A Mozart Implementation of CP(BioNet)

Grégoire Dooms, Yves Deville, and Pierre Dupont

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

Abstract. The analysis of biochemical networks consists in studying the interactions between biological entities cooperating in complex cellular processes. To facilitate the expression of analyses and their computation, we introduced CP(BioNet), a constraint programming framework for the analysis of biochemical networks. An Oz-Mozart prototype of CP(BioNet) is described. This prototype consists of the implementation of a new kind of domain variables, graph domain variables, and the implementation of constraint propagators for constraints over graph-domain variables. These new variables and constraints are implemented in Oz and they can then be used like other domain variables in the Oz-Mozart platform. An implementation of a path constraint propagator is described in depth and constrained path finding tests are analysed to assess the tractability of our approach. Finally, an alternative Oz-Mozart data-structure for the graph-domain variables is presented and compared to the first one.

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

1 Introduction

Biochemical networks are the networks describing the entities and interactions between entities in the cells. Some network models focus on some aspects of the cell [9, 2] while others [12, 1, 13] try to represent as much data as possible in a unified way.

Analyzing biochemical networks is an important issue to improve the understanding of the working of a cell. The analysis of such networks typically consists in answering (parameterized) queries such as:

- find the process(es) transforming A into B in less than X steps,
- find the genes whose expression is affected by entity A,
- find the compounds deriving from a given entity A in less than X steps,
- find the pathways including the list L of entity, ligand, reaction, etc..

Several projects (aMaze [1], KEGG [14], BioCyc [12], Um-BBD [6], Emp [7], PathDB [15], CSNDB [16]) provide a set of predefined queries as those listed

P. Van Roy (Ed.): MOZ 2004, LNAI 3389, pp. 237–250, 2005.

above. Such queries cover several analyses thanks to the choice of their parameters (denoted in capitals in our examples). Available queries are however usually limited to simple ones which can be answered by the database management system or by simple ad-hoc routines.

More advanced queries are interesting from a biological viewpoint but they may require a significant design and programming effort while covering less generic analyses. Combining and/or extending analyses, as well as designing new analyses require lot of programming effort that cannot be reused for other analyses.

In [4, 5], 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 still be able to perform these analyses in reasonable time.

A first evaluation [4, 5] of the CP(BioNet) framework consisted in implementing a prototype and testing it against a complex problem: constrained path finding. The implementation was done using Oz-Mozart. The results of this evaluation are:

- Different and complex analyses of biochemical networks can be done using CP(BioNet).
- Oz-Mozart is adequate to prototype a new computation domain with new variables and propagators.

During the implementation of CP(BioNet), we found Oz-Mozart possesses interesting qualities with respect to other constraint systems. First, it is free and open-source. Second, Oz-Mozart supports functional and procedural programming which can sometimes be more natural than rule-based programming for programming new domains and propagators. It supports several types of domain variables: finite domain variables (as a special case, boolean variables) and finite set variables. Finally, it's object-orientation and its higher-order approach of constraint propagation makes it easily extendable. Our new graph domain variable can then be seen as a new primitive domain variable for the programmer.

This paper focuses 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 is done in the Oz language, no C++ extension is involved.

Section 2 describes the approach used in CP(BioNet) to express a biochemical analysis as a subgraph finding problem then as a constraint program over gd-variables. Section 3 describes the Oz data structure used for our first prototype of CP(BioNet), then some words will be said about another more efficient data-structure. Section 4 describes the implementation of a few propagators. Constraints available in the Mozart system were used whenever possible but a stateful propagator was necessary for the path constraint. Finally section 6 concludes 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 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 adopt the aMAZE [1, 13] model of these networks. This model integrates many aspects of the functioning of the cell in an integrated model. It consists of an object oriented model with 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 Fig. 1). Entities are the physical small objects in the cell: molecules, proteins, compounds, genes, mRNA, etc.. Transforms link a set of entities to another set of entities: reactions, gene transcription, mRNA translation, protein assembly, etc.. Controls link an entity to either a transform or another control: catalysis, inhibition, regulation of gene expression, etc..

The described prototype uses undirected graphs but all the algorithms and data-structures have been extended and applied to directed graphs. We use undirected graphs in this paper for the sake of consistency and simplicity. Different types of arrow glyphs can be seen in Fig. 1. This is the classical representation of biochemical networks in the biological community. But as the type of an arrow is completely determined by the types of its end nodes, we use non-labelled arcs.

2.2 Analysis of Biochemical Networks

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

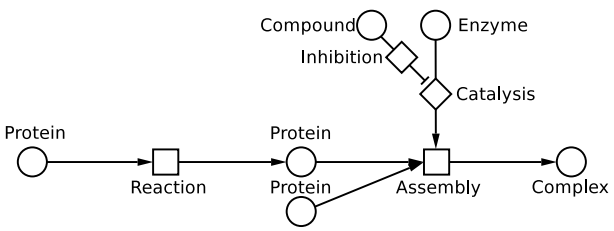

Fig. 1. A small biochemical network in the object-oriented model containing bio-entities, transforms and controls

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 raised needs for specific data-mining tools. These tools are what the term "biochemical network analysis" stands for.

Biochemical networks analysis consists in answering user queries about, for instance, the organization and potential interactions between the components of the cell. 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 to node E".

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 that all gd-variables have the same initial domain, that is 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 that 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 that the arc a (of the reference graph of G) must be present in 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 that 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 semantically 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.

In $NodeInGraph$ and $ArcInGraph$, the parameters n and a must be determined. $NodeInGraph$ and $ArcInGraph$ are reified constraints, they can be used as boolean variables in conjunction with first order logic operators to build more complex constraints (i.e. with a disjunction). 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 Fig. 2). If N denotes the number of nodes in the reference graph, every gd-variable is represented with $N^2 + N$ boolean variables (actually roughly 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

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

The gd-variable itself is implemented as a class. We chose to use a class for design matters (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 upper bound of the domain of the variable (the reference graph), the other one states that the variable is already determined and takes its reference graph and value as parameters. 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 vector of node membership boolean variables

The adjacency matrix is implemented using a Tuple of Tuples of boolean variables (`0#1`). The node membership boolean variables are stored in a Tuple. That matrix is forced to be symmetrical by unifying symmetrical variables in the

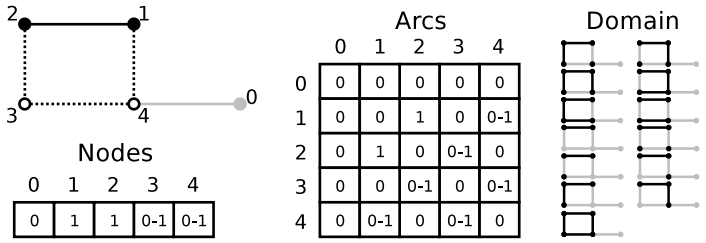

Fig. 2. Adjacency matrix implementation of a graph domain variable. The current domain of a variable, in the middle of the search process, is represented in this graph and coded in tables of boolean domain variables. A node or an arc is filled (nodes 1 and 2 and the arc joining them) when it is present 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

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 second implementation, the adjacency matrix is 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 [4, 5] (these results are plotted in Section 5). A finite set implementation is currently being investigated.

4 Implementation of the Constraint Propagators

In this section, the adjacency matrix of the gd-variable G is denoted $adjMat(G)$, the vector of node membership boolean variables of G is denoted $nodes(G)$. To refer to a specific boolean variable, the matrix is subscripted twice and the vector once.

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

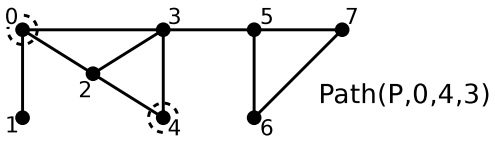

Fig. 3. The Path constraint. The graph domain variable must be a path from 0 to 4 and include at most 3 arcs (at most 2 additional nodes). Nodes 0 and 4 are outlined in the reference graph

For each arc ij in the reference graph:

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

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 slightly weaker and the *Connected* constraint propagator is part of the *Path* propagator.

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 Fig. 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

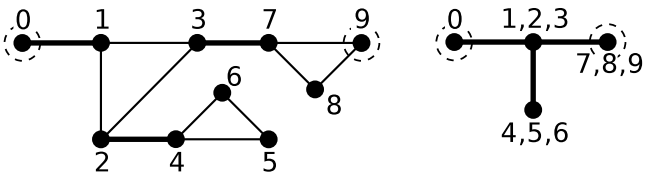

Fig. 4. *BridgeTree* on the right representing the 2-edge connected components and the bridges of the graph on the left. The bridge (2,4) and the 2-edge connected component 4,5,6 cannot be part of the path from 0 to 9 while both other bridges must be in that path

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 its value is false, update the *ConnGraph* accordingly.

   Each time 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) could be 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 [11] for the connected component checking, we avoid redoing this check for every arc deletion. Instead, this connected component checking is performed only when the computation space is stable (all other propagators have done their job). The stability check is not explicit: this stateful part of the propagator is automatically run by the generic distributor available in Mozart. The propagation procedure to be run by the distributor is returned by the path constraint method, the script passes this procedure to the distributor.

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* [10, 8, 3]. 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 $n1$ and $n2$, corresponding respectively to the 2-edge connected component of *ConnGraph* containing n_s and n_e (see Fig. 4).

   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 [10, 8, 3] 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 Experiments

Some experiments were conducted to assess the tractability of this framework for biochemical analyses. Constrained path finding tests were done using real

biological data. This section will first describe the data used for these tests. Then the constrained path finding tests are described along with their results. One other test will show the impact of the data structure used for the gd-variable: an adjacency matrix and an adjacency list implementation are compared.

5.1 Data

Graphs of increasing size (50, 100, 200, and 500 nodes) have been extracted from a metabolic network consisting of 4492 chemical entities and 5281 reactions. This data comes from the KEGG project and concerns two organisms: *Escherichia Coli* and *Saccharomyces Cerevisiae*. Extraction of smaller graphs from this network was performed while preserving approximately the degree distribution in the original graph. More precisely, an extracted graph must be a single connected component. The average degree of its nodes is around 4 and the maximum degree is 18 percent of its number of nodes.

5.2 Tests and Results

Five tests were performed on the extracted graphs. They are path finding problems expressed in CP(BioNet) using the *Path* constraint. The *maxlength* parameter was set to the number of nodes in the graph (no constraint on the length of the extracted path).

1. Path finding between two random nodes in the graph (always a solution since the graph is connected).
2. Path finding between two random nodes in the graph, with the additional constraint of containing two randomly preselected intermediate nodes.
3. Path finding between two random unconnected nodes in a double graph (two separate connected components were created by cloning the extracted graph; no solution).
4. Path finding between two random nodes in the graph, with the additional constraint of containing from one up to five randomly preselected intermediate node(s).
5. Selection of a random path p of k nodes in the graph. Path finding between the first and last nodes of p, with the additional constraint of containing from one up to $k-2$ intermediate nodes randomly preselected from p (always a solution).

The running time of every query was measured. For the first three tests, 1,000 queries were performed on each extracted graph. The fourth and fifth queries were performed on extracted graphs with 200 nodes. The fourth query was performed 1,000 times for every number of intermediate nodes. The fifth query was performed 1,000 times for every number of intermediate nodes and for values of k being 7, 10 and 15.

Figure 5 shows the average and standard deviation of the running time for these tests. Results from tests 2 and 4 are split in two groups: a curve for those where a solution was found and another for those for which no solution was found.

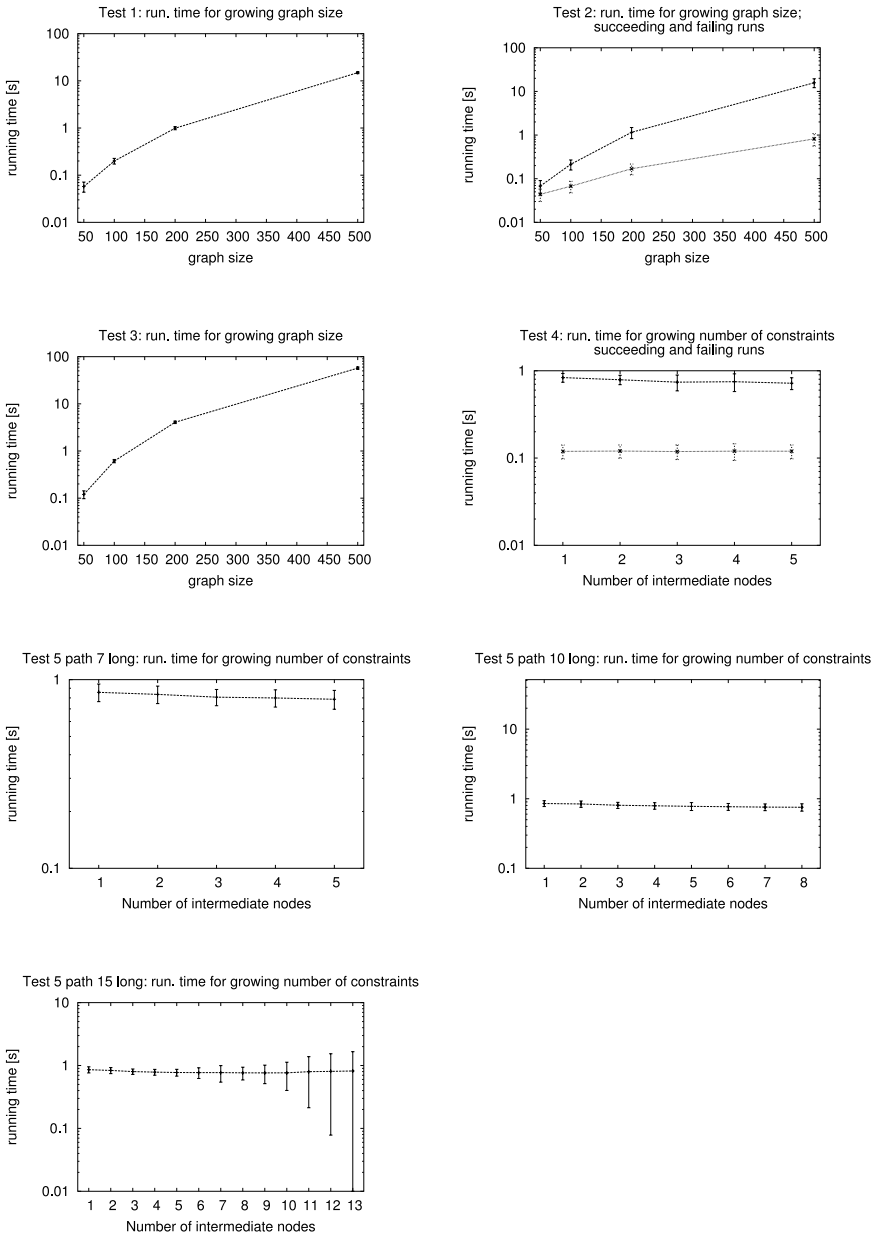

Fig. 5. Running time of the five tests. Logarithmic Y axis

5.3 Analysis

Tests 1 and 3 concern single path finding in a graph. This problem is not relevant alone for analyzing biochemical networks and dedicated algorithms are obviously

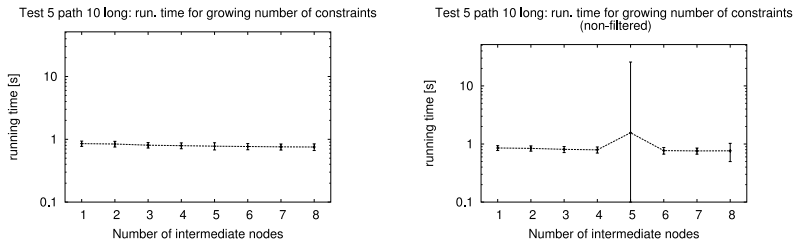

Fig. 6. Comparison of the results of test 5 path length of 10 when results are filtered (on the left) or not (on the right). The only difference lies in 2 runs (among the 8000 presented) for five intermediate nodes: one lasted 765 s and the other 18 s. The standard deviation is more affected by these rare results than the mean

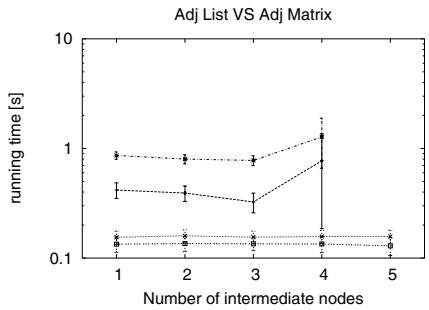

Fig. 7. Comparison of the adjacency matrix and adjacency list implementations of the graph-domain variable data-structure. The test is a test of random constrained path finding (test 4). The curves from top to bottom are "successful" queries with the matrix, with the list, then "failed" queries with matrix and with list

more efficient. These tests were done to analyze the path propagator on its own. For test 3, the reported size in the plots is the size of one component of the graph (the graph having twice that size). The plots for these tests show a sub-exponential curve and very low standard deviations. These tests illustrate the tractability of this propagator over increasing sizes of graphs.

Tests 2, 4, and 5 concern the constrained path finding problem. Two parameters were taken into account for this analysis: the size of the graph and the number of mandatory intermediate nodes. Test 2 shows the evolution of the running time of a query with 2 intermediate nodes versus the size of the graph. The plot shows two curves: one, for successful queries (the CSP solver found a path) and another, below, for failed queries (the CSP found no solution to this query). The results show that the curves are similar to the ones of the path propagator alone. The major difference is a larger standard deviation.

Tests 4 and 5 show the evolution of the running time on the graph of size 200 versus the number of mandatory intermediate nodes. Test 5 was performed to be able to show results of successful runs for high values of the number of intermediate nodes. When these nodes are chosen randomly in the graph (test 4), the odds of having a successful run are very low. The plots show that the average running time of these tests is nearly constant while the standard deviation has a slight tendency to grow.

A small fraction of the runs (from 0.08% up to 1%, depending on the tests) of the constrained path finding tests had running times several orders of magnitude worse than average. This somehow illustrates the NP-Hardness of these problems. Plots with and without these results are compared in Fig. 6. An additional test comparing an adjacency matrix (Tuple of Tuple) and an adjacency list (Tuple of Records) implementation of the gd-variable shows a near twofold speedup when using lists (see Fig. 7).

Our results show that the path constraint is tractable when used alone, although specialized algorithms are more efficient. When used along with other constraints (specifying a NP-Hard problem), the results show that the average running time is approximately the same (apart from rare diverging results) as the running time of the path constraint alone, independently of the number of additional constraints. Additional constraints on the type and attributes of the nodes of the biochemical network can thus be designed and used in our constrained path finding framework. This framework can then exploit the richness of the model of biochemical networks.

6 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, was designed and implemented using the Oz language. New constraints were designed and implemented as well. Much time was saved by reusing domain variables and constraints available in the Mozart system modules (boolean variables, propositional logic constraints, sum of finite domain variables constraint). Another advantage of the Mozart system is the possibility to implement this prototype in C/C++ if necessary.

CP(BioNet) allows the bio-informatician user to specify complex and diverse analyses using a declarative language and should provide him/her with an answer in reasonable time. Constrained path finding tests were conducted to assess the tractability of this framework in the average case. We also showed that this framework is expressive enough to state complex analyses. We now intend to use it on real problems from bio-informaticians.

We intend to design and implement a specific distributor and an optimization search engine (using branch and bound). Current constraints will be improved (definition, mode of usage, propagator, etc.) and new constraints are also under investigation.

References

1. The aMAZE data-base project. `http://www.amaze.ulb.ac.be/`.
2. G.D. Bader, I. Donaldson, C. Wolting, B.F. Ouellette, T. Pawson, and C.W. Hogue. Bind the biomolecular interaction network database. *Nucleic Acids Research*, 29(1):242-5, 2001.
3. Joëlle Cohen. Théorie des graphes et algorithmes. Course notes. `http://www.univ-paris12.fr/lacl/cohen/poly_gr.ps`.
4. G. Dooms, Y. Deville, and P. Dupont. Constrained path finding in biochemical networks. In *Proceedings of JOBIM 2004*, pages JO–40, 2004.
5. 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.
6. L.B.M. Ellis, B. Kyeng Hou, W. Kang, and L.P. Wackett. The university of minesota biocatalysis/biodegradation database : post-genomic data mining. *Nucleic Acids Research*, 31(1):262–265, 2002.
7. EMP project. Informations about EMP can be found at : `http://www.empproject.com/`.
8. Michel Gondran and Michel Minoux. *Graphes et algorithmes*. Eyrolles, 1995. 3ème éd.
9. S. Goto, T. Nishioka, and M. Kanehisa. LIGAND: Chemical database for enzyme reactions. *Bioinformatics*, 14:591–599, 1998.
10. Jonathan Gross and Jay Yellen. *Graph Theory and its Applications*. CRC Press, 1999.
11. 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.
12. P.D. Karp, M. Riley, M. Saier, I.T. Paulsen, J. Collado-Vides, S.M. Paley, A. Pelligrini-Toole, C. Bonavides, and S. Gama-Castro. The EcoCyc database. *Nucleic Acids Research*, 30(1):56–8, 2002.
13. 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.
14. K. Minoru, G. Susumu, K. Shuichi, and N. Akihiro. The KEGG databases at GenomeNet. *Nucleic Acids Research*, 30(1):42–46, 2002.
15. Faye Schilkey. PathDB : a pathway database. http://www.ncgr.org/pathdb.
16. Takako Takai-Igarashi and Tsuguchika Kaminuma. A pathway finding system for the cell signaling networks database. *Silico Biology*, 1:129–146, 1999.