GRASPER

A framework for graph constraint satisfaction problems

Student: Ruben Duarte Viegas[1] Supervisor: Francisco Azevedo
{rviegas,fa}@di.fct.unl.pt

CENTRIA, Departamento de Informática
Universidade Nova de Lisboa

Degree: Master in Computer Science
Year(s): 2006 – 2008

Abstract. In this paper we present GRASPER, a graph constraint solver, based on set constraints, that shows promising results when compared to an existing similar solver in this early stage of development.

Keywords: Constraint Programming, Graphs, Sets

1 Introduction

Constraint Programming (CP) [1, 2] has been successfully applied to numerous combinatorial problems such as scheduling, graph coloring, circuit analysis, or DNA sequencing. Following the success of CP over traditional domains, sets were also introduced [3] to more declaratively solve a number of different problems. Recently, this also led to the development of a constraint solver over graphs [4, 5], since a graph [6, 7] is composed of a set of vertexes and a set of edges.

Graph-based constraint programming can be declaratively used for path and circuit finding problems, possibly applying weights so as to be able to determine shortest or longest paths, to routing, scheduling and allocation problems, etc. CP(Graph) was proposed by G. Dooms et al. [4, 5] as a general approach to solve graph-based constraint problems. It provides a key set of basic constraints which represent the framework's core, and higher level constraints for solving path finding and optimization problems, and to enforce graph properties. CP(Graph) is integrated with the finite domain and finite sets computation domains and has implementations available in *GECODE* (http://www.gecode.org/) and in *Mozart* [8]. Developing a framework upon a finite sets computation domain allows us to abstract from many low-level particularities of set operations and focus entirely on graph constraining, consistency checking and propagation.

In this paper we present GRASPER (GRAph constraint Satisfaction Problem solvER) which is being developed in the Master of Computer Science thesis at Universidade Nova de Lisboa (UNL). GRASPER is an alternative framework for graph-based constraint solving based on *Cardinal* [9], a finite sets constraint

[1] Partially supported by PRACTIC - FCT-POSI/SRI/41926/2001

solver with extra inferences also developed in UNL. We present a set of basic constraints which represent the core of our framework and we provide functionality for directed graphs, graph weighting, graph matching, graph path optimization problems and some of the most common and desired graph properties. In addition, in this MSc work, we intend to integrate GRASPER in CaSPER [10], a programming environment for the development and integration of constraint solvers, using Generic Programming [11] methodology.

This paper is organised as follows. In section 2 we specify the details of our framework, starting with a brief introduction to *Cardinal*, followed by the presentation of our core constraints and other non-trivial ones. Then, in section 3 we describe a problem in the context of biochemical networks which we use to test our framework: we present a model for it together with search strategies to find the solution, and present experimental results, comparing them with the ones obtained with CP(Graph). We conclude in section 4.

2 GRASPER

In this section we start by briefly introducing *Cardinal*, the finite sets constraint solver upon which our framework is based and then we present the concepts which build up our framework: core constraints, non-trivial constraints and operational semantics.

2.1 *Cardinal*

Set constraint solving was proposed in [3] and formalized in [12] with ECLiPSe (http://eclipse.crosscoreop.com/eclipse/) 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 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, disjointness, containment, together with set operations such as union or intersection). *Conjunto* proved its usefulness in declarativeness and efficiency for NP-complete combinatorial search problems dealing with sets, compared to constraint solving over finite integer domains. Afterwards, *Cardinal* (also in ECLiPSe) [9], improved on *Conjunto* by extending propagation on set functions such as cardinality.

2.2 GRASPER Specification

In this subsection we explain how we defined our framework upon *Cardinal*.

A graph is composed by a set of vertexes and by a set of edges, where each edge connects a pair of the graph's vertexes. So, a possible definition for a graph is to see it as a pair (V, E) where both V and E are finite set variables and where each edge is represented by a pair (X, Y) specifying a directed arc from

X towards Y. In our framework we do not constrain the domain of the elements contained in those sets, so the user is free to choose the best representation for the constraint satisfaction problem. The only restriction we impose is that each incidence of an edge in the set of edges must be present in the set of vertexes.

In order to create graph variables we introduce the following constraint:

```
graph(G, V, E)      (G is a compound term of the form graph(V,E))
```

which is true if G is a graph variable whose set of vertexes is V and whose set of edges is E. Notice that since $E \subseteq V \times V$ the consistency between these two sets must be verified: if, for instance, the edge *(1,2)* belongs to the *glb* of the set of edges then both vertexes must belong to the *glb* of the set of vertexes; similarly, if a vertex does not belong to the *lub* of the set of vertexes then no edge incident on that vertex can belong to the *lub* of the set of edges.

All the basic operations for accessing and modifying the vertexes and edges is supported by *Cardinal*'s primitives, so no additional functionality is needed. Therefore, our framework provides the creation and manipulation of graph variables for constraint satisfaction problems just by providing a single constraint for graph variable creation and delegating to *Cardinal* the underlying core operations on sets.

While the core constraints of the framework allow basic manipulation of graph variables, it is useful to define some other, more complex, constraints based on the core ones thus providing a more intuitive and declarative set of functions for graph variable manipulation.

We provide a constraint to weight a graph in order to facilitate the modeling of graph optimization or satisfaction problems. The weight of a graph is defined by the weight of the vertexes and of the edges that compose the graph. Therefore, the sum of the weights of both vertexes and edges in the graph's *glb* define the lower bound of the graph's weight and, similarly, the sum of the weights of the vertexes and of the edges in the graph's *lub* define the upper bound of the graph's weight. The constraint is provided as $weight(G, W_f, W)$, where W_f is a map which associates to each vertex and to each edge a given weight, and can be defined as:

$$\begin{aligned} weight(graph(V,E), W_f, W) \equiv\ & m = \sum_{v\ \in\ glb(V)} W_f(v) + \sum_{e\ \in\ glb(E)} W_f(e) \wedge \\ & M = \sum_{v\ \in\ lub(V)} W_f(v) + \sum_{e\ \in\ lub(E)} W_f(e) \wedge \\ & W :: m..M \end{aligned}$$

This constraint is responsible for incrementing the lower bound of the graph's weight when either a vertex or an edge is added to the graph, for decrementing its upper bound when either a vertex or an edge is removed from the graph, for adding vertexes and edges when its lower bound increases and for removing vertexes and edges when its upper bound is decreased.

Additionally, we provide a subgraph relation $subgraph(SubGraph, Graph)$ stating that a graph $G_1 = graph(V_1, E_1)$ is a subgraph of a graph $G_2 = graph(V_2, E_2)$ iff $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$.

Obtaining the set of predecessors P of a vertex v in a graph G is performed by the constraint $preds(G, v, P)$ which can be expressed as:

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

Similarly, obtaining the successors S of a vertex v in a graph G is performed by the constraint $succs(G, v, S)$ which can be expressed as:

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

This last constraint obtains the set of the immediate successors of a vertex in a graph but we may want to obtain the set of all successors of that vertex, i.e., the set of reachable vertexes of a given initial vertex. Therefore, we provide a $reachables(G, v, R)$ which can be expressed in the following way:

$$\begin{aligned} reachables(graph(V, E), v, R) \equiv\ & R \subseteq V \wedge \\ & \forall r\ \in V : (r \in R \equiv \exists p : p \in paths(graph(V, E), v, r)) \end{aligned}$$

stating that a set of vertexes each reachable from another if there is a path between this last vertex and each of those vertexes. The rule *paths* represents all possible paths between two given vertexes and $p \in paths(graph(V, E), v, r)$ can be expressed as:

$$p \in paths(graph(V, E), v_0, v_f) \equiv \begin{cases} v_0 \in V \wedge p = \emptyset & , if\ v_0 = v_f \\ \\ \exists v_i \in V : (v_0, v_i) \in E \wedge \\ \exists p' : p' \in paths(graph(V, E), v_i, v_f) \wedge & , if\ v_0 \neq v_f \\ p = cons(v_0, p') \end{cases}$$

Additionally, this last constraint will allow us to build other very useful ones. For example, we can make use of the $reachables/3$ constraint to develop the connectivity property of a graph. By [6], a non-empty graph is said connected if any two vertexes are connected by a path, or in other words, if any two vertexes are reachable from one another. In a connected graph, all vertexes must reach all the other ones, so we define a new constraint $connected(G)$ which can be expressed as:

$$connected(graph(V, E)) \equiv \forall v \in V : reachables(graph(V, E), v, R) \wedge R = V$$

Another useful graph property is that of a path: a graph is said a path between an initial vertex v_0 and a final vertex v_f if there is a path between those vertexes in the graph and all other vertexes belong to the path and are visited only once. We provide the $path(G, v_0, v_f)$ constraint, which can be expressed in the following way:

$$path(G, v_0, v_f) \equiv quasipath(G, v_0, v_f) \wedge connected(G)$$

This constraint delegates to $quasipath/3$ the task of restricting the vertexes that are or will become part of the graph to be visited only once and delegates to $connected/1$ the task of ensuring that those same nodes belong to the path between v_0 and v_f so as to prevent disjoint cycles from appearing in the graph.

The $quasipath(G, v_0, v_f)$ constraint (for directed graphs) can be expressed as:

$$quasipath(graph(V,E),v_0,v_f) \equiv \forall v \in V \begin{cases} succs(graph(V,E),S) \wedge \\ \#S = 1 & ,if\ v = v_0 \\ preds(graph(V,E),P) \wedge \\ \#P = 1 & ,if\ v = v_f \\ preds(graph(V,E),P) \wedge \\ \#P = 1 \wedge \\ succs(graph(V,E),S) \wedge \\ \#S = 1 & ,otherwise \end{cases}$$

3 Results

In this section we describe a problem in biochemical networks, model it, and present obtained results, comparing them with CP(Graph).

3.1 Pathways

Metabolic networks [13, 14] 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. In Fig. 1 we present a metabolic network, and in Fig. 2, a possible pathway between the imposed start and finish molecules.

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 catalyze 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. Fig. 2 represents a possible pathway between two given nodes regarding the metabolic network of Fig. 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) and therefore cannot be considered as belonging to an actual pathway. However there are some metabolic networks for which some of these highly connected molecules act as main intermediaries. In Fig. 1 there are three highly connected compounds, 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.

Fig. 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 Fig. 1 we imposed the existence of a *mandatory molecule*, represented by a diagonal lined-filled circle.

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

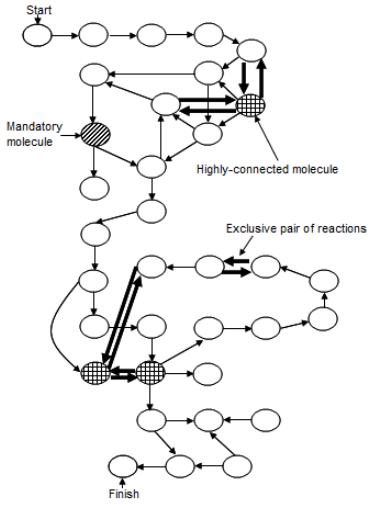

Fig. 1. Metabolic Network

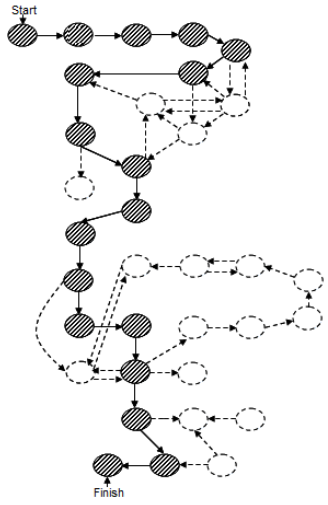

Fig. 2. Metabolic Pathway

3.2 Modeling Pathways

Such network can be represented as a directed bi-partite graph, where the compounds, substrates and products represent one of the partition of the vertexes 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 nodes (the original molecule and a final product or substrate) can be easily performed with a breadth-first 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* vertexes, 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 vertexes representing these molecules to the graph thus ensuring that any solution must contain all the specified vertexes. With this mechanism, however, it is not guaranteed that the intended pathway is the shortest pathway between the given initial and final vertexes (e.g. one of the *mandatory* vertexes does not belong to the shortest path), so we cannot rely on breadth-first search again and must find a different search strategy for solving this problem.

Basically, assuming that $G = graph(V, E)$ is the original graph, composed of all the vertexes and edges of the problem, that v_0 and v_f are the initial and the final vertexes, that $Mand = \{v_1, \ldots, v_n\}$ is the set of *mandatory* vertexes, that $Excl = \{(v_{e11}, v_{e12}), \ldots, (v_{em1}, v_{em2})\}$ is the set of *exclusive* pairs of vertexes and that W_f is a map associating each vertex and each edge to its weight, this problem can be easily modeled in GRASPER as:

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

3.3 Search Strategies

The minimization of the graph's weight ensures that one obtains the shortest pathway constrained to contain all the *mandatory vertexes* and not containing both vertexes of any pair of *exclusive* vertexes. However, it may not uniquely determine all the vertexes (the non-mandatory) which belong to that pathway. This must then be achieved by labeling functions which, in graph problems, decide whether or not a given vertex or edge belongs to the graph.

In this section we describe two search heuristics for the problem of finding the shortest past of the metabolic network subject to the aforementioned constraints.

Starting with the given *mandatory vertexes* and incrementally adding vertexes to the graph we can recursively obtain possible graphs obeying the path constraint. Having that set of vertexes that may possibly verify that path in the graph, it suffices to determine which edges belong to the graph and actually define the path. This verification can be performed employing a naive labeling function, which, for each possible edge (one that can still be added to or removed from the graph), first tries to add it to the graph and, on backtracking, removes it. This heuristic shall henceforth be referred to as *naive*.

Another heuristic is to adopt a *first-fail* heuristic: in each cycle we start by selecting the most constrained vertex and label the edge linking it to its least constrained successor. The most constrained vertex is the one with the lowest

out-degree and the least constrained successor vertex is the one with the highest in-degree. This heuristic is greedier than the naive one in the sense that it will direct the search towards the most promising solution. If the set of vertexes that constitute the graph at any given time is part of the solution then this heuristic may outperform the *naive* one. This heuristic shall henceforth be referred to as *first-fail*.

3.4 Experimental Results

In this subsection we present the results (in seconds) obtained for the problem of solving the shortest metabolic pathways for each of the metabolic chains and for increasing graph orders (the order of a graph is the number of vertexes that belong to the graph).

In Table 1 we present the results obtained with our prototype using the *naive* heuristic and the results obtained by CP(Graph) (presented in [4]) employing the *first-fail* heuristic (GRASPER on an Intel Pentium(R) D CPU 3.4 GHz, 1 Gb of RAM; CP(Graph) on an Intel Xeon 2.66 GHz, 2 Gb of RAM.)

	GRASPER			CP(Graph)		
Order	Glyco	Heme	Lysine	Glyco	Heme	Lysine
50	0,4	0,7	12,5	0,2	0,2	0,2
100	8,6	18,6	548,2	2,5	0,3	4,7
150	11,3	38,5	773,8	41,7	1,0	264,3
200	25,1	32,2	1121,8	55,0	398,8	N.A.
250	43,1	60,5	1728,2	127,6	173,3	N.A.
300	979,4	112,0	1982,1	2174,4	1520,2	N.A.

Table 1. Results obtained for GRASPER and CP(Graph).

In Fig. 3 the speed-up of GRASPER relative to CP(Graph) can be seen, showing that GRASPER presents better results than CP(Graph) for instances of the problem with order of at least 150. The speed-up was calculated as the quotient between CP(Graph) and GRASPER.

CP(Graph) only produces better results for graphs of order 50 and 100 and for the heme chain of order 150. However, this trend is clearly reversed for higher order instances: results for the glycose chain outperform the ones obtained by CP(Graph) from order 150 above, and for the graph of order 300 we achieve almost 20 minutes less; results for the heme chain for graphs of order above 150 are all under 200 seconds managing to decrease the expected time from 1520 to 112 seconds; finally for the lysine chain we could obtain results for instances of order above 150, for which CP(Graph) presents no results.

The comparison between these two frameworks seems to indicate that GRASPER outperforms CP(Graph) for larger problem instances thus providing a more scalable framework.

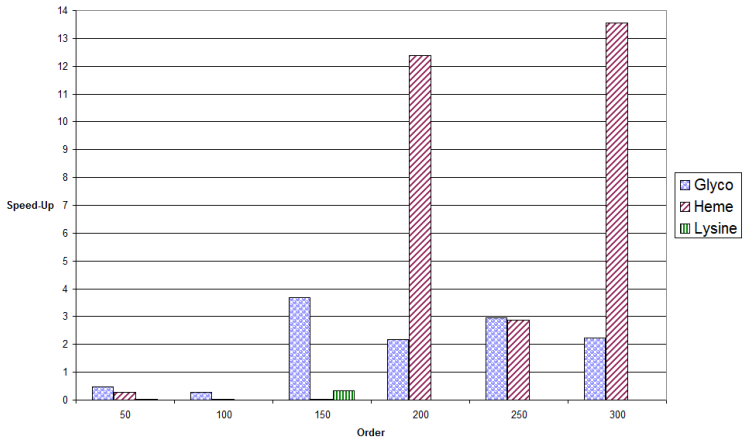

Fig. 3. Grasper's speed-up relative to CP(Graph)

Some important remarks regarding the heme and lysine chains must be made. While the former presented some of the best results, achieving an astonishing result for the graph of order 300, the latter exhibited the worst results, taking more than half an hour long for instances of order 250 and above. We believe that better results can be achieved with the use of a more specialized heuristic.

Even though we used a naive heuristic for the search process, GRASPER still obtained fairly good results, especially for instances of higher order. Thus, one can expect that the implementation of a first-fail heuristic similar to the one used by CP(Graph) (as presented in [5]) can bring some improvements. We also believe that these results can be further improved by introducing cost based filtering which allows introducing bounding in the search as a way to filter models which will not improve the cost of the path.

4 Conclusions and Future Work

In this paper we presented GRASPER, a new framework for the development of graph-based constraint satisfaction problems. This framework being built upon *Cardinal* [9] allows for a clear and concise manipulation of the elements that constitute a graph, the sets of vertexes and edges, and thus appears as a simple and intuitive interface just by defining a few additional rules for graph creation, manipulation and desirable graph properties.

We tested GRASPER with a problem in the context of biochemical networks (metabolic pathways) and compared results with CP(Graph) [4]. Even though CP(Graph) presented better results for small problem instances, GRASPER clearly outperformed it for larger ones, achieving speed-ups of almost 15 even using a naive heuristic.

However, since the presentation of CP(Graph) results in [4], newer and more efficient results were published [5] using a different first-fail strategy based on [15], and a cost-based filtering [16] mechanism to further constrain search space.

Future work includes the incorporation of these techniques since we believe it will help improving efficiency for this and similar problems. We also plan to improve the internal *Cardinal* data structure, since it currently requires linear time cost for the most common rules used in GRASPER, to substantially improve its overall performance. We are also extending our framework to allow manipulation of undirected graphs, which, in some cases, make use of the underlying constraints differently when compared with directed graphs and we will also implement other useful graph properties to provide a more intuitive and easy to use interface to model graph-based constraint satisfaction problems.

References

1. E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London, 1993.
2. K. Marriot and P. J. Stuckey. *Programming with Constraints: An introduction*. MIT Press, 1998.
3. J.-F. Puget. Pecos: A high level constraint programming language. In *Proc. Spicis*, Singapure, 1992.
4. G. Dooms, Y. Deville, and P. Dupont. Cp(graph): Introducing a graph computation domain in constraint programming. In *11th Int. Conf. on Principles and Practice of Constraint Programming*, number 3709 in Lecture Notes in Computer Science, pages 211 – 225, Barcelona, 2005. Springer-Verlag.
5. G. Dooms. *The CP(Graph) Computation Domain in Constraint Programming*. PhD thesis, Faculté des Sciences Appliquées, Université Catholique de Louvain, Louvain-La-Neuve, 2006.
6. R. Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer-Verlag, third edition, 2005.
7. F. Harary. *Graph Theory*. Addison-Wesley, 1969.
8. P. Van Roy and S. Haridi. Mozart: A programming system for agent applications. In *Int. Conf. on Logic Programming (ICLP 99)*, 1999.
9. F. Azevedo. *Cardinal: A Finite Sets Constraint Solver*, volume 12 of *Constraints journal*, pages 93 – 129. Kluwer Academic Publishers, 2007.
10. M. Correia, P. Barahona, and F. Azevedo. Casper: A programming environment for development and integration of constraint solvers. In Azevedo et al., editor, *Proc. of the First Int. Workshop on Constraint Programming Beyond Finite Integer Domains (BeyondFD'05)*, pages 59 – 73, 2005.
11. D. Musser and A. Stepanov. Generic programming. In *ISSAC*, pages 13 – 25, 1988.
12. C. Gervet. *Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language*, volume 1 of *Constraints journal*, pages 191 – 244. Kluwer Academic Publishers, 1997.
13. C. Mathews and K. Van Holde. *Biochemistry*. Benj./Cumm., second edition, 1996.
14. T. Attwood and D. Parry-Smith. *Introduction to bioinformatics*. Prent. Hall, 1999.
15. Y. Caseau and F. Laburthe. Solving small TSPs with constraints. In *Int. Conf. on Logic Programming*, pages 316 – 330, 1997.
16. M. Sellmann. Cost-based filtering for shorter path constraints. In *9th Int. Conf. on Principles and Practice of Constraint Programming (CP)*, volume 2833 of *LNCS*, pages 694 – 708. Springer-Verlag, 2003.