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 Distribution. In this paper, we define a propagator which we call *Reachability* that implements a generalized reachability constraint on a graph. The *Reachability* constraint has two parts. First, it implements a relation between each node of g (the graph that is being constrained) and the set of nodes that it reaches. Second, given a source node $source$ in g, *Reachability* associates each pair of nodes $\langle source, i\rangle$ with the set of nodes and the set of edges that are included in all paths going from $source$ to i.
We show the effectiveness of our *Reachability* propagator by applying it to the Hamiltonian Path problem. 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 distribution 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 [PGPR96][CL97] [FLM99] and Bioinformatics[DDD04].

An approach to solve this problem is by using Concurrent Constraint Programming (CCP) [Sch00],[MÖ1]. In CCP, we solve the problem by interleaving two processes: propagation and distribution. 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 Distribution, 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[Dec03]. In this paper, we define a propagator which we call *Reachability* that implements a generalized reachability constraint on a graph. The *Reachability* constraint has two parts. First, it implements a relation between each node of g (the graph that is being constrained) and the set of nodes that it reaches. Second, given a source node *source* in g, *Reachability* associates each pair of nodes $\langle source, i\rangle$ with the set of nodes and the set of edges that are included in all paths going from *source* to i.

Our contribution is a propagator that is suitable for solving Hamiltonian Path with optional nodes (i.e., we are not forced to visit all the nodes of the graph but a subset of them). 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 [Sel02]. However, if the graph has cycles the problem becomes NP complete since we can easily reduce the standard version of Hamiltonian Path [GJ79][CLR90] to this problem.

From our experimental measurements in Section 4, we observe that the suitability of *Reachability* for dealing with Hamiltonian Path with optional 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. It is important to remark that *Reachability* is a complete propagator (i.e., it prunes all the non-valid values that it is supposed to prune under a given partial instantiation of its arguments).
- The information that *Reachability* provides for implementing smart distribution strategies. By distribution strategy we mean the way the search tree is created, i.e., which constraint is used to for branching. *Reachability* associates each node with the set of nodes that 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 that reaches the most nodes (i) 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. We force a node i to be visited first than a node j by imposing that i reaches j and j reaches 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 Hamiltonian Path with optional nodes in terms of *Reachability*. Finally, we show examples that demonstrate the suitability and the performance of *Reachability* for this type of problem, and we also elaborate on some alternatives that could radically improve the performance of *Reachability*.

2 Reachability propagator

Reachability is an example of a constrained path propagator. On the one hand, *Reachability* establishes a relation between each node of g and the set of nodes that it reaches. On the other hand, given a source node *source* in g, *Reachability* associates each pair of nodes $\langle source, i\rangle$ with the set of nodes and the set of edges that are included in all paths going from *source* to i.

2.1 Reachability Constraint

The Reachability Constraint is the following:

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

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:

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

$$p \in Paths(g, i, j) \leftrightarrow \begin{cases} if\ i = j\ then\ i \in nodes(g) \wedge p = \epsilon \\ if\ i \neq j\ then\ \exists_{k,p'} . p = \langle i, k\rangle \# p' \wedge \\ \qquad\qquad \langle i, k\rangle \in edges(g) \wedge \\ \qquad\qquad p' \in Paths(g, k, j) \end{cases} \quad (3)$$

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

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

The above definition of *Reachability* implies the following properties which are crucial for the pruning that *Reachability* perform:

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, 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 i reaches j and k is a cut node between i to j in g, then k is reached from i and k reaches j:

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

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

$$\forall_{i \in N}.rn(i) \subseteq nodes(g) \tag{9}$$

$$\forall_{i \in N}.cn(i) \subseteq nodes(g) \tag{10}$$

$$\forall_{i \in N}.be(i) \subseteq edges(g) \tag{11}$$

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[DDD04] whose upper bound ($max(G)$) is the greatest graph to which G can be instantiated, and 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 $\langle N_2, E_2 \rangle$.
- *Source* is an integer representing the source in the graph.
- $RN(i)$ is a Finite Integer Set (FS) [DKH+99] 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 S_1 and upper bound 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. The pruning rules are obtained by considering the following rewriting rules:

$$\frac{P_1 \wedge P_2 \wedge ... \wedge P_n \rightarrow R}{\neg R \wedge P_1 \wedge ... \wedge P_{i-1} \wedge P_{i+1} ... \wedge P_n \rightarrow \neg P_i} \tag{13}$$

$$\frac{P \rightarrow Q_1 \wedge Q_2 \wedge ... \wedge Q_i \wedge ... \wedge Q_n}{P \rightarrow Q_i} \tag{14}$$

$$\frac{i \in S}{i \in min(S)} \tag{15} \qquad \frac{i \notin S}{i \notin max(S)} \tag{16}$$

$$\frac{S_1 = S_2}{\begin{array}{l} min(S_1) = min(S_2) \\ max(S_1) = max(S_2) \end{array}} \tag{17} \qquad \frac{S_1 \subseteq S_2}{\begin{array}{l} min(S_1) \subseteq min(S_2) \\ max(S_1) \subseteq max(S_2) \end{array}} \tag{18}$$

The pruning rules are the following:

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

$$\frac{\langle i,j \rangle \in edges(min(G))}{j \in min(RN(i))} \tag{19}$$

$$\frac{j \notin max(RN(i))}{\langle i,j \rangle \notin edges(max(G))} \tag{20}$$

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

$$\frac{j \in min(RN(i)) \wedge k \in min(RN(j))}{k \in min(RN(i))} \tag{21}$$

$$\frac{k \notin max(RN(i)) \wedge j \in min(RN(i))}{k \notin max(RN(j))} \tag{22}$$

$$\frac{k \notin max(RN(i)) \wedge k \in min(RN(j))}{j \notin max(RN(i))} \tag{23}$$

- 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:

$$\frac{i \in min(RN(Source)) \wedge j \in min(CN(i))}{j \in min(RN(Source))} \tag{24}$$

$$\frac{j \notin max(RN(Source)) \wedge j \in min(CN(i))}{i \notin max(RN(Source))} \tag{25}$$

$$\frac{j \notin max(RN(Source)) \wedge i \in min(RN(Source))}{j \notin max(CN(i))} \quad (26)$$

$$\frac{i \in min(RN(Source)) \wedge j \in min(CN(i))}{i \in min(RN(j))} \quad (27)$$

$$\frac{i \notin max(RN(j)) \wedge j \in min(CN(i))}{i \notin max(RN(Source))} \quad (28)$$

$$\frac{i \notin max(RN(j)) \wedge i \in min(RN(Source))}{j \notin max(CN(i))} \quad (29)$$

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

$$\frac{j \notin Reach(max(G),i)}{j \notin max(RN(i))} \quad (30)$$

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

$$\frac{j \in CutNodes(max(G), Source, i)}{j \in min(CN(i))} \quad (31)$$

$$\frac{j \notin CutNodes(min(G), Source, i)}{j \notin max(CN(i))} \quad (32)$$

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

$$\frac{e \in Bridges(max(G), Source, i)}{e \in min(BE(i))} \quad (33)$$

$$\frac{e \notin Bridges(min(G), Source, i)}{e \notin max(BE(i))} \quad (34)$$

– 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:

$$\frac{k \in min(RN(i))}{k \in nodes(min(G))} \quad (35) \qquad \frac{k \notin nodes(max(G))}{k \notin max(RN(i))} \quad (36)$$

$$\frac{k \in min(CN(i))}{k \in nodes(min(G))} \quad (37) \qquad \frac{k \notin nodes(max(G))}{k \notin max(CN(i))} \quad (38)$$

$$\frac{e \in min(BE(i))}{e \in edges(min(G))} \quad (39) \qquad \frac{e \notin edges(max(G))}{e \notin max(BE(i))} \quad (40)$$

2.3 Complexity and Level of Consistency

In our pruning rules we have three functions:

- *Reach* that is $O(V + E)$ since it is basically a call to *DFS* [CLR90].
- *CutNodes* whose algorithm is based on the following definition:

$$k \in CutNodes(g, i, j) \leftrightarrow j \notin Reach(RemoveNode(g, k), i) \tag{41}$$

  So, checking whether a node is a cut node is $O(V + E)$. Notice that we assume that *RemoveNode* returns the same graph when $k \notin nodes(g)$.
- *Bridges* whose algorithm is based on the following definition:

$$e \in Bridges(g, i, j) \leftrightarrow j \notin Reach(RemoveEdge(g, e), i) \tag{42}$$

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

Notice that *Reachability* is a complete propagator. This affirmation is based on the following facts:

- If i reaches j in $min(G)$ the presence of j in the lower bound of RN is ensured by the pruning rules derived from Property 7 and 8.
- If i does not reach j in $max(G)$ the removal of j from the upper bound of RN is ensured by Rule 34
- If i is a cut node, Rule 31 ensures the presence of it in the lower bound of CN.
- If i is not a cut node, Rule 32 ensures the removal of it from the upper bound of CN.
- If e is a bridge, Rule 33 ensures the presence of it in the lower bound of BE.
- If e is not a bridge, Rule 34 ensures the removal of it from the upper bound of BE.

3 Solving *Hamiltonian Path* with *Reachability*

In this section we will elaborate on the important role that *Reachability* can play in solving Hamiltonian Path (i.e., finding a path in a directed graph where all the nodes are visited [GJ79][CLR90]). The contribution of *Reachability* consists in discovering nodes/edges that are part of the Hamiltonian path early on. This information is obtained by computing the cut nodes and bridges in each distribution step. Let us consider the following two cases:

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

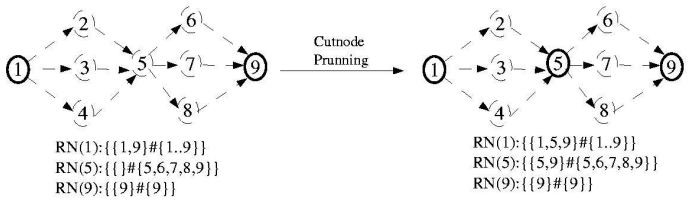

Fig. 1. Discovering cut nodes

– Consider the graph variable on the left of Figure 2. 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.
  - node 4 at least reaches nodes 4,5.

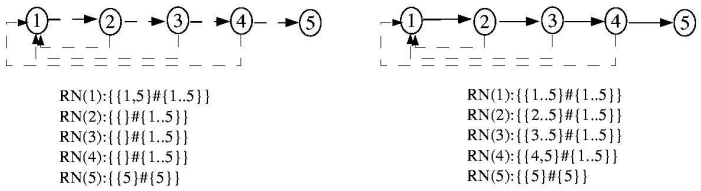

Fig. 2. Discovering bridges

Consider the following definition of Hamiltonian Path with optional nodes: given a directed graph g, a source node *source*, a destination node *dest*, and a set of mandatory nodes *mandnodes*, find a path in g that goes from *source* to *dest*, going through *mandnodes* and visiting each node only once.

Notice that the standard Hamiltonian Path (i.e., the one where we have to visit all the nodes) can be reduced to this problem by defining *mandnodes* as $nodes(g) - \{source, dest\}$.

The above definition of Hamiltonian Path with optional nodes can be formally defined as follows:

$$Ham(g, source, dest, mandnodes, p) \leftrightarrow$$
$$p \in Paths(g, source, dest)$$

$NoCycle(p)$
$mandnodes \subset nodes(p)$

Where:

$$NoCycle(p) \leftrightarrow NoCycle'(p, \emptyset)$$

$$NoCycle'(p, ns) \leftrightarrow \begin{cases} p = \epsilon \\ \\ p = \langle i, j \rangle \# p' \wedge \\ ns' = \{i\} \cup ns \wedge \\ j \notin ns' \wedge \\ NoCycle'(p', ns') \end{cases} \quad (43)$$

Hamiltonian Path can be solved by using *AllDiff* [Rég94] and *NoCycle*[CL97]. It is however possible to state redundant constraints inducing a better pruning. We can easily show that:

$$Ham(g, source, dest, mandnodes, p) \rightarrow \begin{array}{l} Reachability(p, source, rn, cn, be) \wedge \\ dest \in rn(source) \wedge \\ cn(dest) \supseteq mandnodes \end{array} \quad (44)$$

since the destination is reached by the source and the path contains the mandatory nodes.

The two algorithms are summarized in Table 1 and compared in the next section. 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 algorithm. 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).

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

Table 1. Two approaches for solving Hamiltonian Path

4 Experimental Results

In this section we present a set of experiments that show that *Reachability* is suitable for Hamiltonian Path with optional nodes. I.e., in our experiments *Algo-*

rithm 2 (in Table 1) outperforms *Algorithm 1*. These experiments also show that Hamiltonian Path with optional 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 distribution 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 3. The column Order is true for the instances whose mandatory nodes are visited in the order given. The time, in Table 2 is measured in seconds. The number of failures means the number of failed alternatives tried before getting the solution.

Name	**Figure**	**Source**	**Destination**	**Mand. Nodes**	**Order**
Ham_22	3	1	22	4 7 10 16 18 21	false
Ham_22full	4	1	22	all	false
Ham_52a	5	1	52	11 13 24 39 45	false
Ham_52b	5	1	52	4 5 7 13 16 19 22 24 29 33 36 39 44 45 49	false
Ham_52full	6	1	52	all	false
Ham_52Order_a	5	1	52	45 39 24 13 11	true
Ham_52Order_b	5	1	52	11 13 24 39 45	true

Table 2. Hamiltonian Path instances

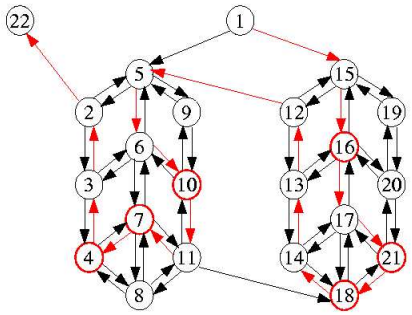

Fig. 3. Ham_22:A path from 1 to 22 visiting 4 7 10 16 18 21

Table 3 is a summary of the tests that we have performed. Notice that Ham_52Order_b has no solution. In our experiments, we have made five types of tests:

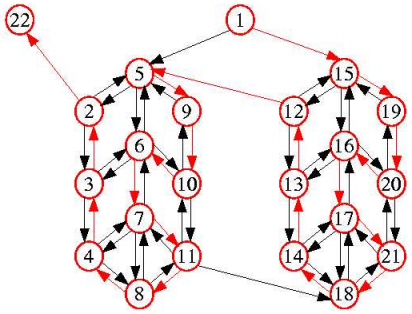

Fig. 4. Ham_22full:A path from 1 to 22 visiting all the nodes

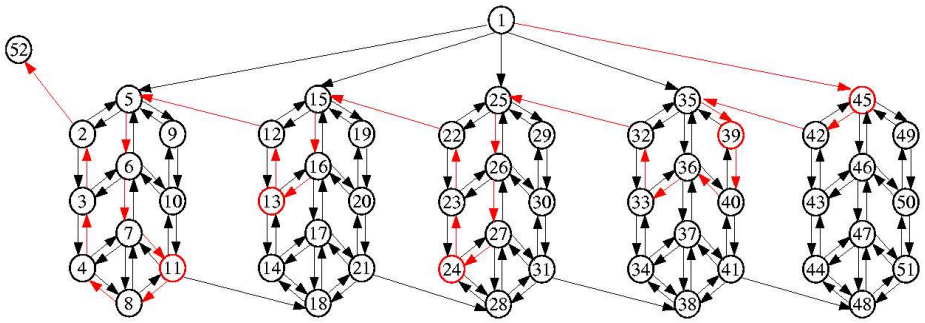

Fig. 5. Ham_52b:A path from 1 to 52 visiting 11 13 24 39 45

Problem		Ham		Ham+R+CN		Ham+R+BE		Ham+R		Ham+R+CN+BE	
Instance	Figure	Failures	Time	Failures	Time	Failures	Time	Failures	Time	Failures	Time
Ham_22	3	+130000	+1800	40	6.55	70	13.76	91	6.81	13	4.45
Ham_22full	4	213	1.44	0	0.42	19	2.76	19	0.95	0	1.22
Ham_52b	-	-	-	+700	+1800	+1000	+1800	+900	+1800	100	402
Ham_52full	6	3012	143	3	8.51	+700	1800	774	765	3	45.03
Ham_52Order_a	5	+12000	+1800	55	81	27	97	51	46.33	16	57.07
Ham_52Order_b	-	+12000	+1800	81	157	+400	+1800	+1500	+1800	41	117

Table 3. Hamiltonian Path tests

– Using *Ham* without *Reachability* (column "Ham").
– Using *Ham* and *Reachability* but without computing bridges (column "Ham+R+CN").

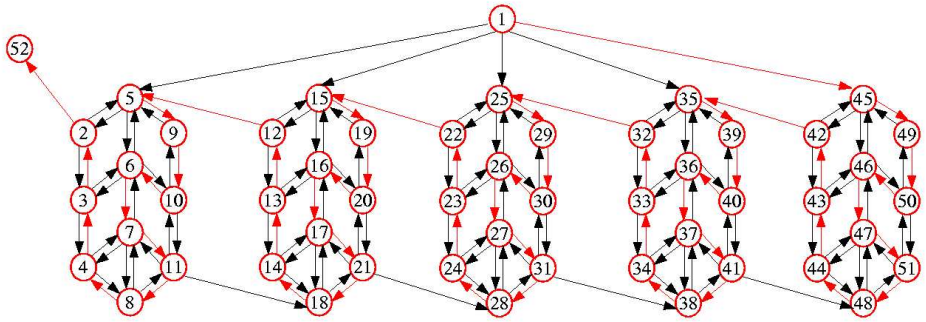

Fig. 6. Ham_52full:A path from 1 to 52 visiting all the nodes

- Using *Ham* and *Reachability* but without computing cut nodes (column "Ham+R+BE").
- Using *Ham* and *Reachability* but without computing cut nodes nor bridges (column "Ham+R").
- Using *Ham* and *Reachability* (column "Ham+R+CN+BE").

As it can be observed in Table 3, we were not able to get a solution for Ham_22 in less than 30 minutes without using *Reachability*. In fact, we did not even try to solve Ham_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 *Ham* on two things:

- The use *AllDiff* [Rég94] that lets us efficiently remove branches when there is no possibility of associating different successors to the nodes.
- The use *NoCycle* [CL97] that avoids re-visiting nodes.

The reason why *Ham* 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 4, all the tests were performed using *Reachability* on the graph of the 52 nodes.

Even though, in Ham_22, the benefit caused by the computation of bridges is not that significant, we were not able to obtain a solution for Ham_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 extremely costly (one order higher than the computation of cut nodes), that computation pays off in most of the cases.

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 4. Performance when dealing with optional nodes with *Reachability*

4.1 Implementation of *Reachability*

Reachability has been implemented using a message passing approach on top of the multi-paradigm programming language Oz [Moz04]. I.e., *Reachability* is a multi-agent system where agents interchange synchronous and asynchronous messages and their transition state functions rely on data flow and constraint programming primitives [VH03]. In fact, we have already found this approach quite appropriate for implementing global constraints [QGV03].

An important aspect of our implementation is the utilization of a batch approach for the computation of cut nodes and bridges. I.e., we do not compute cut nodes and bridges each time an edge is added/removed from the the lower/upper bound of the graph variable. Instead, we wait until having a certain amount of changes in the lower/upper bound for computing the cut nodes and bridges.

4.2 Distribution Strategy

Reachability provides interesting information for implementing smart distribution strategies due to that 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 Figure 5).

4.3 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 first than a node j is by imposing that i reaches j and j does not reach i. The tests on the instances Ham_52Order_a and Ham_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 Ham_52Order_a (which is an extension of Ham_52a) in 57.07 seconds. We are also able to detect the inconsistency of Ham_52Order_b in 117 seconds.

Our implementation of *NoCycle* maintains, for each node, a FS variable for keeping track of the reached nodes. I.e., *NoCycle* has its own *RN* because whenever it is known that i reaches j *NoCycle* imposes the constraint that i is not in the nodes that j reaches. So, we can use the same approach for dealing with orderings since the reached nodes sets are computed anyway.

The reason why *Reachability* does so well is because of the fact explained in figures 1 and 2. In particular, the reason why the computation of bridges is important is because it can infer orders on the nodes in cases where the computation of cut nodes can not (as shown in Figure 2).

5 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 Hamiltonian Path.

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 distribution 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 1 is a context where the computation of bridges cannot infer anything since there is no bridge. Similarly, Figure 2 represents a context where the computation of bridges discover 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?. So far, we have not been able to find a satisfactory answer to this question. But 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 con-

straints can be removed). Instead of starting from scratch when such changes take place, what we want is to use the pruning previously performed in order to repair the pruning.

References

[CL97] Yves Caseau and Francois Laburthe. Solving small TSPs with constraints. In *International Conference on Logic Programming*, pages 316–330, 1997.

[CLR90] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[DDD04] G. Dooms, Y. Deville, and P. Dupont. Constrained path finding in biochemical networks. In *5èmes Journées Ouvertes Biologie Informatique Mathématiques*, 2004.

[Dec03] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[DKH+99] Denys Duchier, Leif Kornstaedt, Martin Homik, Tobias Müller, Christian Schulte, and Peter Van Roy. *Finite Set Constraints*. December 1999. Available at `http://www.mozart-oz.org/`.

[FLM99] F. Focacci, A. Lodi, and M. Milano. Solving tsp with time windows with constraints. In *CLP'99 International Conference on Logic Programming Proceedings*, 1999.

[GJ79] Michael Garey and David Johnson. *Computers and Intractability: A Guide to the The Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[MÖ1] Tobias Müller. *Constraint Propagation in Mozart*. Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2001.

[Moz04] Mozart Consortium. The Mozart Programming System, version 1.3.0, 2004. Available at `http://www.mozart-oz.org/`.

[PGPR96] G. Pesant, M. Gendreau, J. Potvin, and J. Rousseau. An exact constraint logic programming algorithm for the travelling salesman with time windows, 1996.

[QGV03] L. Quesada, S. Gualandi, and P. Van Roy. Implementing a distributed shortest path propagator with message passing. In *2nd International Workshop on Multiparadigm Constraint Programming Languages (MultiCPL 2003), at the 9th International Conference on Principles and Practice of Constraint Programming (CP2003)*, 2003.

[Rég94] Jean Charles 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.

[Sch00] Christian Schulte. *Programming Constraint Services*. Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2000.

[Sel02] Meinolf Sellmann. *Reduction Techniques in Constraint Programming and Combinatorial Optimization*. Doctoral dissertation, University of Paderborn, Paderborn, Germany, 2002.

[VH03] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 2003.