozart-Oz. The results are encouraging both on the analysis and implementation sides of the problem.

This paper will focus on the implementation of the prototype of CP(BioNet) over the Oz-Mozart system. This includes the implementation of a new kind of domain variables, graph domain variables (from now on denoted gd-variables) and of a few propagators for constraints over these gd-variables. All the implementation was done in the Oz language, no C++ extension was involved.

Section 2 will describe the approach we used in CP(BioNet) to express a biochemical analysis as a subgraph finding problem then as a constraint program over gd-variables.

Section 3 will describe the Oz data structure we used for our first prototype of CP(BioNet), then some words will be said about a later and more efficient data-structure.

Section 4 will describe the implementation of a few propagators. Whenever it was possible, constraints available on the Mozart system were used. On the other hand, we implemented a stateful propagator for the path constraint.

Finally Section 5 will conclude with current and future work on this prototype.

2 CP(BioNet)

This section will briefly describe our biochemical networks modeling and our approach to their analysis. Then it will shortly describe CP(BioNet), a new constraint programming computing domain for the analysis of biochemical networks. CP(BioNet) introduces graph domain variables and constraints over these variables.

2.1 Biochemical networks model

Biochemical networks are networks representing the working of the cell. We adopted the aMAZE [1, 8] model of these networks. This model integrates many aspects of the functioning of the cell in an integrated model. It consist of a object oriented model for most of the data with additional relations to represent as many biological concepts as possible.

For the analysis of these networks, we model them as graphs whose nodes have attributes. The set of attributes attached to each node is determined according to the family of analyses under consideration. The simplest attributes are the three main classes present in the object-oriented aMAZE model: *entities*, *transforms* and *controls* (see Figure 1). Entities are the physical small objects in the cell: molecules, proteins, compounds, genes, mRNA, *etc.* Transforms link a set of entities to an other set of entities: reactions, gene transcription, mRNA translation, protein assembly, *etc.* Controls link an entity to either a transform or an other control: catalysis, inhibition, gene expression regulation, *etc.*

The described prototype uses non-oriented graphs but all the algorithms and data-structures have been extended and applied to oriented graphs. We keep non-oriented graphs in this paper for the sake of consistency and simplicity.

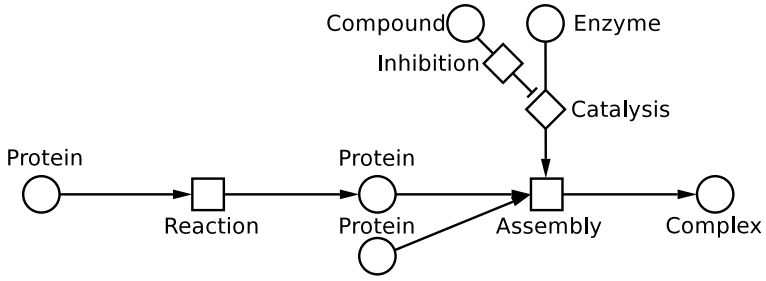

Figure 1: A small biochemical network in the object-oriented model containing bio-entities, transforms and controls.

2.2 Biochemical network analysis

The size of biochemical networks became gigantic since a few years and these networks are no longer printable as a whole (even on huge posters) nor possible to store in a single head. They were then stored in computers using models such as the aMAZE model. This computer storage of biochemical data leads to needs for specific data-mining tools. These tools are what biochemical network analysis stands for.

Biochemical networks analysis consists in answering user queries. We chose to model these queries as subgraph finding problems. The answer to a query is a graph extracted from the biochemical network under analysis. We think that kind of model covers a broad range of current and future queries about biochemical networks.

Queries like "Find the process transforming A into B in less than X steps", "Find all the paths expressed by a set of genes" or "Show how gene G is affected by entity E" are typical examples. They are translated into, respectively, "Find a path from A to B of length less than X, going only through entities and transforms", "Find the biggest subgraph containing no other gene than those given and respecting common biochemistry semantics rules (*e.g.* discard a reaction if its catalyst or one of its substrate is missing)", or "Find all the paths from any regulation node attached to the expression of gene G and node A".

2.3 CP(BioNet): constraint programming model

To model and solve these subgraph extraction problems, we designed CP(BioNet). CP(BioNet) consists of graph domain variables and constraints over these variables.

Graph domain variables are variables which initial domain is the set of all subgraphs of a reference graph. This reference graph is the maximum element of their initial domain. In the present work, it is assumed all gd-variables have the same initial domain, the same reference graph. Problems including the comparison of different graphs are not covered by this work.

The constraints over gd-variables currently defined and implemented are:

- The unary constraint $NodeInGraph(G, n)$ on the gd-variable G states the node n (of the reference graph of G) must be present in graph G.
- The unary constraint $ArcInGraph(G, a)$ on the gd-variable G states the arc a (of the reference graph of G) must be in present graph G.

– The unary constraint $EveryArc(G)$ on the gd-variable G states that if two nodes are in G and an arc joining these nodes belongs to the reference graph of G, then this arc must also belong to G.
– The binary constraint $SubGraph(P, G)$ on the gd-variables P and G states that P must be a subgraph of G (nodes and arcs of P must be in G too). P and G have the same reference graph.
– A constraint $Path(P, n_s, n_e, maxlength)$ states the gd-variable P must be a path from node n_s to node n_e (both in the reference graph of P) of length at most $maxlength$.
– A constraint $ExistsPath(G, n_s, n_e, maxlength)$ on the gd-variable G, derived from the $Path$ constraint but weaker, states that there must exist a path from n_s to n_e in G (and possibly other nodes and arcs). This is logically equivalent to the introduction of a new gd-variable P and using the $SubGraph(P, G)$ and $Path(P, n_s, n_e, maxlength)$ constraints. However, such an expression would be far too inefficient.
– The unary constraint $Connected(G)$ states that a gd-variable G must be a connected graph. This is semantically equivalent to stating that the $ExistsPath$ constraint must be satisfied for any pair of nodes in G.

$NodeInGraph$ and $ArcInGraph$ are reified constraints, they can be used in conjunction with logical operators to build more complex constraints. For $Path$ and $ExistsPath$, n_s and n_e must be determined and if $maxlength$ is a domain variable, the highest value of its domain is used.

3 The data structure used for graph domain variables

A gd-variable G can be implemented using boolean domain variables. A boolean variable per node in the reference graph states whether this node is present in the domain of the gd-variable. This vector of boolean variables is denoted $nodes(G)$. The presence of arcs in the domain of gd-variables is currently encoded with an adjacency matrix of boolean variables (see Figure 2). If N denotes the number of nodes in the reference graph, every gd-variable is represented with $N^2 + N$ boolean variables (actually half this number as the matrix is symmetric). This matrix is denoted $adjMat(G)$. Every graph domain variable has an associated constraint on its boolean domain variables to ensure that if an arc is present then both of its endpoint nodes must be present as well. Such a constraint can be implemented by a set of boolean constraints of the form

$$adjMat(G)_{ij} \Rightarrow nodes(G)_i \wedge nodes(G)_j$$

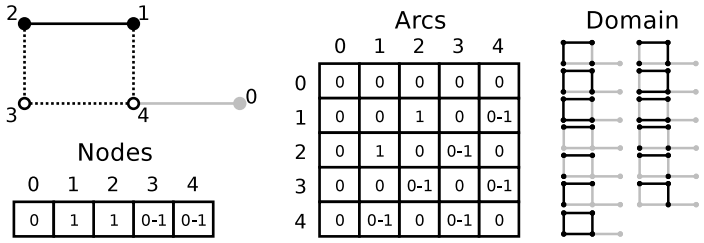

Figure 2: Implementation of a graph domain variable. The current domain of a variable, in the middle of the search process, is represented in this graph and coded in tables of boolean domain variables. A node or an arc is filled (nodes 1 and 2 and the arc joining them) when it is present in all graphs in the domain of the gd-variable. A light gray node or arc (node 0 and arc (0,4)) is never included in a graph of the domain. A dashed arc or unfilled node (all other nodes and arcs), may be present or absent in the graphs of the domain. All the graphs of the current domain of this gd-variable are displayed on the right.

The gd-variable itself is implemented as a class. We chose to use a class as a design tool not because we need to encapsulate a state. A new gd-variable is created by instantiation of the class and by telling it its domain using an init method. Two init methods are available: one states the domain of the variable (the reference graph), the other one states the variable is determined and gives its reference graph and value. The constraints are available as methods of the gd-variable instance. The instance variables of the class are:

1. the domain graph
2. the adjacency matrix
3. the array of node presence boolean variables

The adjacency matrix is implemented using a Tuple of Tuples of boolean variables (`V::0#1`). The node presence boolean variables are stored in a Tuple too.

That matrix is forced to be symmetrical by unifying symmetrical variables in the matrix. The built-in constraints for forcing that matrix and tuple of nodes to represent a graph are implemented using $N^2/2$ implication constraints (`FD.impl`).

In a later implementation, the adjacency matrix was replaced by an adjacency list: a Tuple of Records having a boolean variable only where the reference graph has an arc. This lead to an average twofold speedup relative to the test results showed in [3, 4]. A finite set implementation is currently being investigated.

4 Implementation of the constraint propagators

Most of the constraints listed above are very straightforward to implement using available constraints over boolean variables (or more generally finite domain variables):

- $NodeInGraph$ and $ArcInGraph$ are both reified. They just return the boolean variable under consideration.
- $EveryArc$ simply posts an implication constraint for each arc ij in the reference graph:

$$nodes(G)_i \wedge nodes(G)_j \Rightarrow adjMat(G)_{ij}$$

- $SubGraph(S, G)$ posts again a set of implications. For each node i in the reference graph:

$$nodes(S)_i \Rightarrow nodes(G)_i$$

  For each arc ij in the reference graph:

$$adjMat(S)_{ij} \Rightarrow adjMat(G)_{ij}$$

On the other hand, the $Path$, $ExistsPath$ and $Connected$ constraints were partly implemented using a stateful propagator of our own. This section will focus on the $Path$ constraint as the $ExistsPath$ constraint is just a little weaker and the $Connected$ constraint propagator is a piece of the $Path$ propagator.

Here is the sketch of the path propagator implementation:

4.1 The path propagator implementation

The propagator of the constraint $Path(P, n_s, n_e, maxlength)$ is implemented in three parts. The first part uses integer domain propagators provided by the Oz-Mozart system. The second part is implemented using standard graph algorithms. The third part uses more advanced graph algorithms to further reduce the domain of the gd-variable.

1. P is constrained to contain only nodes of degree one or two. The start node n_s and end nodes n_e have a degree of one, the other nodes have a degree of two. By stating this simple constraint, P is forced to contain a path from n_s to n_e and possibly some cycles on nodes not in the path (in Figure 3, a graph P consisting in a path from 0 to 4 and the cycle 5,6,7 is satisfying this first constraint). This first part of the propagator is implemented using the sum constraint on the rows of the adjacency matrix of the graph domain variable forcing the rows to contain exactly x (1 or 2) boolean variables with the value *true* (true is 1 while false is 0 in the sum):

$$\forall n \in \{n_s, n_e\} : \sum_j adjMat(P)_{n,j} = 1$$

$$\forall n \in nodes(P) \setminus \{n_s, n_e\} : \sum_j adjMat(P)_{n,j} = 2$$

These `FD.sum` constraints are posted when the path constraint is called on the gd-variable instance.

The cycles in other connected components are avoided by the second part of the propagator. It is also possible to constrain the number of nodes in the path using the *maxlength* information. A path of maximal length *maxlength* can contain at most $maxlength + 1$ nodes:

$$\sum_i nodes(P)_i \leq maxlength + 1$$

2. P is constrained to be a single connected component. This implies that P will only be the path from n_s to n_e as the cycles are in other connected components. A graph data structure *ConnGraph* is built. It is the supremum (with respect to graph inclusion) of all the graphs in the current domain of P. A node or an arc of the reference graph is not in *ConnGraph* if and only if its boolean variable in P is set to *false*. If this boolean variable is *true* or unknown (*i.e.* {*true*,*false*}) then the node/arc is in *ConnGraph*.

   This *ConnGraph* is implemented with a class. This class holds the *ConnGraph* data structure (a Dictionary of Dictionaries of integers) and methods to operate on it. A *ConnGraph* instance is associated to a gd-variable and stores the maximum element of its domain. We use threads watching each boolean variable of the gd-variable to keep this instance up to date with the domain of the gd-variable. The job of each thread is to wait until a boolean variable is determined and if it was set to false, update the *ConnGraph* accordingly.

   Each time[1] the boolean variable associated with an arc in the adjacency matrix is set to false, all the already included nodes of P (among those are n_s and n_e) are checked to see if they are still in the same connected component. Two cases can arise:

   - the constraint fails if they are not in the same connected component;
   - otherwise, all nodes and arcs in other components can be eliminated from the domain of P.

   A standard breadth-first depth-limited (*maxlength*) search in *ConnGraph* performs the connected component checking. During this search, all nodes in the same component as n_s are collected within a *maxlength* radius (if *maxlength* is an integer domain variable, the highest value of its domain is taken). As a by-product, the graph can be checked to see if it contains cycles. If there are no cycles, the connected component of *ConnGraph* starting from n_s is a tree. In that case, the graph P can be forced to be the only available path from n_s to n_e in *ConnGraph*. This is implemented with a depth-first search from n_s to n_e in *ConnGraph*.

   As we do not use an incremental algorithm [7] for the connected component checking, we avoid redoing this check for every arc deletion. Instead, we perform this connected component checking only when the computation space is stable. The stability check is not explicit: this stateful part of the propagator is automatically run by `FD.distribute generic(...)`. The propagation procedure to be run by the distributor is returned by the path constraint method.

3. Parts 1 and 2 guarantee to find a solution whenever there is one. An additional routine improves the propagation by detecting as soon as possible that some arcs must or must not belong to the graph P.

   A *bridge* in a connected component of a graph is an arc the removal of which breaks the connected component into two unconnected components. A connected component is said to be *2-edge connected* if it does not contain any bridge. A 2-edge connected component algorithm is used to find all bridges in *ConnGraph* [6, 5, 2]. It uses *BridgeTree*, an additional data structure representing a tree. The nodes of this tree correspond to the 2-edge components of *ConnGraph* and its arcs are the bridges of *ConnGraph*. Two nodes of *BridgeTree* are labeled

[1] In theory. See last paragraph of the description of this algorithm step.

$n1$ and $n2$, corresponding respectively to the 2-edge connected component of *ConnGraph* containing n_s and n_e (see Figure 4).

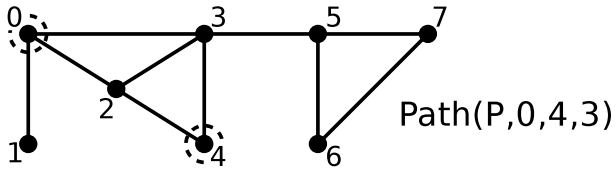

Figure 3: The Path constraint. The graph domain variable must be a path from 0 to 4 and include at most 3 arcs (at most 2 additional nodes). Nodes 0 and 4 are outlined in the reference graph.

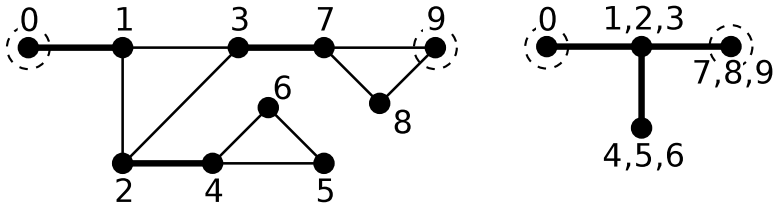

Figure 4: *BridgeTree* on the right representing the 2-edge connected components and the bridges of the graph on the left. The bridge (2,4) and the 2-edge connected component 4,5,6 cannot be part of the path from 0 to 9 while both other bridges must be in that path.

In this *BridgeTree*, all arcs on the path from $n1$ to $n2$ must be in P and all other arcs (and the 2-edge connected components on the other end) cannot be present in P. This information is propagated by adding or removing these arcs and nodes from the domain of P.

The *BridgeTree* is just a theoretic definition. It is not built by the implementation. The selection of positive and negative bridges is implemented using the previously cited algorithm [6, 5, 2] which computes a DFS spanning tree of *ConnGraph* (stored as an adjacency list over the nodes of *ConnGraph*: Tuple of Dictionaries). The "Low" values (lowest node reachable from each node) are then computed in this tree which enables to find all bridges. A depth first search in the tree allows to find a path from n_s to n_e and all bridges on this path are the bridges to be included in the gd-variable while all others can be taken out of the domain.

A similar reasoning can be made about *cut-nodes* (nodes the removal of which breaks the connected component) and the same algorithm can take care of these nodes.

5 Conclusion

This paper showed how we used Oz-Mozart to implement a prototype of CP(BioNet) a new computing domain in constraint programming. A new type of variables, graph domain variables were designed and implemented using the Oz language. New constraints were designed and implemented using the Oz language too. Many time was saved by re-using domain variables and constraints available in the Mozart system modules. Another advantage of the Mozart system is we could implement this prototype in C if we had efficiency troubles.

We intend to design and implement a specific distributor and an optimization search engine (using branch and bound). New constraints are also under investigation.

References

[1] The aMAZE data-base project. `http://www.amaze.ulb.ac.be/`.

[2] Joëlle Cohen. Théorie des graphes et algorithmes. Course notes. `http://www.univ-paris12.fr/lacl/cohen/poly_gr.ps`.

[3] G. Dooms, Y. Deville, and P. Dupont. Constrained path finding in biochemical networks. In *Proceedings of JOBIM 2004*, pages JO–40, 2004.

[4] G. Dooms, Y. Deville, and P. Dupont. Recherche de chemins contraints dans les réseaux biochimiques. In F. Mesnard, editor, *Programmation en logique avec contraintes, actes des JFPLC 2004*, pages 109–128. Hermes Science, 2004.

[5] Michel Gondran and Michel Minoux. *Graphes et algorithmes*. Eyrolles, 1995. 3ème éd.

[6] Jonhatan Gross and Jay Yellen. *Graph Theory and its Applications*. CRC Press, 1999.

[7] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal ACM*, 48(4):723–760, 2001.

[8] Chrisian Lemer, Erick Antezana, Fabian Couche, Frédéric Fays, Xavier Santolaria, Rekin's Janky, Yves Deville, Jean Richelle, and Shoshana J. Wodak. The aMAZE lightbench: a web interface to a relational database of cellular processes. *Nucleic Acids Research*, 32:D443–D448, 2004.