Speeding up constrained path solvers with a reachability propagator

Luis Quesada, Peter Van Roy, and Yves Deville
Université catholique de Louvain
Place Sainte Barbe, 2, B-1348 Louvain-la-Neuve, Belgium
{luque, pvr, yde}@info.ucl.ac.be

Abstract

Constrained path problems have to do with finding paths in graphs subject to constraints. One way of constraining the graph is by enforcing reachability on nodes. For instance, it may be required that a node reaches a particular set of nodes by respecting some restrictions like visiting a particular set of nodes or edges and using less than a certain amount of resources. The reachability constraints of this paper were suggested by a practical problem regarding mission planning in the context of an industrial project.

We deal with this problem by using concurrent constraint programming where the problem is solved by interleaving Propagation and Labeling. In this paper, we define a propagator which we call *Reachability* that implements a generalized reachability constraint on a directed graph g. Given a source node $source$ in g, we can identify three parts in the *Reachability* constraint: (1) the relation between each node of g and the set of nodes that it reaches, (2) the association of each pair of nodes $\langle source, i\rangle$ with its set of cut nodes, and (3) the association of each pair of nodes $\langle source, i\rangle$ with its set of bridges.

We show the effectiveness of our *Reachability* propagator by applying it to the Simple Path problem with mandatory nodes. We do an experimental evaluation of *Reachability* that shows that it provides strong pruning, obtaining solutions with very little search. Furthermore, we show that *Reachability* is also useful for defining a good labeling strategy and dealing with ordering constraints among mandatory nodes. These experimental results give evidence that *Reachability* is a useful primitive for solving constrained path problems over graphs.

1 Introduction

Constrained path problems have to do with finding paths in graphs subject to constraints. One way of constraining the graph is by enforcing reachability on nodes. For instance, it may be required that a node reaches a particular set of nodes by respecting some restrictions like visiting a particular set of nodes or edges and using less than a certain amount of resources. We have instances of this problem in Vehicle routing [14, 4, 9] and Bioinformatics[7].

An approach to solve this problem is by using Concurrent Constraint Programming (CCP) [18, 13]. In CCP, we solve the problem by interleaving two processes: propagation and labeling. In Propagation, we are interested in filtering the domains of a set of finite domain variables according to the semantics of the constraints that have to be respected. In Labeling, we are interested in specifying which alternative should be selected when searching for the solution.

Our goal is to implement so-called *Constrained Path Propagators (CPPs)* for achieving global consistency [6]. In this paper, we define a propagator which we call *Reachability* that implements a generalized reachability constraint on a directed graph g. Given a source node $source$ in g, we can identify three parts in the *Reachability* constraint: (1) the relation between each node of g and the set of nodes that it reaches, (2) the association of each pair of nodes $\langle source, i\rangle$ with its set of cut nodes, and (3) the association of each pair of nodes $\langle source, i\rangle$ with its set of bridges.

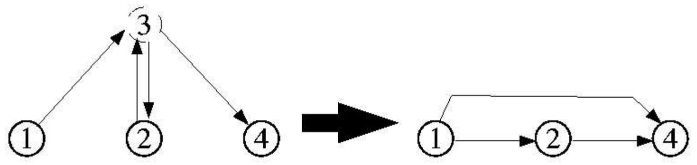

Figure 1: Relaxing Simple Path with mandatory nodes by eliminating the optional nodes

Our contribution is a propagator that is suitable for solving the Simple Path problem with mandatory nodes [19, 3]. This problem consists in finding a simple path in a directed graph containing a set of mandatory nodes. A simple path is a path where each node is visited once. Certainly, this problem can be trivially solved if the graph has no cycles since in that case there is only one order in which we can visit the mandatory nodes [19]. However, if the graph has cycles the problem is NP complete since we can easily reduce the Hamiltonian Path problem [10, 5] to this problem.

Notice however that we can not trivially reduce Simple Path with mandatory nodes to Hamiltonian path. One could think that optional nodes (i.e. nodes that are not mandatory) can be eliminated in favor of new edges as a preprocessing step, which finds a path between each pair of mandatory nodes. However, the problem is that the paths that are precomputed may share nodes. This may lead to violations of the requirement that a node should be visited only once.

In figure 1, we illustrate this situation. Mandatory nodes are in solid lines. In the second graph we have eliminated the optional nodes by connecting each pair of mandatory nodes depending on whether there is a path between them. However, we observe that the second graph has a simple path going from node 1 to node 4 (visiting all the mandatory nodes) while the first one does not. Indeed, the simple path in the second graph is not a valid solution to the original problem since it implies that node 3 is visited twice.

The other reason that makes the elimination of optional nodes difficult is that finding k pairwise disjoint paths between k pairs of nodes $\langle s_1, d_1\rangle$,$\langle s_2, d_2\rangle$,...,$\langle s_k, d_k\rangle$ is NP complete [20].

In general, we can say that the set of optional nodes that can be used when going from a mandatory node a to a mandatory node b depends on the path that has been traversed before reaching a. This is because the optional nodes used in the path going from the source to a can not be used in the path going from a to b.

From our experimental measurements in Section 4, we observe that the suitability of *Reachability* for dealing with Simple Path with mandatory nodes is based on the following aspects:

- The strong pruning that *Reachability* performs. Due to the computation of cut nodes and bridges (i.e., nodes and edges that are present in all the paths going from a given node to another), *Reachability* is able to discover non-viable successors early on.

- The information that *Reachability* provides for implementing smart labeling strategies. By labeling strategy we mean the way the search tree is created, i.e., which constraint is used for branching. *Reachability* associates each node with the set of nodes that it reaches. This information can be used to guide the search in a smart way. For instance, one of our observations is that, when choosing first the node i that reaches the most nodes and selecting as a successor of i first a node that i reaches, we obtain paths that minimize the use of optional nodes.

An additional feature of *Reachability* is its suitability for imposing ordering constraints among mandatory nodes (which is a common issue in routing problems). In fact, it might be the case that we have to visit the nodes of the graph in a particular (partial) order. Taking into account that a node a reaches a node b if there is a path going from node a to node b, we force a node i to be visited before a node j by imposing that i reaches j and j does not reach i. We have performed experiments that show that *Reachability* takes the most advantage of this information to avoid branches in the search tree with no solution.

The structure of the paper is as follows: first, we introduce *Reachability* by presenting its semantics and deriving pruning rules in a systematic way. Then, we show how we can model Simple Path with mandatory nodes in terms of *Reachability*. Finally, we show examples that demonstrate the performance of *Reachability* for this type of problem, and elaborate on some works that are related to our approach.

2 The reachability propagator

2.1 Reachability constraint

The Reachability Constraint is defined as follows:

$$Reachability(g, source, rn, cn, be) \equiv \forall_{i \in N}. \begin{array}{l} rn(i) = Reach(g,i) \wedge \\ cn(i) = CutNodes(g, source, i) \wedge \\ be(i) = Bridges(g, source, i) \end{array} \tag{1}$$

Where:

- g is a graph whose set of nodes is a subset of N.
- $source$ is a node of g.
- $rn(i)$ is the set of nodes that i reaches.
- $cn(i)$ is the set of nodes appearing in all paths going from $source$ to i.
- $be(i)$ is the set of edges appearing in all paths going from $source$ to i.
- *Reach*, *Paths*, *CutNodes* and *Bridges* are functions that can be formally defined as follows:

$$j \in Reach(g,i) \leftrightarrow \exists_p . p \in Paths(g,i,j) \tag{2}$$

$$p \in Paths(g,i,j) \leftrightarrow \begin{array}{l} p = \langle k_1, ..., k_h \rangle \in nodes(g)^h \wedge k_1 = i \wedge k_h = j \wedge \\ \forall_{1 \le f < h} . \langle k_f, k_{f+1} \rangle \in edges(g) \end{array} \tag{3}$$

$$k \in CutNodes(g,i,j) \leftrightarrow \forall_{p \in Paths(g,i,j)} . k \in nodes(p) \tag{4}$$

$$e \in Bridges(g,i,j) \leftrightarrow \forall_{p \in Paths(g,i,j)} . e \in edges(p) \tag{5}$$

The above definition of *Reachability* implies the following properties which are crucial for the pruning that *Reachability* performs. These properties define relations between the functions rn, cn, be, nodes and edges. These relations can then be used for pruning, as we show in section 2.2.

1. If $\langle i, j \rangle$ is an edge of g, then i reaches j.

$$\forall_{\langle i,j \rangle \in edges(g)} . j \in rn(i) \tag{6}$$

2. If i reaches j, then i reaches all the nodes that j reaches.

$$\forall_{i,j,k \in N}.j \in rn(i) \wedge k \in rn(j) \rightarrow k \in rn(i) \tag{7}$$

3. If $source$ reaches i and j is a cut node between $source$ and i in g, then j is reached from $source$ and j reaches i:

$$\forall_{i,j \in N}.i \in rn(source) \wedge j \in cn(i) \rightarrow j \in rn(source) \wedge i \in rn(j) \tag{8}$$

4. Reached nodes, cut nodes and bridges are nodes and edges of g:

$$\forall_{i \in N}.rn(i) \subseteq nodes(g) \quad (9) \qquad \forall_{i \in N}.cn(i) \subseteq nodes(g) \quad (10) \qquad \forall_{i \in N}.be(i) \subseteq edges(g) \quad (11)$$

2.2 Pruning rules

We implement the constraint in Equation 1 with the propagator

$$Reachability(G, Source, RN, CN, BE) \tag{12}$$

In this propagator we have that:

- G is a graph variable (i.e., a variable whose domain is a set of graphs [8]). The upper bound of G ($max(G)$) is the greatest graph to which G can be instantiated, and its lower bound ($min(G)$) is the smallest graph to which G can be instantiated. So, $i \in nodes(G)$ means $i \in nodes(min(G))$ and $i \notin nodes(G)$ means $i \notin nodes(max(G))$ (the same applies for edges). In what follows, $\{\langle N_1, E_1\rangle \# \langle N_2, E_2\rangle\}$ will denote a graph variable whose lower bound is $\langle N_1, E_1\rangle$ and upper bound is $\langle N_2, E_2\rangle$. I.e., if $g = \langle n, e\rangle$ is the graph that G approximates, then $N_1 \subseteq n \subseteq N_2$ and $E_1 \subseteq e \subseteq E_2$.
- $Source$ is an integer representing the source in the graph.
- $RN(i)$ is a Finite Integer Set (FS) [11] variable associated with the set of nodes that can be reached from node i. The upper bound of this variable ($max(RN(i))$) is the set of nodes that could be reached from node i (i.e., nodes that are not in the upper bound are nodes that are known to be unreachable from i). The lower bound ($min(RN(i))$) is the set of nodes that are known to be reachable from node i. In what follows $\{S_1 \# S_2\}$ will denote a FS variable whose lower bound is the set S_1 and upper bound is the set S_2.
- $CN(i)$ is a FS variable associated with the set of nodes that are included in every path going from $Source$ to i.
- $BE(i)$ is a FS variable associated with the set of edges that are included in every path going from $Source$ to i.

The definition of *Reachability* and its derived properties give place to a set of propagation rules. We show here the most representative ones. The others are given in [16]. A propagation rule is defined as $\frac{C}{A}$ where C is a condition and A is an action. If C is true, the pruning defined by A can be performed.

- From (6) $\forall_{\langle i,j\rangle \in edges(g)}.j \in rn(i)$ we obtain:

$$\frac{\langle i, j\rangle \in edges(min(G))}{j \in min(RN(i))} \tag{13}$$

- From (7) $\forall_{i,j,k \in N}. j \in rn(i) \wedge k \in rn(j) \rightarrow k \in rn(i)$ we obtain:

$$\frac{j \in min(RN(i)) \wedge k \in min(RN(j))}{k \in min(RN(i))} \tag{14}$$

- From (8)$\forall_{i,j \in N}. i \in rn(source) \wedge j \in cn(i) \rightarrow j \in rn(source) \wedge i \in rn(j)$ we obtain:

$$\frac{i \in min(RN(Source)) \wedge j \in min(CN(i))}{j \in min(RN(Source))} \tag{15}$$

$$\frac{i \in min(RN(Source)) \wedge j \in min(CN(i))}{i \in min(RN(j))} \tag{16}$$

- From (1)$\forall_{i \in N}. rn(i) = Reach(g, i)$ we obtain:

$$\frac{j \notin Reach(max(G), i)}{j \notin max(RN(i))} \tag{17}$$

- From (1)$\forall_{i \in rn(source)}. cn(i) = CutNodes(g, source, i)$ we obtain:

$$\frac{j \in CutNodes(max(G), Source, i)}{j \in min(CN(i))} \tag{18}$$

- From (1)$\forall_{i \in rn(source)}. be(i) = Bridges(g, source, i)$ we obtain:

$$\frac{e \in Bridges(max(G), Source, i)}{e \in min(BE(i))} \tag{19}$$

- From (9) $\forall_{i \in N}. rn(i) \subseteq nodes(g)$, (10)$\forall_{i \in N}. cn(i) \subseteq nodes(g)$ and (11)$\forall_{i \in N}. be(i) \subseteq edges(g)$ we obtain:

$$\frac{k \in min(RN(i))}{k \in nodes(min(G))} \quad (20) \qquad \frac{k \in min(CN(i))}{k \in nodes(min(G))} \quad (21) \qquad \frac{e \in min(BE(i))}{e \in edges(min(G))} \quad (22)$$

2.3 Implementation of *Reachability*

Reachability has been implemented using a message passing approach [21] on top of the multi-paradigm programming language Oz [12]. In [15], we discuss the implementation of *Reachability* in detail. In this section we will simply refer to some of the functions that are used in the implementatin of *Reachability*:

In our pruning rules we have three functions:

- *Reach* that is $O(N + E)$ since it is basically a call to *DFS* [5].

- *CutNodes* whose algorithm is based on the following defi nition:

$$k \in CutNodes(g, i, j) \leftrightarrow j \notin Reach(RemoveNode(g, k), i) \tag{23}$$

  So, checking whether a node is a cut node is $O(N + E)$. Notice that we assume that $RemoveNode$ returns the same graph when $k \notin nodes(g)$.

- *Bridges* whose algorithm is based on the following definition:

$$e \in Bridges(g, i, j) \leftrightarrow j \notin Reach(RemoveEdge(g, e), i) \qquad (24)$$

  So, checking whether an edge is a bridge is $O(N + E)$. Notice that we assume that $RemoveEdge$ returns the same graph when $e \notin edges(g)$.

The computation of cut nodes and bridges in each propagation step implies running *DFS* per each potential cut node and per each potential bridge. However, we take advantage of the fact that the cut nodes and the bridges are part of the tree returned by *DFS* (assuming that the destination node is reached from the source). In fact, this means that both the computation of cut nodes and the computation of bridges have the same complexity since the number of edges in the DFS tree is proportional to the number of nodes. I.e., updating $CN\ /BE$ after removing a set of edges from the upper bound of G is $O(N * (N + E))$ [1].

As explained in [15], we do not compute cut nodes and bridges each time an edge is removed since this certainly leads to a considerably amount of unnecessary computation. This is due to the fact that the set of cut nodes/bridges evolves monotonically. What we actually do is to consider all the removals at once and make one computation of cut nodes and bridges per set of edges removed.

3 Solving *Simple Path with mandatory nodes* with *Reachability*

In this section we will elaborate on the important role that *Reachability* can play in solving Simple Path with mandatory nodes. This problem consists in finding a simple path in a directed graph containing a set of mandatory nodes. A simple path is a path where each node is visited once. I.e., given a directed graph g, a source node $source$, a destination node $dest$, and a set of mandatory nodes $mandnodes$, we want to find a path in g that goes from $source$ to $dest$, going through $mandnodes$ and visiting each node only once.

The contribution of *Reachability* consists in discovering nodes/edges that are part of the path early on. This information is obtained by computing the cut nodes and bridges in each labeling step. Let us consider the following two cases [2]:

- Consider the graph variable on the left of Figure 2. Assume that node 1 reaches node 9. This information is enough to infer that node 5 belongs to the graph, node 1 reaches node 5, and node 5 reaches node 9.

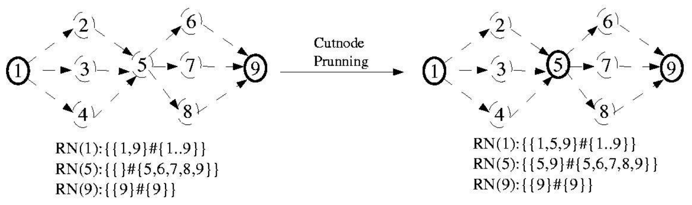

Figure 2: Discovering cut nodes

[1]Notice that the notion of cut node that we have presented does not correspond to the notion of articulation point. An articulation point is a node whose removal increases the number of strongly connected components. A cut node, in our definition, does not necessarily have this property.

[2]In Figures 2 and 3, nodes and edges that belong to the lower bound of the graph variable are in solid line. For instance, the graph variable on the left side of Figure 2 is a graph variable whose lower bound is the graph $\langle\{1, 5\}, \emptyset\rangle$, and whose upper bound is the graph $\langle\{1, 2, 3, 4, 5, 6, 7, 8, 9\}, \{\langle 1, 2\rangle, \langle 1, 3\rangle, \langle 1, 4\rangle, \langle 2, 5\rangle, \langle 3, 5\rangle, \langle 4, 5\rangle, \langle 5, 6\rangle, \langle 5, 7\rangle, \langle 5, 8\rangle, \langle 6, 9\rangle, \langle 7, 9\rangle, \langle 8, 9\rangle\}\rangle$.

- Consider the graph variable on the left of Figure 3. Assume that node 1 reaches node 5. This information is enough to infer that edges $\langle 1,2\rangle$, $\langle 2,3\rangle$,$\langle 3,4\rangle$ and $\langle 4,5\rangle$ are in the graph, which implies that node 1 reaches nodes 1,2,3,4,5, node 2 at least reaches nodes 2,3,4,5, node 3 at least reaches nodes 3,4,5 and node 4 at least reaches nodes 4,5.

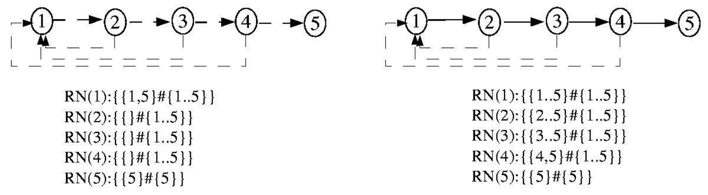

Figure 3: Discovering bridges

Notice that Hamiltonian Path (i.e., the problem where we have to find a simple path between two nodes containing all the nodes of the graph [10, 5]) can be reduced to Simple Path with mandatory nodes by defining the set of mandatory nodes as $nodes(g) - \{source, dest\}$.

The above definition of Simple Path with mandatory nodes can be formally defined as follows:

$$SPMN(g, source, dest, mandnodes, p) \leftrightarrow \begin{array}{l} p \in Paths(g, source, dest) \wedge \\ NoCycle(p) \wedge \\ mandnodes \subset nodes(p) \end{array} \tag{25}$$

$SPMN$ stands for "Simple Path with mandatory nodes". $NoCycle(p)$ states that p is a simple path (i.e., a path where no node is visited twice). This definition of Simple Path with mandatory nodes implies the following property:

$$Reachability(p, source, rn, cn, be) \wedge dest \in rn(source) \wedge cn(dest) \supseteq mandnodes \tag{26}$$

This is because the destination is reached by the source and the path contains the mandatory nodes. This derived property and the fact that we can implement $SPMN$ in terms of the *AllDiff* constraint [17] and the *NoCycle* constraint [4] suggest the two approaches for Simple Path with mandatory nodes summarized in Table 1 (which are compared in the next section). In the first approach, we basically consider *AllDiff* and *NoCycle*. In the second approach we additionally consider *Reachability*.

Notice that, even though the computation of bridges plays a crucial role in the pruning that *Reachability* performs, we do not use the be argument in the second approach. In fact be can play an important role in solving Constrained Euler Path problems (i.e., problems where the objective is to find a path visiting a set of edges by respecting some additional constraints).

Approach 1	**Approach 2**
$SPMN(g, source, dest, mandnodes, p)$	$SPMN(g, source, dest, mandnodes, p)$ $Reachability(p, source, rn, cn, be)$ $dest \in rn(source)$ $cn(dest) \supseteq mandnodes$

Table 1: Two approaches for solving Simple Path with mandatory nodes

4 Experimental results

In this section we present a set of experiments that show that *Reachability* is suitable for Simple Path with mandatory nodes. I.e., in our experiments *Approach 2* (in Table 1) outperforms *Approach 1*. These experiments also show that Simple Path with mandatory nodes tends to be harder when the number of optional nodes increases if they are uniformly distributed in the graph. We have also observed that the labeling strategy that we implement with *Reachability* tends to minimize the use of optional nodes (which is a common need when the resources are limited).

In Table 2, we define the instances on which we made the tests of Table 4. The id of the destination is also the size of the graph. The column Order is true for the instances whose mandatory nodes are visited in the order given. Notice that SPMN_52Order_b has no solution. The time, in Table 4, is measured in seconds. The number of failures means the number of failed alternatives tried before getting the solution.

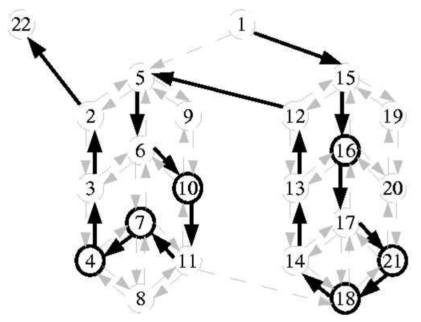

Figure 4: SPMN_22:A path from 1 to 22 visiting 4 7 10 16 18 21

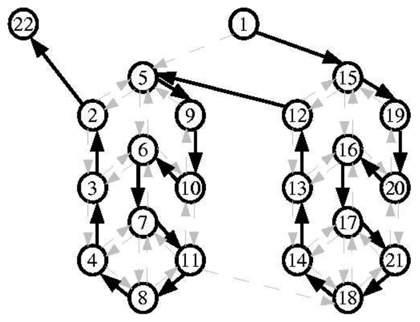

Figure 5: SPMN_22full:A path from 1 to 22 visiting all the nodes

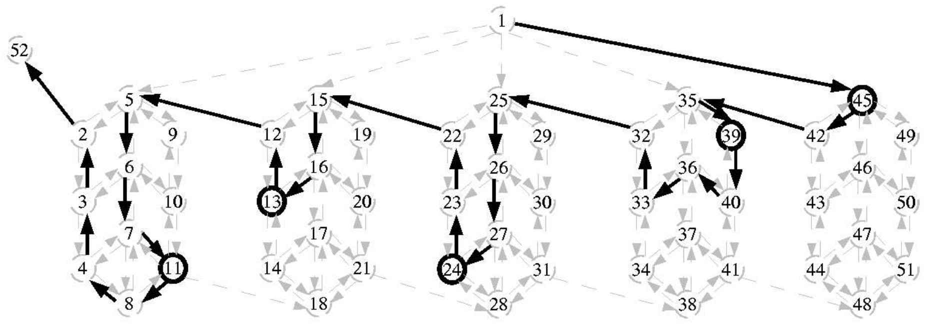

Figure 6: SPMN_52a:A path from 1 to 52 visiting 11 13 24 39 45

In our experiments, we have made five types of tests: using *SPMN* without *Reachability* (column "SPMN"), using *SPMN* and *Reachability* but without computing cut nodes nor bridges (column "SPMN+R"), using *SPMN* and *Reachability* but without computing bridges (column "SPMN+R+CN"), using *SPMN* and *Reachability* but without computing cut nodes (column "SPMN+R+BE"), and using *SPMN* and *Reachability* (column "SPMN+R+CN+BE").

As it can be observed in Table 4, we were not able to get a solution for SPMN_22 in less than 30 minutes

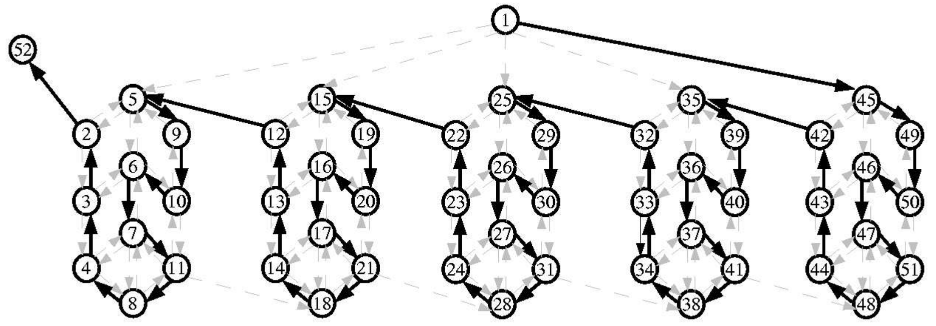

Figure 7: SPMN_52full:A path from 1 to 52 visiting all the nodes

Name	Figure	Source	Destination	Mand. Nodes	Order
SPMN_22	4	1	22	4 7 10 16 18 21	false
SPMN_22full	5	1	22	all	false
SPMN_52a	6	1	52	11 13 24 39 45	false
SPMN_52b	6	1	52	4 5 7 13 16 19 22 24 29 33 36 39 44 45 49	false
SPMN_52full	7	1	52	all	false
SPMN_52Order_a	6	1	52	45 39 24 13 11	true
SPMN_52Order_b	6	1	52	11 13 24 39 45	true

Table 2: Simple Path with mandatory nodes instances

Opt. Nodes	Failures	Time
5	30	89
10	42	129
15	158	514
20	210	693
25	330	1152
32	101	399
37	100	402
42	731	3518
47	598	3046

Table 3: Performance with respect to optional nodes

without using *Reachability*. In fact, we did not even try to solve SPMN_52b without it. However, even though the number of failures is still inferior, the use of *Reachability* does not save too much time when dealing with mandatory nodes only. This is due to the fact that we are basing our implementation of *SPMN* on two things: the use *AllDiff* [17] (that lets us effi ciently remove branches when there is no possibility of associating different successors to the nodes) and the use *NoCycle* [4] (that avoids re-visiting nodes).

The reason why *SPMN* does not perform well with optional nodes is because we are no longer able to impose the global *AllDiff* constrain on the successors of the nodes since we do not know a priori which nodes are going to be used. In fact, one thing that we observed is that the problem tends to be harder to solve when the number of optional nodes increases. In Table 3, all the tests were performed using *Reachability* on the graph of 52 nodes.

Even though, in SPMN_22, the benefi t caused by the computation of bridges is not that signifi cant, we were not able to obtain a solution for SPMN_52b in less than 30 minutes, while we obtained a solution in 402 seconds by computing bridges. So, even though the computation of bridges is costly, that computation pays off in most of the cases. Nevertheless, cut nodes should be computed in order to profi t from the computation of bridges.

Problem		SPMN		SPMN+R		SPMN+R+CN		SPMN+R+BE		SPMN+R+CN+BE	
Instance	**Figure**	**Failures**	**Time**	**Failures**	**Time**	**Failures**	**Time**	**Failures**	**Time**	**Failures**	**Time**
SPMN_22	4	+130000	+1800	91	6.81	40	6.55	70	13.76	13	4.45
SPMN_22full	5	213	1.44	19	0.95	0	0.42	19	2.76	0	1.22
SPMN_52b	-	-	-	+900	+1800	+700	+1800	+1000	+1800	100	402
SPMN_52full	7	3012	143	774	765	3	8.51	+700	1800	3	45.03
SPMN_52Order_a	6	+12000	+1800	51	46.33	55	81	27	97	16	57.07
SPMN_52Order_b	-	+12000	+1800	+1500	+1800	81	157	+400	+1800	41	117

Table 4: Simple Path with mandatory nodes tests

4.1 Labeling strategy

Reachability provides interesting information for implementing smart labeling strategies due to the fact that it associates each node with the set of nodes that it reaches. This information can be used to guide the search in a smart way. For instance, we observed that, when choosing first the node that reaches the most nodes, we obtain paths that minimize the use of optional nodes (as it can be observed in 6).

4.2 Imposing order on nodes

An additional feature of *Reachability* is the suitability for imposing dependencies on nodes (which is a common issue in routing problems). In fact, it might be the case that we have to visit the nodes of the graph in a particular (partial) order.

Our way of forcing a node i to be visited before a node j is by imposing that i reaches j and j does not reach i. The tests on the instances SPMN_52Order_a and SPMN_52Order_b show that *Reachability* takes the most advantage of this information to avoid branches in the search tree with no solution. Notice that we are able to solve SPMN_52Order_a (which is an extension of SPMN_52a) in 57.07 seconds. We are also able to detect the inconsistency of SPMN_52Order_b in 117 seconds.

5 Related work

- The cycle constraint of CHIP [1, 2] $cycle(N, [S_1, ..., S_n])$ models the problem of finding N distinct circuits in a directed graph in such a way that each node is visited exactly once. Certainly, Hamiltonian Path can be implemented using this constraint. In fact, [2] shows how this constraint can be used to deal with the Euler knight problem (which is an application of Hamiltonian Path). However, this constraint only covers the case where we are to visit all the nodes of the graph, which is a specific case of Simple Path with mandatory nodes.

- [19] suggests some algorithms for discovering mandatory nodes and non-viable edges in directed acyclic graphs. These algorithms are extended by [3] in order to address directed graphs in general with the notion of strongly connected components and condensed graphs. Nevertheless, examples like SPMN_52a represent tough scenarios for this approach since almost all the nodes are in the same strongly connected component.

- CP(Graph) introduces a new computation domain focussed on graphs including a new type of variable, graph domain variables, as well as constraints over these variables and their propagators [7, 8]. CP(Graph) also introduces node variables and edge variables, and is integrated with the finite domain and finite set computation domain. Consistency techniques have been developed, graph constraints have been built over the kernel constraints and global constraints have been proposed. $Path(p, s, d, maxlength)$ is one of these global constraints. This constraint is satisfied if p is a simple path from s to d of length at most $maxlength$. Certainly, Simple Path with mandatory nodes can be implemented in terms of *Path*. However, we still have to compare the performance of *Reachability* with respect to this approach.

6 Conclusion and future work

We presented *Reachability*: a constrained path propagator that can be used for speeding up constrained path solver. After introducing its semantics and pruning rules, we showed how the use of *Reachability* can speed up a standard approach for dealing with Simple Path with mandatory nodes.

Our experiments show that the gain is increased with the presence of optional nodes. This is basically because we are no longer able to apply the global *AllDiff* since we do not know a priori which nodes participate in the path.

From our observations, we infer that the suitability of *Reachability* is based on the strong pruning that it performs and the information that it provides for implementing smart labeling strategies. We also found that *Reachability* is appropriate for imposing dependencies on nodes. Certainly, we still have to see whether our conclusions apply to other types of graphs.

It is important to remark that both the computation of cut nodes and the computation of bridges play an essential role in the performance of *Reachability*. The reason is that each one is able to prune when the other can not. Notice that Figure 2 is a context where the computation of bridges cannot infer anything since there is no bridge. Similarly, Figure 3 represents a context where the computation of bridges discovers more information than the computation of cut nodes.

A drawback of our approach is that each time we compute cut nodes and bridges from scratch, so one of our next tasks is to overcome this limitation. I.e., given a graph g, how can we use the fact that the set of cut nodes between i and j is s for recomputing the set of cut nodes between i and j after the removal of some edges? We believe that a dynamic algorithm for computing cut nodes and bridges will improve our performance in a radical way.

As mentioned before, the implementation of *Reachability* was suggested by a practical problem regarding mission planning in the context of an industrial project. Our future work will concentrate on making propagators like *Reachability* suitable for non-monotonic environments (i.e., environments where constraints can be removed). Instead of starting from scratch when such changes take place, the pruning previously performed can be used to repair the current pruning.

References

[1] N. Beldiceanu and E. Contejean. Introducing global constraints in chip, 1994.

[2] E. Bourreau. *Traitement de contraintes sur les graphes en programmation par contraintes*. Doctoral dissertation, Université Paris, Paris, France, 1999.

[3] H. Cambazard and E. Bourreau. Conception d'une contrainte globale de chemin. In *10e Journées nationales sur la résolution pratique de problèmes NP-complets (JNPC'04)*, pages 107–121, Angers, France, June 2004.

[4] Y. Caseau and F. Laburthe. Solving small TSPs with constraints. In *International Conference on Logic Programming*, pages 316–330, 1997.

[5] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[6] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[7] G. Dooms, Y. Deville, and P. Dupont. Constrained path finding in biochemical networks. In *5èmes Journées Ouvertes Biologie Informatique Mathématiques*, 2004.

[8] G. Dooms, Y. Deville, and P. Dupont. CP(Graph):introducing a graph computation domain in constraint programming. In *CP2005 Proceedings*, 2005.

[9] F. Focacci, A. Lodi, and M. Milano. Solving tsp with time windows with constraints. In *CLP'99 International Conference on Logic Programming Proceedings*, 1999.

[10] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the The Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[11] C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *CONSTRAINTS journal*, 1(3):191–244, 1997.

[12] Mozart Consortium. The Mozart Programming System, version 1.3.0, 2004. Available at *http://www.mozart-oz.org/*.

[13] T. Müller. *Constraint Propagation in Mozart*. Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2001.

[14] G. Pesant, M. Gendreau, J. Potvin, and J. Rousseau. An exact constraint logic programming algorithm for the travelling salesman with time windows, 1996.

[15] L. Quesada, P. Van Roy, and Y. Deville. Reachability: a constrained path propagator implemented as a multi-agent system. In *CLEI2005 Proceedings*, 2005.

[16] L. Quesada, P. Van Roy, and Y. Deville. The reachability propagator. Research Report INFO-2005-07, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2005.

[17] J. C. Régin. A filtering algorithm for constraints of difference in csps. In *In Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 362–367, 1994.

[18] C. Schulte. *Programming Constraint Services*. Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2000.

[19] M. Sellmann. *Reduction Techniques in Constraint Programming and Combinatorial Optimization*. Doctoral dissertation, University of Paderborn, Paderborn, Germany, 2002.

[20] Y. Shiloach and Y. Perl. Finding two disjoint paths between two pairs of vertices in a graph. *Journal of the ACM*, 1978.

[21] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 2004.